Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation

Falou Ndoye

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE
Sciences et Technologie de l'Information, des Télécommunications et des Systèmes
INRIA Paris-Rocquencourt

DISCIPLINE : Génie Informatique

THÈSE DE DOCTORAT

présentée et soutenue publiquement le 03/04/2014

par

Falou **NDOYE**

Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation

Composition du jury :

Directeur de thèse :	Yves SOREL	Directeur de recherche (INRIA Paris-Rocquencourt)
Rapporteurs :	Emmanuel GROLLEAU	Professeur (ISAE-ENSMA Poitiers)
	Jean-Luc SCHARBARG	Professeur (INPT/ENSEEIHT Toulouse)
Examinateurs :	Alain MERIGOT	Professeur (Université Paris-Sud Orsay)
	Laurent GEORGE	Maître de conférences HDR (Université Marne-la-Vallée)

Résumé

Dans cette thèse nous étudions le problème d'ordonnancement temps réel multiprocesseur préemptif avec prise en compte du coût exact du système d'exploitation. Ce coût est formé de deux parties : une partie facile à déterminer, correspondant au coût de l'ordonnanceur et une partie difficile à déterminer, correspondant au coût de la préemption. Cette difficulté est due au fait qu'une préemption peut en engendrer une autre, pouvant ainsi créer un phénomène d'avalanche.

Dans un premier temps, nous avons étudié l'ordonnancement hors ligne multiprocesseur de **tâches indépendantes** avec prise en compte du coût exact de la préemption et proposé une analyse d'ordonnançabilité fondée sur une heuristique d'ordonnancement multiprocesseur. Cette heuristique utilise la stratégie d'ordonnancement multiprocesseur par partitionnement. Pour prendre en compte le coût exact de la préemption sur chaque processeur nous avons utilisé la condition d'ordonnançabilité proposée par Meumeu et Sorel. Cette condition d'ordonnançabilité pour des tâches à priorités fixes, est basée sur une opération binaire d'ordonnancement qui permet de compter le nombre exact de préemption et d'ajouter leur coût dans l'analyse d'ordonnançabilité des tâches. L'heuristique proposée permet de maximiser le facteur d'utilisation restant afin de répartir équitablement les tâches sur les processeurs et de réduire leur temps de réponse. Elle produit une table d'ordonnancement hors ligne.

Dans un second temps, nous avons étudié l'ordonnancement hors ligne multiprocesseur de **tâches dépendantes** avec prise en compte du coût exact de la préemption. Puisque la condition d'ordonnançabilité utilisée pour ordonnancer les tâches indépendantes ne s'applique qu'à des tâches à priorités fixes, elle ne permet pas de gérer les inversions de priorités que peuvent entraîner les tâches dépendantes. Nous avons donc proposé une nouvelle condition d'ordonnançabilité pour des tâches à priorités dynamiques. Elle prend en compte le coût exact de la préemption et les dépendances sans aucune perte de données. Ensuite en utilisant toujours la stratégie d'ordonnancement par partitionnement, nous avons proposé pour des tâches dépendantes une heuristique d'ordonnancement multiprocesseur qui réutilise cette nouvelle condition d'ordonnançabilité au niveau de chaque processeur. Cette heuristique d'ordonnancement prend en compte les coûts de communication inter-processeurs. Elle permet aussi de minimiser sur chaque processeur le makespan (temps total d'exécution) des tâches. Cette heuristique produit pour chaque processeur une table d'ordonnancement hors ligne contenant les dates de début et de fin de chaque tâches et de chaque commmunication inter-processeur.

En supposant que nous avons une architecture multiprocesseur de type dirigée par le temps (Time Trigger TT) pour laquelle tous les processeurs ont une référence de temps unique, nous avons proposé pour chacun des processeurs un

ordonnanceur en ligne qui utilise la table d'ordonnancement produite lors de l'ordonnancement hors ligne. Cet ordonnanceur en ligne a l'avantage d'avoir un coût constant et facile à déterminer de manière exacte. En effet il correspond uniquement au temps de lecture dans la table d'ordonnancement pour obtenir la tâche sélectionnée lors de l'analyse d'ordonnançabilité hors ligne, alors que dans les "ordonnanceurs classiques" en ligne ce coût correspond à mettre à jour la liste des tâches qui sont dans l'état prêt à l'exécution puis à sélectionner une tâche selon un algorithme, par exemple RM, DM, EDF, etc. Il varie donc avec le nombre de tâches prêtes à s'exécuter qui change d'une invocation à l'autre de l'ordonnanceur. C'est ce coût qui est utilisé dans les analyses d'ordonnançabilités évoquées ci-dessus. Un autre avantage est qu'il n'est pas nécessaire de synchroniser l'accès aux mémoires de données partagées par plusieurs tâches, car cette synchronisation a été déjà effectuée lors de l'analyse d'ordonnançabilité hors ligne.

Rémerciements

Je tiens à remercier Yves Sorel, mon directeur de thèse, de m'avoir accueilli comme stagiaire au sein de l'équipe-projet AOSTE et ensuite m'avoir donné l'opportunité de faire une thèse. Il a su guider mes travaux tout en me laissant mon autonomie.

Je remercie également Emmanuel Grolleau et Jean Luc-Scharbarg qui ont accepté d'être les rapporteurs de cette thèse. De même, un grand merci à Alain Merigot et Laurent George qui ont accepté de faire partie des membres du jury.

Je remercie aussi tous les membres de l'équipe-projet AOSTE pour les bons moments passés ensemble.

Enfin, mention spéciale à ma fammille et à mes amis qui m'ont soutenus tout au long de cette thèse.

Table des matières

Table des figures

clearemptydoublepage

Liste des tableaux

Introduction générale

Contexte

Cette thèse s'inscrit dans le cadre des recherches menées dans l'équipe-projet AOSTE (Analyse et Optimisation des Systèmes Temps réel Embarquée) d'INRIA Paris-Rocquencourt sur les systèmes distribués temps réel embarqués que l'on trouve dans les domaines applicatifs tels que l'automobile, l'avionique, le ferroviaire, etc. Ces systèmes étant critiques, il faut assurer que les contraintes temps réel et les contraintes de ressources sont satisfaites au risque de conséquences catastrophiques, par exemple de pertes humaines. Satisfaire ces contraintes temps réel et de ressources est d'autant plus difficile que les architectures cibles actuelles sont multicomposant (multiprocesseur, distribués, multi/many-core, parallèles). Les spécifications fonctionnelles et les simulations réalisées par les automaticiens conduisent à ce que les fonctions, qui deviendront des tâches temps réel, aient des périodes différentes, des dépendance entre elles, des contraintes d'échéance et de latence. Afin d'assurer que les conditions d'ordonnançabilités temps réel soient réellement respectées lorsqu'une application s'exécute en temps réel sur une architecture cible, il faut prendre en compte dans l'analyse d'ordonnançabilité le coût du système d'exploitation (OS pour "Operating System") qui réalise effectivement l'ordonnancement pendant la vie de l'application, mais qui est en général approximé, voire négligé. Ceci conduit au mieux à du gaspillage de ressources et au pire au non respect des contraintes temps réel lors du fonctionnement de l'application.

Objectifs

Le coût du système d'exploitation est composé de deux parties : une partie facile à déterminer, correspondant au coût de l'ordonnanceur et une partie difficile à déterminer, correspondant au coût de la préemption. Cette difficulté est due au fait qu'une préemption peut en engendrer une autre, pouvant ainsi créer un phénomène d'avalanche [Yom09]. Meumeu et Sorel [Yom09] ont proposé une

condition d'ordonnançabilité qui permet de prendre en compte le coût exact de la préemption, donc le coût de l'OS. Mais cette condition d'ordonnançabilité n'est applicable qu'à des tâches indépendantes à priorités fixes et sur une architecture monoprocesseur. Notre premier objectif est d'étendre cette condition d'ordonnançabilité pour étudier l'ordonnancement de tâches temps réel indépendantes sur une architecture multiprocesseur et en tenant compte du coût exact de la préemption. Les applications temps réel embarquées étant généralement composées de tâches dépendantes, notre deuxième objectif est d'étudier l'ordonnancement de tâches dépendantes sur une architecture multiprocesseur et toujours avec prise en compte du coût exact de la préemption. Pour cela nous allons étudier une nouvelle condition d'ordonnançabilité pour des tâches dépendantes, qui permet de prendre en compte le coût exact de la préemption puisque la condition d'ordonnançabilité proposée par Meumeu et Sorel est applicable uniquement pour des tâches indépendantes à priorités fixes et ne permet donc pas de gérer les inversions de priorités que peut entraîner l'ordonnancement de tâches dépendantes. Enfin notre troisième et dernier objectif est, à partir de l'analyse d'ordonnançabilité que nous avons proposée, d'utiliser la table d'ordonnancement déterminée hors ligne dans un nouveau type d'ordonnanceur en ligne prenant mieux en compte le coût de l'OS que les ordonnanceurs classiques.

Plan de la thèse

Ce manuscrit est constitué de trois parties.

La première partie est consacrée aux concepts de bases de l'ordonnancement temps réel pour faciliter la compréhension de la suite de la thèse et à l'état de l'art dans lequel on met l'accent sur les méthodes d'allocation des tâches dans l'ordonnancement multiprocesseur, sur la prise en compte du coût exact de la préemption et sur les dépendances de données.

La deuxième partie est consacrée à l'ordonnancement multiprocesseur de tâches indépendantes avec prise en compte du coût exact de la préemption. Dans le premier chapitre de cette partie on présente la condition d'ordonnançabilité proposée par Meumeu et Sorel qui permet de prendre en compte le coût exact de la préemption en monoprocesseur pour des tâches à priorités fixes. Le deuxième chapitre de cette deuxième partie présente l'étude d'ordonnancement multiprocesseur de tâches indépendantes qui étend la condition d'ordonnançabilité de Meumeu et Sorel en multiprocesseur.

La troisième partie est consacrée à l'ordonnancement multiprocesseur de tâches dépendantes. Dans le premier chapitre de cette partie, on étudie l'ordonnancement des tâches dépendantes en monoprocesseur en proposant une nouvelle condition d'ordonnançabilité qui permet d'ordonnancer des tâches dépendantes en prenant

en compte du coût exact de la préemption et les transferts de données entre tâches qui conduisent à des inversions de priorités. Dans le deuxième chapitre de cette partie, on présente l'étude d'ordonnancement multiprocesseur de tâches dépendantes qui étend cette nouvelle condition d'ordonnançabilité monoprocesseur en multiprocesseur tout en tenant compte le coût exact de la préemption et les coûts de communication inter-processeurs. Dans le troisième chapitre de cette partie, on présente l'ordonnanceur en ligne de chaque processeur qui utilise la table d'ordonnancement produite hors ligne, pour ce processeur.

Première partie

Concepts de base et État de l’art

Chapitre 1

Concepts de base

L'objectif de ce chapitre est de présenter les concepts de base de l'ordonnancement temps réel.

1.1 Système temps réel

1.1.1 Définition

Un système réactif doit réagir continûment aux stimuli venant d'un processus qu'il cherche à commander. Un système temps réel est un système réactif qui doit respecter des contraintes de temps. Un système temps réel doit être capable de traiter les informations venant du processus dans un délai qui ne nuit pas à la commande du processus. Réagir trop tard peut conduire à des conséquences catastrophiques pour le système lui-même ou le processus. Le respect des contraintes temporelles est la principale contrainte à satisfaire. La validité d'un système temps réel dépend non seulement des résultats du traitement effectué mais aussi de l'aspect temporel (un calcul juste mais hors délai est un calcul non valable). Dans un système temps réel, les évènements d'entrée sont produits par des capteurs et les évènements de sortie sont consommés par des actionneurs. La figure 1.1 donne une illustration d'un système réactif temps réel. On rencontre des applications temps réel dans le domaine de l'aéronautique, l'automobile, télécommunication, robotique, etc.

1.1.2 Classification des systèmes temps réel

La criticité des contraintes temporelles conduit à classifier les système temps réel en trois catégories suivantes :

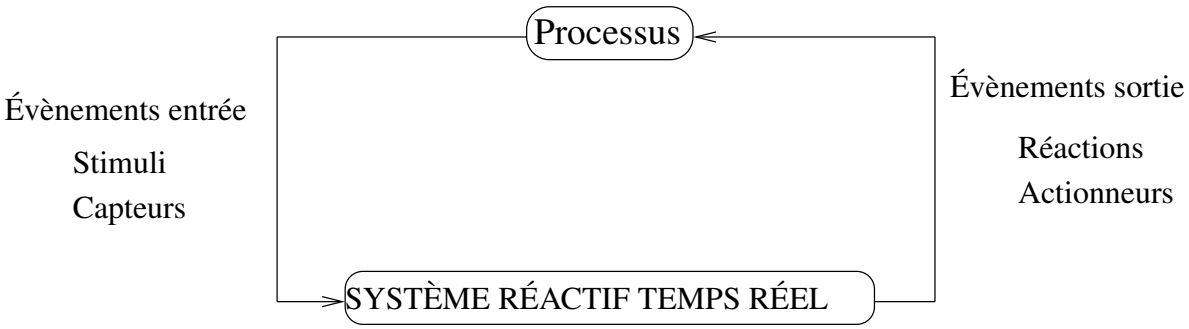

FIGURE 1.1 – Système réactif temps réel

- système temps réel strict : c'est un système soumis à des contraintes temporelles strictes, c'est-à-dire pour lequel la moindre faute temporelle peut avoir des conséquences humaines ou économiques catastrophiques. La plupart des applications dans les domaines avioniques, automobile, etc., sont temps réel strict;
- système temps réel souple : c'est un système soumis à des contraintes temporelles souples, un certain nombre de fautes temporelles peut être tolérées. On parle alors de qualité de service;
- système temps réel mixte : c'est un système soumis à des contraintes temporelles strictes et à des contraintes temporelles souples.

1.2 Tâche temps réel

Une tâche temps réel est formée d'un ensemble d'instructions pouvant s'exécuter en séquence sur un ou plusieurs processeurs et respecter des contraintes de temps. Dans la suite nous ferons l'hypothèse qu'une tâche ne s'exécutera pas en parallèle. Elle peut se répéter un nombre quelconque de fois, éventuellement infini. Chacune de ses exécutions est appelée instance ou travail ("job"). Un système temps réel est composé d'un ensemble de tâches temps réel soumises à des contraintes temps réel (voir la section 1.4).

Une tâche temps réel peut être :

- périodique : ses instances (exécutions) se répètent indéfiniment et il existe une durée constante entre deux activations d'instances successives appelée période,
- sporadique : ses instances (exécutions) se répètent indéfiniment et il existe une durée minimum entre deux instances successives,
- apériodique : il y a pas de corrélation entre deux instances successives.

1.3 Tâches périodiques

Le modèle de tâches périodiques classique dit de Liu et Layland de [LL73] est le plus utilisé dans la modélisation des systèmes temps réel. Ce modèle permet de définir plusieurs paramètres pour une tâche. Ces paramètres sont de deux types : paramètres statiques relatifs à la tâche elle-même et les paramètres dynamiques relatif à chaque instance de la tâche. Les paramètres statiques de base d'une tâche périodique $\tau_i = (r_i^1, C_i, D_i, T_i)$ sont :

- r_i^1 (release time) : date de première activation de la tâche τ_i, date à laquelle τ_i peut commencer sa toute première exécution,
- C_i (computing time) : durée d'exécution de τ_i. Ce paramètre est considéré dans plusieurs travaux sur l'ordonnancement temps réel comme le pire cas des temps d'exécution (WCET pour Worst Case Execution Time) sur le processeur sur lequel elle va s'exécuter;
- D_i : échéance relative ou délai critique relatif à chaque activation de τ_i,
- T_i : période d'exécution de τ_i.

D'autres paramètres statiques sont dérivés des paramètre de base :

- $u_i = \frac{C_i}{T_i}$: le facteur d'utilisation du processeur par τ_i, $u_i \leq 1$,
- $CH_i = \frac{C_i}{D_i}$: la densité de la tâche τ_i, $CH_i \leq 1$.

Les paramètres dynamiques d'une tâches déterminent le comportement pendant l'exécution de la tâche τ_i. Ces paramètres sont définis pour chaque instance k noté τ_i^k par :

- r_i^k (release time) : date à laquelle τ_i^k peut commencer son exécution appelée date d'activation ou date de réveil de la tâche,
- s_i^k, f_i^k : respectivement la date de début d'exécution effective et date de fin d'exécution de τ_i^k,
- R_i^k : temps de réponse de τ_i^k donné par $R_i^k = f_i^k - r_i^k$. Le pire temps de réponse de τ_i noté R_i est défini par $R_i = max\{R_i^k\}_{k \geq 1}$;
- $d_i^k = r_i^k + D_i$: échéance absolue de τ_i^k relative à r_i^k, c'est la date dont le dépassement entraîne une faute temporelle,
- $L_i^k(t)$ (laxity) : c'est la laxité de l'instance τ_i^k à la date t, qui représente le retard maximum pour reprendre l'exécution de la tâche τ_i quand elle s'exécute seule.

1.3.1 Tâches concrètes/non concrètes

Si toutes les dates de première activation de toutes les tâches sont connues on dit que les tâches sont concrètes. Au contraire si on ne connaît pas ces dates de première activation on dit que ces tâches sont non concrètes.

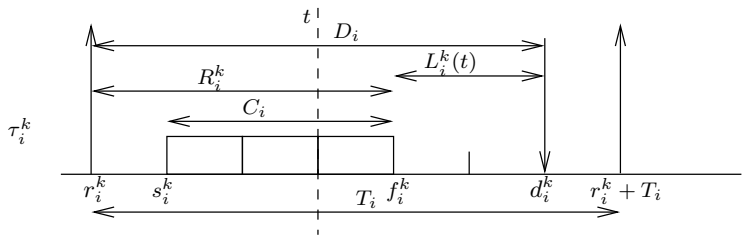

FIGURE 1.2 – Paramètres dynamiques de l'instance k de τ_i

1.3.2 Tâches synchrones/asynchrones

Si toutes les tâches sont concrètes et ont les mêmes dates de première activation, on dit que les tâches sont à activations synchrones. Sinon, elles sont asynchrones.

1.4 Contraintes temps réel

Dans l'ordonnancement temps réel, les tâches peuvent être soumises à plusieurs contraintes telles que des contraintes d'échéances, de périodicité stricte, de dépendances et de précédences, etc.

1.4.1 Échéances

Les contraintes d'échéances permettent d'exprimer une condition sur la date de fin d'exécution au plus tard d'une tâche. Considérons un système de tâches périodiques, suivant la relation entre la période T_i et l'échéance relative D_i de chaque tâche τ_i nous distinguons trois types d'échéances :

- $D_i = T_i$: à chaque activation de τ_i, la tâche τ_i doit être exécutée avant sa prochaine activation. On parle de tâches à échéances sur activations, échéances implicites, ou échéances sur requêtes;
- $D_i \leq T_i$: à chaque activation de τ_i, la tâche τ_i doit être exécutée au plus tard à une date inférieure ou égale à la date de sa prochaine activation. On parle de tâches à échéances contraintes;
- $D_i \neq T_i$: il n'y a pas de corrélation entre la date de fin d'exécution au plus tard de τ_i lors d'une activation et sa prochaine activation de τ_i. On peut avoir $D_i \leq T_i$ ou $D_i > T_i$, on parle de tâches à échéances arbitraires.

1.4.2 Périodicité stricte

Considérons une tâche périodique τ_i dans un système de tâches temps réel, une contrainte de périodicité stricte impose que le temps qui s'écoule entre deux dates de début d'exécution consécutives s_i^k et s_i^{k+1} corresponde exactement à la période de la tâche τ_i. L'intérêt de cette contrainte est que la connaissance de la date de début effective de la première instance s_i^1 de τ_i implique la connaissance de la date de début effective de toutes les instances suivantes de la même tâche [Cuc04], cela s'exprime par la relation $s_i^{k+1} = s_i^1 + kT_i, \forall k \geq 1$. C'est une contrainte qui est utile pour les tâches correspondant aux capteurs et actionneurs et les traitements réalisant une boucle de commande d'un processus.

1.4.3 Dépendances entre tâches

Une dépendance entre deux tâches τ_i et τ_j peut être de deux types : une dépendance de précédence et/ou une dépendance de données. Une dépendance de précédence entre (τ_i, τ_j) impose que la tâche τ_j commence son exécution après que la tâche τ_i ait complètement fini de s'exécuter [Cuc04, HCAL89, CL87, Xu93]. Les contraintes de précédences sont indirectement des contraintes temps réel et on dit que la tâche τ_i est un prédécesseur de la tâche τ_j et τ_j est un successeur de τ_i. Si τ_i s'exécute exactement une fois avant une exécution de τ_j on a une contrainte de précédence simple sinon une contrainte de précédence étendue [RCR01, FGPR11, PFB$^+$11].

Une dépendance de données signifie que la tâche τ_i produit une donnée qui est consommée par τ_j [Cuc04, HCAL89], cette dépendance entraîne forcément une précédence entre les tâches. Les tâches sont dites indépendantes lorsqu'elles ne sont définies que par leurs paramètres temporels.

L'ensemble des dépendances entre les tâches peut être modélisé par un graphe orienté où les sommets représentent les tâches et l'ensemble des arcs la relation de dépendances entre les tâches. Cette relation est antisymétrique et transitive, c'est donc une relation d'ordre sur l'exécution des tâches. C'est une relation d'ordre partiel car certaines tâches ne sont pas dépendantes. Un exemple de graphe de dépendances de tâches est présenté sur la figure 5.1.

On distingue deux types de dépendances de données :

- sans pertes de données : les données, produites par chaque instance de la tâche productrice sont toutes consommées par l'instance de la tâche consommatrice,
- avec pertes de données : la tâche consommatrice peut perdre des données qui sont écrasées par d'autres données produites par l'exécution d'autres instances de la tâche productrice.

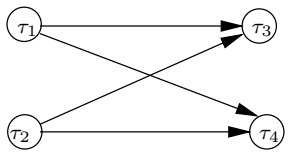

FIGURE 1.3 – Exemple de graphe de dépendances

1.4.4 Latence

La latence est définie pour des tâches dépendantes éventuellement par transitivité dans le cas d'un chemin de tâches. Soient τ_i et τ_j deux liées par un chemin dans un graphe de tâches dépendantes. La latence entre τ_i et τ_j notée $L(\tau_i, \tau_j)$ est la durée entre le début d'exécution de τ_i et la fin d'exécution de τ_j. La figure 1.4 donne une illustration de la latence entre deux tâches.

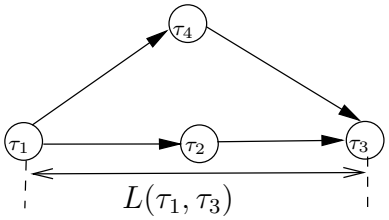

FIGURE 1.4 – Illustration de la latence d'un graphe de tâches

Une contrainte de latence entre τ_i et τ_j signifie que $L(\tau_i, \tau_j)$ doit être inférieure ou égale à une certaine valeur.

1.5 Ordonnancement des systèmes temps réel

1.5.1 Algorithmes d'ordonnancement

Un système temps réel est constitué d'un ou plusieurs processeurs et d'un ensemble de tâches ayant des contraintes temps réel. Un algorithme d'ordonnancement détermine l'ordre total et les dates de démarrage des exécutions des tâches sur un ou plusieurs processeurs. Il existe plusieurs classes d'algorithmes d'ordonnancement temps réel.

Monoprocesseur/Multiprocesseur

L'ordonnancement est de type monoprocesseur si toutes les tâches s'exécutent sur un seul processeur. Si les tâches peuvent s'exécuter sur plusieurs processeurs l'ordonnancement est multiprocesseur.

Préemptif/Non préemptif

Un algorithme d'ordonnancement est préemptif lorsqu'une tâche s'exécutant sur un processeur peut être suspendue au profit d'une tâche, dite plus prioritaire, puis reprendre son exécution plus tard. Dans le cas contraire il est non préemptif.

En ligne/Hors ligne

Un algorithme d'ordonnancement hors ligne ("off-line" en anglais) construit la séquence complète des dates de début d'exécution des tâches sur la base des paramètres temporels. Cette séquence est connue avant l'exécution de l'ensemble des tâches. En pratique, l'ordonnancement a la forme d'un plan hors ligne (ou statique), exécuté de façon répétitive ou cyclique.

Un ordonnancement en ligne ("on-line" en anglais) construit la séquence des dates de début d'exécution des tâches à partir des paramètres temporels des tâches pendant l'exécution de l'ensemble des tâches. Les algorithmes en ligne sont plus robustes vis-à-vis des dépassements des WCETs.

Priorité fixe/dynamique

Un algorithme d'ordonnancement est à priorités fixes, si les priorités des tâches sont basées sur des paramètres statiques (par exemple la période). Il existe des algorithmes d'ordonnancement à priorités fixes aux tâches et d'autres qui sont à priorité fixes aux instances ("job").

Un algorithme d'ordonnancement est à priorités dynamiques, si les priorités des tâches sont basées sur des paramètres dynamiques (par exemple la laxité).

Optimal/Non optimal

Par définition, un algorithme d'ordonnancement optimal pour une classe de problème d'ordonnancement donnée (hors ligne, en ligne, priorités fixes ou dynamiques, etc.) est tel que : si un système est ordonnançable par au moins un algorithme de la même classe, alors le système est aussi ordonnançable par l'algorithme d'ordonnancement optimal. En conséquence, si un système n'est pas

ordonnançable par l'algorithme d'ordonnancement optimal d'une classe donnée, alors il ne l'est pas par aucun autre algorithme d'ordonnancement de la même classe.

1.5.2 Analyse de faisabilité

Une analyse de faisabilité d'un ensemble de tâches permet de déterminer s'il existe un algorithme d'ordonnancement qui peut ordonnancer cet ensemble de tâches. La condition que doit vérifier l'ensemble des tâches pour être faisable est appelée condition de faisabilité.

1.5.3 Analyse d'ordonnançabilité

Une analyse d'ordonnançabilité d'un ensemble de tâches permet de vérifier si cet ensemble de tâches est ordonnançable relativement à un algorithme d'ordonnancement, c'est-à-dire vérifier que toute instance de chaque tâche respecterason échéance. La condition que doit vérifier l'ensemble des tâches pour être ordonnançable est appelée condition d'ordonnançabilité.

Définition 1.5.1 *Une condition d'ordonnançabilité suffisante est une condition qui lorsqu'elle est vérifiée par toutes les tâches assure que celles-ci sont ordonnançables. Dans le cas contraire, elles ne sont pas forcément non ordonnançables.*

Définition 1.5.2 *Une condition d'ordonnançabilité nécessaire est une condition qui lorsqu'elle n'est pas vérifiée par toutes les tâches assure que celles-ci ne sont pas ordonnançables. Dans le cas contraire, elles ne sont pas forcément ordonnançables.*

Définition 1.5.3 *Une condition d'ordonnançabilité nécessaire et suffisante est une condition qui lorsqu'elle n'est pas vérifiée par toutes les tâches assure que celles-ci ne sont pas ordonnançables et dans le cas contraire assure que celles-ci sont ordonnançables.*

Une analyse d'ordonnançabilité peut se faire selon deux approches.

1.5.3.1 Approche analytique

L'analyse d'ordonnançabilité par approche analytique consiste à identifier le ou les pires cas d'exécution et de déterminer analytiquement une condition d'ordonnançabilité des tâches. Dans cette approche on peut utiliser :

– le facteur d'utilisation des tâches : dans certains cas le facteur d'utilisation des tâches permet de conclure si un ensemble de tâches est ordonnançable ou non,
– le pire temps de réponse : le pire temps de réponse d'une tâches est calculé puis comparé à son échéance. Si une tâche a un pire temps de réponse inférieur ou égal à son échéance relative alors elle est ordonnançable;
– le temps de demande processeur : c'est le temps requis par le processeur pour exécuter un ensemble de tâches activées puis terminées dans un intervalle donné. Si ce temps est inférieur à la longueur de l'intervalle considéré alors les tâches activées dans cet intervalle sont faisables. Sinon les tâches ne sont pas faisables.

1.5.3.2 Approche par simulation

L'approche par simulation revient à faire l'analyse temporelle de l'exécution des tâches temps réel. Lorsque les tâches sont périodiques, cette exécution dure indéfiniment. Il suffit alors de faire l'analyse temporelle sur un intervalle fini appelé intervalle d'étude de l'ordonnancement [LM80, Goo99]. Ainsi si toutes les tâches respectent leurs contraintes sur cet intervalle alors le système de tâches est ordonnançable.

1.6 Viabilité d'une condition d'ordonnançabilité

Une condition d'ordonnançabilité est dite viable ("sustainable") si l'ordonnancement trouvé grace à cette condition d'ordonnançabilité reste valide lors de l'exécution quand intervient un changement intuitivement positif c'est-à-dire par exemple la dimunition de la durée d'exécution d'une tâche, l'augmentation de la période d'une tâche ou de l'échéance relative, etc.

Dans ce travail, on suppose que les périodes et les échéances ne seront pas modifiées. Donc nous considérons qu'une condition d'ordonnançabilité est viable ("sustainable") lorsque l'ordonnancement qu'elle trouve reste valide même si la durée d'exécution d'une tâche diminue lors de son exécution.

1.7 Ordonnanceur

Un ordonnanceur temps réel est un programme temps réel chargé d'allouer une ou des tâche(s) à ordonnancer au(x) processeur(s). Il est invoqué à certains instants. Et ceci suivant un algorithme ordonnancement donné de tel sorte que toutes les tâches respectent leurs contraintes temps réel.

1.7.1 Invocation de l'ordonnanceur

Il existe plusieurs manières d'invoquer un ordonnanceur :

- invocation guidée par les évènements (Event-Trigger ET) [CMTN08] : l'ordonnanceur est invoqué sur réception d'un évènement tel que l'activation d'une tâche, la libération d'une ressource (processeur, périphériques d'entrées et de sorties, etc.), la fin d'exécution d'une tâche. Ce mode d'invocation est utilisé dans l'ordonnancement en ligne qui prend les décisions d'ordonnancement en fonction des évènements qui se produisent lors de l'exécution des tâches;
- invocation guidée par le temps (Time-Trigger TT) [Kop98] : les invocations sont indépendantes des évènements. Les dates d'invocation de l'ordonnanceur sont fixées avant l'exécution des tâches et sont généralement périodiques. Cela définit des tranches de temps (slot) pendant lesquelles on peut effectuer un traitement, alors que dans les approches dirigées par les évènements (Event-Trigger) le traitement est directement activé par un événement. Ce mode d'invocation de l'ordonnanceur est utilisé dans l'ordonnancement hors ligne où toutes les décisions d'ordonnancement ont été prises hors ligne.

1.7.2 États et gestion des tâches

Le partage du ou des processeur(s) et des ressources introduit plusieurs états pour une tâche :

- *Inactif* : la tâche n'est pas encore activée,
- *Prêt* : la tâche est activée et elle dispose de toutes les ressources dont elle a besoin pour s'exécuter,
- *Bloqué* : la tâche est en attente de ressources,
- *Exécution* : la tâche s'exécute,
- *Passif* : la tâche n'a pas de requête en cours, elle a fini son exécution.

L'ordonnanceur est chargé d'assurer la transition d'une tâche d'un état à un autre. À chaque invocation, l'ordonnanceur met à jour la liste des tâches prêtes en y ajoutant toutes les tâches activées et qui disposent de leurs ressources et en supprimant les tâches qui ont fini leurs exécutions ou qui sont bloquées par l'attente d'une ressource. Ensuite parmi les tâches prêtes, l'ordonnanceur sélectionne la tâche la plus prioritaire pour s'exécuter. Ainsi, une tâche dans l'état *inactif* peut passer à l'état *prêt*. Une tâche à l'état *prêt* peut passer à l'état *exécution* ou à l'état *bloqué*. Une tâche à l'état *exécution* peut revenir à l'état *prêt* si elle est préemptée par une autre tâche plus prioritaire, elle peut passer à l'état *bloqué* si elle attend la

libération d'une ressource ou si elle a fini son exécution, elle passe à l'état *passif*. Une tâche peut passer de l'état *bloqué* à l'état *prêt*. Et enfin une tâche peut passer de l'état *passif* à l'état *prêt*. La figure 1.5 donne une illustration des différents états et leurs transitions.

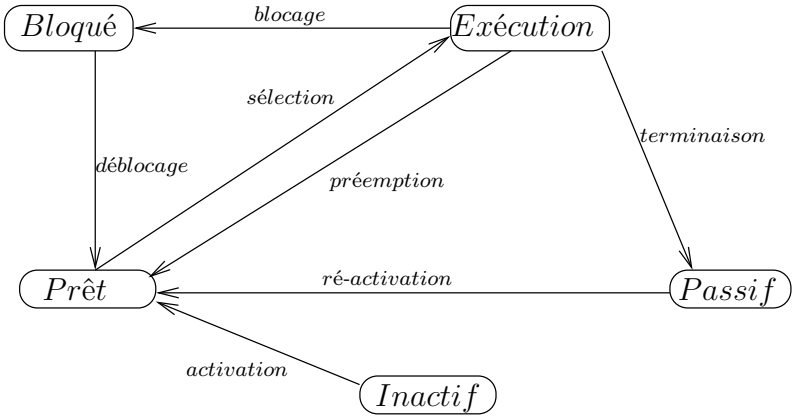

FIGURE 1.5 – États et transitions d'une tâche

1.8 Conclusion

Dans ce chapitre nous avons présenté les concepts de base de l'ordonnancement temps réel que nous aurons à utiliser dans la suite de cette thèse. Dans le chapitre qui suit nous allons présenter un état de l'art l'ordonnancement temps réel.

Chapitre 2

État de l'art

Après avoir défini les concepts de base de l'ordonnancement temps réel, nous présentons dans ce chapitre un état de l'art de l'ordonnancement temps réel monoprocesseur et multiprocesseur.

2.1 Ordonnancement monoprocesseur

2.1.1 Priorités fixes

2.1.1.1 Priorités fixes aux tâches

Une priorité fixe aux tâches est une priorité qui ne varie pas au cours de l'exécution de la tâche. Les algorithmes d'ordonnancement à priorités fixes aux tâche les plus utilisés sont "Rate Monotonic" et "Deadline Monotonic".

Rate Monotonic (RM)

L'algorithme d'ordonnancement RM a été introduit par Liu et Layland en 1973 [LL73]. C'est un algorithme d'ordonnancement préemptif qui s'applique à des tâches périodiques indépendantes, et à échéance sur requête ($T_i = D_i$). La priorité d'une tâche est inversement proportionelle à sa période, c'est-à-dire que plus la période d'une tâche est petite, plus sa priorité est grande. Cet algorithme est optimal dans la classe des algorithmes à priorités fixes pour les tâches indépendantes préemptibles à échéance sur requête non conrètes ou synchrones. Une condition suffisante d'ordonnançabilité de l'algorithme d'ordonnancement RM [LL73] pour un ensemble de tâches périodiques Γ_n à échéance sur requête est donnée par :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

Deadline Monotonic (DM)
L'algorithme d'ordonnancement DM a été introduit par Leung et Whitehead en 1982 [LW82] pour des tâches à échéance sur contrainte. La priorité d'une tâche est inversement proportionelle à son échéance relative, c'est-à-dire que plus l'échéance relative d'une tâche est petite, plus sa priorité est grande. Cet algorithme est optimal dans la classe des algorithmes préemptif à priorités fixes pour les tâches indépendantes préemptibles à échéance contrainte ($D_i \leq T_i$). La condition suffisante d'ordonnançabilité de l'algorithme d'ordonnancement DM [LW82] pour un ensemble de tâches périodiques Γ_n à échéance sur contrainte est donnée par :

$$\forall \tau_i \in \Gamma_n, C_i + \sum_{j \in hp(\tau_i)} \lceil \frac{D_i}{T_j} \rceil . C_i \leq D_i$$

avec $hp(\tau_i)$ est l'ensemble de tâches de Γ_n de priorités supérieures ou égales à celle de τ_i, n'incluant pas τ_i.

Analyse du temps de réponse

Le temps de réponse des tâches peut être utilisé pour faire l'analyse d'ordonnançabilité. Ainsi Joseph and Pandya [JP86] ont proposé une condition d'ordonnançabilité nécessaire et suffisante basée sur le calcul du pire temps réponse. Si Γ_n, un ensemble de tâches, est ordonnancé par un algorithme à priorités fixes aux tâches, le pire temps de réponse de $\tau_i \in \Gamma_n$ est donné par le plus petit point fixe (supérieur à C_i) de :

$$R_i = C_i + \sum_{j \in hp(\tau_i)} \lceil \frac{R_i}{T_i} \rceil . C_i \tag{2.1}$$

avec $hp(\tau_i)$ est l'ensemble de tâches de Γ_n de priorités supérieures ou égales à celle de τ_i. Déterminer la valeur de R_i pour τ_i est équivalent à résoudre le point fixe 2.1. Ainsi dans le cas des échéances contraintes, Γ_n est ordonnançable par l'ordonnancement à priorités fixes aux tâches si et seulement si :

$$\forall \tau_i \in \Gamma_n, R_i \leq D_i$$

2.1.1.2 Priorités fixes aux instances

Une priorité fixe au niveau des instances est une priorité qui ne varie pas au cours de l'exécution des instances. L'algorithme d'ordonnancement à priorités fixes aux instances le plus utilisé est "Earliest Deadline First".

Earliest Deadline First (EDF)

L'algorithme d'ordonnancement EDF a été introduit par Lui et Layland en 1973 [LL73]. C'est un algorithme d'ordonnancement qui peut être préemptif ou non préemptif et qui s'applique à des tâches périodiques indépendantes à échéance sur requête ($T_i = D_i$). La plus grande priorité à la date t est allouée à la tâche dont l'échéance absolue est la plus proche. EDF est optimal pour les tâches indépendantes préemptibles. Une condition d'ordonnançabilité nécessaire et suffisante de EDF pour un ensemble de tâches périodiques à échéance sur requête noté Γ_n est donnée par :

$$\forall \tau_i \in \Gamma_n, \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

2.1.2 Priorités dynamiques

Une priorité dynamique varie durant l'exécution d'une instance. L'algorithme d'ordonnancement à priorités dynamiques le plus utilisé est "Least laxity Fisrt".

Least-Laxity First(LLF)

L'algorithme d'ordonnancement LLF se base sur la laxité. La tâche dont la laxité est la plus faible comparée à toutes les tâches prêtes aura la plus grande priorité [Mok83]. Cet algorithme est optimal pour les tâches indépendantes préemptibles. D'après [CDKM00], la condition d'ordonnançabilité de LLF et celle de EDF sont les mêmes. c'est-à-dire que la condition d'ordonnançabilité nécessaire et suffisante de LLF pour un ensemble de tâches périodiques à échéance sur requête Γ_n est donnée par :

$$\forall \tau_i \in \Gamma_n, \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Lorsque plusieurs tâches possèdent la même laxité, l'algorithme LLF a l'inconvénient d'engendrer un grand nombre de préemption ce qui explique qu'il soit aussi peu utilisé dans le cas monoprocesseur [Bim07].

2.2 Ordonnancement multiprocesseur

2.2.1 Architecture multiprocesseur

Une architecture multiprocesseur est composée de plusieurs processeurs. Plusieurs tâches peuvent s'exécuter en parallèle sur les différents processeurs. Une

architecture multiprocesseurs peut être composée de :

- processeurs identiques : processeurs qui ont la même puissance de calcul,
- processeurs uniforme : chaque processeur est caractérisé par sa puissance de calcul (nombre d'instructions traitées par seconde), avec l'interprétation suivante : lorsqu'un travail (instance d'une tâche) s'exécute sur un processeur de puissance de calcul s pendant t unités de temps, il réalise $s \times t$ instructions sur ce processeur. La vitesse de progression d'un travail varie d'un processeur à un autre;
- processeurs indépendants : un taux d'exécution $r_{i,j}$ est associé à chaque couple de travail-processeur (τ_i^k, p_j), avec l'interprétation suivante : si le travail τ_i^k s'exécute sur p_j pendant t unités de temps alors τ_i^k réalise $r_{i,j} \times t$ instructions. La vitesse de progression sur un processeur varie d'un travail à un autre.

Selon le type de mémoire nous pouvons distinguer :

- une architecture multiprocesseur parallèle : les processeurs ont une mémoire partagée par le bais de laquelle ils communiquent,
- une architecture multiprocesseur distribuée : chaque processeur à sa propre mémoire et les communications inter-processeurs se font par envoi/réception de messages.

2.2.2 Stratégies d'ordonnancement

Considérons un ensemble de n tâches temps réel indépendantes préemptibles à échéances sur requètes noté Γ_n et une architecture multiprocesseur composée de m processeurs. Un algorithme d'ordonnancement multiprocesseur détermine pour chaque tâche, le processeur sur lequel cette tâche doit s'exécuter (problème d'allocation) et sur chaque processeur, la date et l'ordre de démarrage d'exécution des tâches (problème d'ordonnancement).

Dans l'ordonnancement temps réel multiprocesseur nous distinguons deux cas :

- nombre de processeurs non fixé [DL78, OS93, OS95], donc nous pouvons utiliser autant de processeurs que nécessaire pour l'ordonnancement,
- nombre de processeurs fixé [OB98, BLOS95], donc impossible de dépasser ce nombre.

Dans le premier cas nous n'avons pas besoins de faire une analyse d'ordonnançabilité car le système de tâches à ordonnancer sera toujours ordonnançable (il suffit de prendre un processeur par tâche). Par contre dans le cas où le nombre de processeurs est fixé il peut arriver que le système ne soit pas ordonnançable, ce

qui nécessite une étude d'ordonnançabilité.

L'ordonnancement temps réel multiprocesseur peut être effectué en utilisant la stratégie d'ordonnancement par partitionnement ou la stratégie d'ordonnancement globale. Il existe aussi une stratégie d'ordonnancement hybride dite de semi-partitionnement obtenue par combinaison de la stratégie globale et par partitionnement.

Dans la suite de ce documents nous limiterons à utiliser le terme stratégie par partitionnement pour désigner la stratégie d'ordonnancement par partitionnement (respectivement stratégie globale pour désigner stratégie d'ordonnancement global).

2.2.2.1 Stratégie par partitionnement

La stratégie par partitionnement [Goo07, DB09, ZA05, AJ] consiste à partitionner l'ensemble des n tâches en m sous-ensembles disjoints (Γ^1, Γ^2, $\cdots$, Γ^m tels que $\cup_{i=1}^{m}\Gamma^i$=Γ_n et m étant le nombre de processeurs) puis d'ordonnancer chaque sous-ensemble de tâches Γ^i sur un processeur p_j avec une stratégie d'ordonnancement "locale" monoprocesseur. Il y a un ordonnanceur par processeur. Les tâches allouées aux processeurs ne sont pas autorisées à migrer d'un processeur à l'autre, la préemption ne peut entraîner de migration. La stratégie par partitionnement a l'avantage de ramener le problème d'ordonnancement multiprocesseur en plusieurs problèmes d'ordonnancement monoprocesseur pour lesquels il existe de très nombreuses solutions dans la littérature. Cependant le partitionnement des tâches est NP-difficile [GJ79] car équivalent au problème de "Bin Packing" [CGMV98] qui consiste à ranger un ensemble d'objets de tailles différentes dans un nombre minimum de boîtes de tailles identiques.

Les travaux sur l'ordonnancement temps réel multiprocesseur avec un nombre de processeurs non fixé qui utilisent la stratégie de partitionnement ont principalement comme objectif de minimiser le nombre de processeurs nécessaires à l'ordonnancement. Parmi ces travaux nous avons :

– dans la famille des algorithmes d'ordonnancement à priorités fixes aux tâches, Dhall et Liu [DL78], sont les premiers à proposer deux algorithmes préemptifs avec des tâches indépendantes et périodiques utilisant les heuristiques de "Bin Packing" (voir 2.3.2.2.1) sur processeurs identiques: Rate-Monotonic-Next-Fit (RM-NF) et Rate-Monotonic-First-Fit (RM-FF). Ces deux algorithmes d'ordonnancement multiprocesseur utilisent Rate Monotonic pour l'ordonnancement des tâches sur chaque processeur et les heuristiques respectivement "next-fit" et "first-fit" comme heuristiques d'allo-

cation des tâches aux processeurs. Dans RM-NF et RM-FF les tâches sont triées dans l'ordre croissant de leurs périodes avant le début de l'allocation. Oh et Son ont repris dans [OS93] ces deux algorithmes d'ordonnancement pour corriger une erreur sur l'évaluation des performances et présentent RM-BF utilisant la méthode d'allocation "best-fit";

- dans [DD86, OS95] est proposé l'algorithme RM First-Fit Decreasing Utilisation Factor (RM-FFDUF), un algorithme similaire à "first-fit" où les tâches sont triées dans l'ordre décroissant de leurs facteurs d'utilisation avant l'allocation;
- deux autres algorithmes à priorités fixes aux tâches sont proposés dans [BLOS95]. Ce sont le RM-Small-Tasks (RM-ST) et RM-General-Tasks (RM-GT). Le premier algorithme RM-ST alloue les tâches aux processeurs avec l'heuristique "next-fit", mais partitionne les tâches de telle sorte que β (défini dans le théorème 5 [BLOS95]) ait la plus petite valeur possible. Le deuxième algorithme RM-GT partitionne l'ensemble des tâches en deux groupes selon le critère défini dans [BLOS95], ensuite sur les tâches du premier groupe RM-ST est appliqué, et sur le second groupe de tâches "first-fit" est appliqué, avec la condition d'ordonnançabilité définie dans [LSD89];
- dans le cas des algorithmes d'ordonnancement préemptifs à priorités fixes aux instances, Lopez et al. dans [LDG04] présentent EDF-NF,EDF-FF et EDF-WF, des algorithmes d'ordonnancement multiprocesseur utilisant EDF pour l'ordonnancement et les heuristiques de "bin packing" pour l'allocation des tâches.

Au lieu de considérer un nombre infini de processeurs disponibles, d'autres travaux fixent ce nombre et cherchent des conditions d'ordonnançabité pour le système de tâches. Parmi ces travaux :

- Oh et Baker [OB98] montrent qu'en appliquant "first-fit" sous RM on peut ordonnancer n'importe quel ensemble de tâches Γ sur $m \geq 2$ processeurs identiques. Dans ce même article ils ont démontré qu'avec un algorithme d'ordonnancement multiprocesseur préemptif à priorités fixes aux tâches, le pire facteur d'utilisation des m processeurs noté U_{min} est borné par :

$$m(2^{1/2} - 1) < U_{min} \leq (m + 1)/(1 + 2^{(m+1)})$$

  m étant le nombre de processeurs. Le pire facteur d'utilisation des processeurs U_{min} est défini tel que si $U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{min}$ alors le système constitué des n tâches est ordonnançable sur les m processeurs;
- dans [LDG03] Lopez et al. ont amélioré la borne du facteur d'utilisation définie dans [OB98] et l'ont montrée égale à :

$$U_{RM-FF}(n, m) = (m-1)(2^{(1/2-1)}) + (n-m+1)(2^{1/(n-m+1)} - 1) \quad (2.2)$$

Ce résultat est particulièrement intéressant dans le cas ou le nombre de processeurs est petit. Si les facteurs d'utilisation des tâches sont plus petits qu'un nombre α alors U_{RM-FF} est donné par :

$$U_{RM-FF}(n,m,\alpha) = (m-1)(2^{1/(\beta+1)}-1)\beta+(n-\beta(m-1))(2^{1/(n-\beta(m-1))}-1) \quad (2.3)$$

Où $\beta = \lfloor 1/log_2(\alpha+1) \rfloor$. En remplaçant α par 1 dans 2.3 on retrouve 2.2;

- Buchard et Liebeher dans [BLOS95] ont calculé les bornes maximales des facteurs d'utilisation des algorithmes RM-ST et RM-GT. Avec un système de $m \geq 2$ processeurs, $U_{RM-ST} = (m-2)(1-\alpha)+1-ln2$, α étant le facteur d'utilisation maximal de l'ensemble des n tâches. Donc l'algorithme RM-ST trouve un partitionnement possible sur un ensemble de m processeurs si la somme des facteurs d'utilisation des tâches périodiques notée U_n reste inférieure à U_{RM-ST}. Avec RM-GT le résultat est différent, U_{RM-GT} est donné par $U_{RM-GT} = 0.5(m-1.42)$ et ne dépend pas de α et si $U_n \leq U_{RM-GT}$ alors le systèmes de tâches est ordonnançable. Dans [LGD04] est récapitulé le calcul des bornes maximales et minimales des facteurs d'utilisation permettant de garantir les conditions d'ordonnançabilité d'un ensemble de n tâches sur m processeurs avec RM;
- concernant les algorithmes à priorités fixes aux instances, il existe des algorithmes préemptifs basés sur EDF qui fournissent des conditions d'ordonnançabilité utilisant le facteur d'utilisation des processeurs : EDF-FF et EDF-BF dans [LGDG00] avec $U_{EDF-FF} = U_{EDF-BF} = \frac{\beta_{EDF}m+1}{\beta_{EDF}+1}$ (avec $\beta_{EDF} = \lfloor \frac{1}{\alpha} \rfloor$ et α représentant le facteur d'utilisation maximal des tâches);
- dans [ESHA94, ESHA90], les algorithmes génétiques sont utilisés pour résoudre le problème d'ordonnancement avec des contraintes de précédences.

2.2.2.2 Stratégie globale

Une stratégie gloable applique globalement un algorithme d'ordonnancement sur l'ensemble de l'architecture multiprocesseur [Goo07, DB09, ZA05]. Toutes les tâches sont dans la même queue des tâches prêtes qui est partagée par l'ensemble des processeurs. Dans cette queue les m tâches les plus prioritaires sont sélectionnées pour être exécutées sur les m processeurs, par exemple en utilisant globalement RM, EDF, etc. Pour ces stratégies globales outre la préemption des tâches, les migrations de ces dernières sont autorisées. Une tâche peut commencer son exécution sur un processeur (soit p_j), être préemptée par l'arrivée d'une nouvelle tâche plus prioritaire et reprendre son exécution sur un autre processeur (disons $p_{j'}, j' \neq j$). Dans la stratégie globale, nous avons un seul ordonnanceur et préempter une tâches revient éventuellement à la faire migrer vers un autre processeur. C'est ce phénomène de migration de tâches qui caractérise la stratégie

globale.

L'avantage de la stratégie d'ordonnancement globale est de permettre une meilleure utilisation des processeurs. Son inconvénient est que le coût de migration des tâches est non négligeable. Parmi les travaux basés sur la stratégie globale notons :

- pour les tâches indépendantes, préemptives et périodiques, il existe plusieurs algorithmes d'ordonnancement à priorités fixes aux instances [SB02, Bar04, BG03, GFB03] ou fixes aux tâches [ABJ01, RM00] qui reposent sur la stratégie globale. Dans [SB02] est présenté l'algorithme EDF-US sur m processeurs identiques (avec des tâches indépendantes et périodiques) avec comme condition d'ordonnançabilité $U_n \leq m^2/(2m-1)$, U_n est la somme des facteurs d'utilisation des tâches à ordonnancer. Cet algorithme est une modification de EDF en choisissant les priorités des tâches selon la règle suivante : si $u_i > m/(2m-1)$ alors aux "jobs" de la tâche τ_i est allouée la plus haute priorité, sinon si $u_i \leq m/(2m-1)$ on applique EDF. Goossens et al. [GFB03] ont donné la condition suffisante d'ordonnaçabilité avec EDF pour des tâches périodiques à échéances sur requête, suivante: si $U_n \leq m(1-\alpha)+\alpha$, $\alpha = max(u_i), 1 \leq i \leq n$ alors le système de tâches Γ est ordonnançable sur les m processeurs.
- un algorithme à priorités fixes aux tâches RM-US similaire à EDF-US est présenté dans [ABJ01] avec $U_{min} = m^2/(3m-2)$. Des algorithmes d'ordonnancement préemptifs à priorités fixes aux instances avec des tâches sporadiques sont présentés dans [BG08, AB08, Bar07, BB08]. Dans [Bak03] nous avons un algorithme à priorités fixes aux tâches avec Deadline Monotonic qui donne le résultat suivant : avec des tâches périodiques et $m \geq 2$ processeurs, la condition suffisante d'ordonnançabilité est donnée par: $U_n \leq \frac{m}{2}(1-\alpha)\alpha$.
- dans [AJ03] nous avons des algorithmes d'ordonnancement basés sur la notion d'équité proportionnée (proportionate fairness). Ces algorithmes dits "pfair" diffèrent des algorithmes d'ordonnancement classiques dans la mesure où il est requis avec ces algorithmes que les tâches s'exécutent à un taux régulier (quasi constant) en divisant une tâche en série de sous-tâches. Ces sous-tâches doivent s'exécuter dans des intervalles de tailles identiques appelés fenêtre.

Il faut noter que les stratégies globales et par partitionnement ne sont pas comparables d'après Leung et Whitehead [LW82]. Ils ont montré que pour les algorithmes d'ordonnancement à priorités fixes : i) il y a des systèmes de tâches périodiques qui sont ordonnançables en utilisant m processeurs identiques à l'aide d'une approche partitionnée mais pour lesquels aucune stratégie globale existe

pour les ordonnancer, ii) il y a des systèmes de tâches périodiques ordonnançables avec une approche globale en utilisant m processeurs mais pour lesquels aucun partitionnement de m sous-ensembles ordonnançables existe.

2.2.2.3 Stratégie par semi-partitionnement

Elle est obtenue par combinaison de la stratégie par partitionnement et de la stratégie globale. Chaque tâche est allouée à un processeur pour toute son exécution, cependant si une tâche ne peut s'exécuter entièrement sur aucun processeur, son exécution est partagée sur deux ou plusieurs processeurs dont la charge maximale n'est pas atteinte (les migrations sont autorisées pour ce type de tâches). Différents types d'algorithmes de semi-partitionnement sont présentés dans [KY08, AT06, KY07]. Cette approche permet de minimiser les migrations de tâches.

2.3 Allocation des tâches temps réel dans l'ordonnancement multiprocesseur

Les termes allocation, distribution, partitionnement ou assignation des tâches sont équivalents dans le vocabulaire de l'ordonnancement temps réel multiprocesseur. Dans la suite on utilisera le terme allocation. Allouer une tâche à un processeur signifie l'exécuter sur un processeur de façon à ce que cette tâche et toutes les autres tâches, qui sont déjà allouées à ce même processeur, soient ordonnançables selon une condition d'ordonnançabilité qui est précisée. L'allocation d'une tâche à un processeur se faisant en fonction d'un critère, le problème à résoudre est un problème d'optimisation. Le critère d'optimisation peut être le nombre de processeurs, le facteur d'utilisation, la latence, etc. Dans cette partie nous étudions les différentes méthodes existantes dans la résolution du problème d'allocation des tâches temps réel. Nous distinguons deux grandes catégories de méthodes de résolution : celles qui sont dites exactes et les méthodes approchées.

2.3.1 Méthodes exactes

Une méthode exacte est une méthode qui cherche, en parcourant toutes les allocations possibles, une solution (l'ordonnancement où toutes les contraintes sont respectées), si jamais celle-ci existe. Par ailleurs si cette méthode ne trouve pas de solutions, on en déduit que le problème n'a pas de solution. Et puisque le problème d'allocation inclut une optimisation, la solution trouvée par une méthode exacte est la meilleure solution possible dite optimale.

2.3.1.1 Branch & Bound

L'algorithme de "Branch and Bound" (B&B) (séparation et évaluation) [CY90, Tal09] est basé sur l'énumération de toutes les solutions du problème d'optimisation. L'espace de recherche est exploré dynamiquement en construisant un arbre dont la racine représente le problème à résoudre avec son espace de recherche associé. Les feuilles de l'arbre sont les solutions potentielles au problème et les noeuds internes les solutions partielles (en ordonnancement c'est une solution ne contenant pas toutes les tâches). La solution optimale est la meilleure parmi les solutions potentielles. La construction et l'exploration de cet arbre se font à l'aide de deux opérations : la séparation et l'évaluation des solutions partielles. L'opération de séparation détermine l'ordre de parcours de l'espace des solutions partielles (parcours en profondeur, en largeur, etc.). L'évaluation des solutions partielles se fait à l'aide d'une fonction coût qui permet d'exclure ou de garder une solution partielle. La performance d'un algorithme de B&B dépend non seulement de la méthode de séparation des solutions partielles mais aussi de la fonction de coût utilisée pour évaluer ces solutions.

L'algorithme de B&B est utilisé dans de nombreux travaux d'ordonnancement temps réel multiprocesseur comme dans [CY90, Xu93, PSA97].

2.3.1.2 Branch & Cut

L'algorithme de "Branch and Cut" [Mit02] utilise aussi un arbre pour représenter l'espace des solutions mais toutes les solutions ne sont pas énumérées. En effet pour chaque tâche candidate à l'ordonnancement, l'algorithme teste son ordonnançabilité sur chaque processeur puis à l'aide d'une fonction de coût il sélectionne le meilleur processeur pour la tâche sélectionnée. Si la tâche τ_i n'est ordonnançable sur aucun processeur alors l'algorithme de "Branch and Cut" fait un retour-arrière ("backtrack"), il revient à la tâche τ_{i-1} qui a été ordonnancée juste avant et il lui cherche un autre processeur. Si aucun processeur n'est trouvé pour ordonnancer la tâche τ_{i-1} alors l'algorithme B&C fait un deuxième retour-arrière. Tant qu'une tâche reste non allouée, ces retour-arrières sont itérés jusqu'à la première tâche qui a été allouée pour lui chercher un autre processeur. Ce qui permet de trouver une allocation possible si toutefois elle existe.

2.3.1.3 Programmation par contraintes

La programmation par contrainte (PPC) [Ben06, Hla04] est utilisée pour résoudre plusieurs types de problèmes tels que la planification, l'ordonnancement, etc. Dans la PPC, le problème est modélisé à l'aide de variables de décisions et de contraintes, où une contrainte est une relation entre une ou plusieurs variables

qui limite les valeurs que peuvent prendre simultanément chacune des variables qui est liée à cette contrainte. Un domaine est associé à chaque variable et celui-ci est constitué de l'ensemble des valeurs pouvant être affectées à cette variable. Lorsque le domaine d'une variable est réduit à une valeur, on dit que la variable est instanciée. Un ensemble de variables instanciées est une instanciation. Une solution au problème est alors définie comme une instanciation des variables qui satisfait la conjonction de toutes les contraintes.

Les algorithmes de recherche de solutions en PPC s'appuient généralement sur la propagation de contraintes pour réduire l'espace de recherche de solutions. La propagation de contraintes est le mécanisme qui consiste à déduire de nouvelles contraintes à chaque fois qu'un domaine de variable est modifié. Les algorithmes de recherche de solutions garantissent de trouver une solution, si toutefois elle existe, sinon ils peuvent prouver qu'il n'existe pas une solution qui satisfait toutes les contraintes.

Cette technique de programmation par contraintes a été utilisée dans plusieurs travaux [Hla04, PEHDN08, EJ00, CA95] pour résoudre le problème d'ordonnancement temps réel multiprocesseur.

2.3.1.4 Programmation linéaire

La programmation linéaire (PL) [Van96, DT03] permet de résoudre les problèmes d'optimisation dont la fonction objective et les contraintes sont linéaires. C'est un domaine central dans l'optimisation car les problèmes de programmation linéaire sont les problèmes d'optimisation les plus faciles (toutes les contraintes y sont linéaires). Dans un problème de PL, il est toujours question de minimiser ou de maximiser une fonction objectif exprimée à l'aide de variables de décisions. Les solutions à trouver doivent être représentées par des variables réelles. S'il est nécessaire d'utiliser des variables discrètes dans la modélisation, on parle de programmation linéaire en nombre entiers (PLNE). Un problème de PLNE est un problème de PL dans lequel on ajoute la contrainte supplémentaire que certaines variables ne peuvent prendre que des valeurs entières. La programmation linéaire mixte (PLM), permet de traiter des problèmes dans lesquels les variables sont mixtes (entières et réelles) [She11].

Dans [Sub05] est présenté une formulation sous la forme de PLNE pour le problème d'ordonnancement des tâches temps réel dépendantes dans un système multiprocesseur homogène.

2.3.1.5 Programmation dynamique

La programmation dynamique [Sar84] est basée sur la division récursive d'un problème en plusieurs sous-problèmes plus simples. Elle applique le principe de l'optimalité formulée par R. Bellman qui est le suivant : "Dans une séquence optimale de décisions, quelle que soit la première décision prise, les décisions subséquentes forment une sous-séquence optimale, compte tenu des résultats de la première décision". Ce principe implique que le problème à étudier puisse être formulé comme l'évolution d'un système. Il est aussi nécessaire de pouvoir décomposer le problème en plusieurs étapes qui permet une grande économie de calculs.

2.3.1.6 Les Réseaux de flots

Un réseau de flots [FJF62] est un graphe dont chaque arc a une capacité et reçoit une quantité de flots. Le flot reçu par un arc ne doit pas dépasser la capacité de l'arc. Dans un réseau de flots la quantité de flots entrant par un sommet doit être égale à la quantité de flot qui ressort par ce même sommet, à l'exception du sommet source qui n'a pas d'arcs entrants et de sommet puits qui n'a pas d'arcs sortants. Il est également possible d'avoir plusieurs sources et puits dans un réseau de flots et d'associer un coût à chaque arc correspondant au coût d'utilisation de cet arc.

Dans les réseaux de flots, le problème du minimum k-$coupe$ [FJF62] consiste à déterminer un ensemble d'arcs du graphe de telle sorte que la suppression de ses arcs sépare le graphe en exactement k composantes connexes (dans une composante connexe pour deux sommets il existe au moins un chemin qui les relie) et chaque composante a au moins un sommet puits ou un sommet source.

En représentant les tâches et les processeurs dans un même graphe comme un réseau de flots dont les puits et sources sont les processeurs, la solution au problème du minimum k-$coupe$ permet de résoudre le problème d'allocation des tâches en considérant que k est le nombre de processeurs disponibles pour l'allocation des tâches.

Généralement les travaux utilisant les réseaux de flots pour résoudre le problème d'allocation des tâches ne prennent pas en compte les contraintes temps réel [Sto77, CHMTE80]. Dans [BF97] un modèle de réseau de flots est présenté pour l'allocation des tâches temps réel avec respect des contraintes de temps dans le cas de tâches indépendantes.

2.3.2 Méthodes approchées

Les méthodes exactes, bien que produisant des solutions optimales ne sont pas généralement applicables à cause de leur temps de calcul prohibitif pour des problèmes réalistes. Les méthodes approchées permettent des donner des solutions qui dans la plupart des cas ne sont pas optimales, mais ont des temps de calcul acceptables qui les rendent applicables dans les problèmes de grandes tailles. Nous distinguons deux catégories de méthodes approchées, les méthodes de recherches non guidées qui ne dépendent pas du domaine étudié appelées aussi métaheuristiques et les méthodes guidées ou heuristiques qui dépendent du domaine étudié.

2.3.2.1 Méthodes de recherche non guidées ou métaheuristiques

Ce sont des méthodes d'optimisation globales inspirées de plusieurs domaines tels que l'intelligence artificielle, la biologie, la sidérurgie, etc. Nous distinguons celles qui sont basées sur la notion de population de solutions [Tal09, CCH$^+$99] (ensemble de solutions appelées individus) comme les algorithmes génétiques et celles qui se basent sur la recherche locale [Tal09, VM05, EL97] comme le recuit simulé. Dans cette dernière, la solution, déterminée à l'aide d'une fonction de coût, est remplacée par la meilleure solution située dans son voisinage. Le recuit simulé et la méthode Tabou font parties de ces métaheuristiques qui utilisent la recherche locale.

2.3.2.1.1 Recuit simulé

L'algorithme de recuit simulé [Tal09] commence par générer de façon aléatoire une solution initiale unique. À cette solution initiale sont associées une fonction d'énergie qui caractérise le système et une température pour permettre à l'algorithme de passer d'une phase à l'autre où la valeur de l'énergie diminue puis augmente pour diminuer par la suite jusqu'à l'obtention d'une valeur minimale, qui correspond à l'optimum global. Cette température est appelée "facteur d'agitation aléatoire". En ordonnancement temps réel la fonction d'énergie peut être une combinaison de plusieurs paramètres tels que le temps de réponse, les coûts de communications inter-processeurs, etc.

Dans [TBW92, CA95], le recuit simulé est utilisé pour résoudre le problème d'allocation des tâches temps réel avec des contraintes de précédences et des coûts de communication inter-processeurs.

2.3.2.1.2 Recherche Tabou

Un algorithme de recherche Tabou [Tal09, GL93] combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes lui permettant de surmonter l'inconvénient des optimums locaux, tout en évitant les problèmes de cycles (visiter plusieurs fois une solution). En effet il possède une mémoire (une liste tabou) qui garde les mouvements déjà effectués pour éviter d'y revenir et d'être piégé dans un cycle qui impliquera la convergence vers un optimum local. Cela permet d'explorer d'autres espaces de solutions et d'améliorer les solutions.

Les algorithmes de recherche tabou peuvent être utilisés dans la résolution du problème d'ordonnancement temps réel multiprocesseur avec des contraintes de précédences et des coûts de communication. [KBKP04, FGLM96].

2.3.2.1.3 Algorithmes génétiques

Les algorithmes génétiques [Tal09] s'inspirent de l'évolution biologique des espèces en exploitant la sélection et la survie de l'espèce la mieux adaptée. Le principe des algorithmes génétiques repose sur la sélection des individus qui représentent les solutions. La reproduction se fait dans le but de générer une autre population avec de meilleurs individus, en leur transmettant certaines de leur caractéristiques. Une fonction de coût est utilisée afin d'évaluer les individus et de sélectionner les meilleurs pour constituer une autre population, à laquelle on appliquera de nouveau la sélection et la reproduction. Ce processus est réitéré jusqu'à ce qu'un critère d'arrêt soit atteint. La reproduction des individus est réalisée avec le croisement et la mutation. Le croisement consiste à couper deux individus en un point qui peut être choisi aléatoirement et à échanger les parties coupées. La mutation, quand à elle, modifie les gènes des individus afin de produire d'autres individus (par exemple en ordonnancement on crée d'autres solution à partir d'une solution existante en modifiant l'allocation des tâches dans cette solution) [Tal09].

Dans [ESHA94, ESHA90], les algorithmes génétiques sont utilisés pour résoudre le problème d'ordonnancement temps réel multiprocesseur avec des contraintes de précédences.

2.3.2.2 Méthodes guidées ou heuristiques

Les méthodes d'allocation que nous présentons dans cette partie sont guidées par le domaine de l'ordonnancement temps réel multiprocesseur. Ce qui justifie l'utilisation par la suite de la notion de tâche, de processeur, de coûts de communication, etc. Ces méthodes heuristiques sont basées sur l'évolution d'une solution partielle (solution qui ne contient pas toutes les tâches à allouer). Bien que

ces heuristiques soient plus rapides que les métaheuristiques, à fortiori que les méthodes exactes, expérimentalement les heuristiques fournissent des solutions moins proches de l'optimal que les métaheuristiques.

2.3.2.2.1 Heuristiques gloutonnes

Les heuristiques gloutonnes construisent la solution incrémentalement. À chaque étape de l'heuristique gloutonne une tâche est rajoutée selon un critère glouton, correspondant à une décision locale déterminant le meilleur choix (à court terme) [Tal09, VM05]. Chaque choix fait est définitif, ces algorithmes n'autorisent pas les retour-arrières ("backtracking"). Ainsi, les solutions obtenues sont très dépendantes de l'ordre de considération des tâches. Les heuristiques gloutonnes donnent très rapidement des solutions qui sont rarement optimales.

Parmi les heuristiques gloutonnes nous avons les heuristiques utilisées dans la résolution du problème de remplissage de boîtes ("bin packing"). Par la suite nous utiliserons la terminologie heuristiques de *"bin packing"* pour désigner ces types d'heuristiques.

Dans le problème d'ordonnancement multiprocesseur, chaque tâche est modélisée par un objet et la taille de cet objet est modélisée par le facteur d'utilisation du processeur par cette tâche. Chaque processeur est modélisé par une boîte de capacité égale à la charge maximale que l'on peut atteindre sur ce processeur avec un algorithme d'ordonnancement fixé. Voici quelques heuristiques gloutonnes :

- heuristique "next-fit" :
  Étape 1: initialiser le compteur des tâches i=1
  Étape 2: initialiser le compteur des processeurs j=1
  Étape 3: Si τ_i est ordonnançable sur p_j (selon Liu et Layland [LL73] par exemple) alors allouer τ_i à p_j; Sinon $j = j + 1$ et allouer τ_i à p_j
  Étape 4: Si $i < n$ alors $i = i + 1$ et reprendre à partir de l'étape 3, sinon fin et système ordonnançable avec j processeurs;
- heuristique "first-fit" :
  avec "next-fit" si la tâche τ_i n'est pas ordonnançable sur p_j on passe immédiatement à p_{j+1} même si τ_i est ordonnançable sur l'un des $j - 1$ processeurs déjà utilisés. Ce qui a tendance à augmenter le nombre de processeurs. L'heuristique "first-fit" tente d'abord d'allouer une tâche au processeur, de plus petit indice j parmi les processeurs déjà utilisés et pouvant l'ordonnancer. Si cela est impossible on passe à un processeur vide;
- heuristique "best-fit" :

cette heuristique alloue une tâche au processeur en suivant une fonction coût f définie pour chaque processeur (par exemple la capacité restante du processeur). Ainsi une tâche est allouée au processeur de plus petite valeur de f qui peut l'ordonnancer. Si deux processeurs ont la même valeur de f, la tâche est allouée au processeur de plus petit indice;
- heuristique de "worst-fit" :
  c'est une heuristique similaire à "best-fit", mais avec "worst-fit", une tâche est allouée au processeur ayant la plus grande valeur de la fonction coût f.

Ces heuristiques de type "bin packing", cherchent uniquement à minimiser le nombre de processeurs utilisés dans le partitionnement des tâches, à l'exception de "best-fit" et "worst-fit" avec lesquels nous pouvons optimiser une fonction de coût définie.

2.3.2.2.2 Heuristiques de listes

Généralement ces méthodes heuristiques utilisent un graphe acyclique orienté (GAD) pour représenter les dépendances entres les tâches en associant des poids aux arcs (symbolisant les coûts de communication entre processeurs) qui relient les sommets (représentant les tâches), ainsi que des poids aux sommets (symbolisant les durées d'exécution des tâches). Les heuristiques de liste [SS97, ACD74, MKTM93] définissent une liste de tâches en attribuant à chacune d'elles une priorité. Cette liste détermine l'ordre du choix d'allocation d'une tâche aux différents processeurs. À chaque étape de l'heuristique, une tâche est sélectionnée dans la liste des tâches candidates pour être ordonnancée sur un processeur puis la liste est mise à jour (suppression de la tâche ordonnancée et ajout des tâches qui n'ont pas de prédécesseurs dans le graphe des tâches). Ceci est répété jusqu'à ce que la liste soit vide. Par opposition aux heuristiques gloutonnes, certaines approches des heuristiques de liste autorisent les retour-arrières [AaR95, Ram90] sur un choix quand une tâche ne peut pas être allouée à un processeur. Cela a pour conséquence d'augmenter considérablement le temps de recherche et selon Adan et al. [AaR95] et Ramamritham [Ram90] n'améliore pas toujours les résultats. Sans les retour-arrières les heuristiques de listes sont rapides d'exécution mais, comme avec toutes les heuristiques, il y a un risque de ne pas trouver une solution alors qu'il en existe.

2.3.2.2.3 Heuristiques de regroupement ou "clustering"

Les heuristiques de regroupement [CHMTE80, GVY90, KT00] consistent à regrouper les tâches en différents sous-ensembles appelés "clusters" (groupes)

en fonction des objectifs du problème (respect des contraintes temps réel, minimisation des coûts de communication, etc.), les tâches d'un "cluster" devant être ensuite allouées à un même processeur, ce qui diminue la taille du problème. La résolution du problème d'allocation par des heuristiques de regroupement est donc décomposée en deux phases. La première phase constitue le regroupement proprement dit et la seconde consiste à allouer les tâches sur les processeurs en fonctions des contraintes temps réel, de précédence entre les tâches, etc. Cette seconde phase peut être menée, par exemple, par des heuristiques de liste. Les heuristiques de regroupement ont l'avantage de réduire l'espace de recherche de solutions en explorant les "clusters" au lieu des tâches elles-mêmes. Par contre le regroupement des tâches en "clusters" peut engendrer la non satisfaction de contraintes (temps réel, minimisation du temps de réponse de l'ordonnancement, etc.).

2.3.2.2.4 Heuristique de duplication de tâches

Ces heuristiques dupliquent les tâches sur les processeurs afin de minimiser une fonction coût qui peut être le temps de réponse de l'ordonnancement des tâches. Dans [XTL10] est présentée une heuristique de duplication de tâches appelée "Heterogeneous Earliest Finish with Duplicator" (HEFD). L'heuristique HEFD utilise le principe des heuristiques de liste en définissant une liste pour les tâches candidates à l'ordonnancement. À chaque étape de l'heuristique une tâche candidate est sélectionnée puis sur chaque processeur la date de démarrage au plus tôt de la tâche est déterminée par duplication des tâches prédécesseurs immédiats de cette tâche candidate sur tous les processeurs sur lesquels elles ne sont pas encore dupliquées. Si la duplication d'une tâche prédécesseur immédiat sur un processeur entraîne un retard sur la date de démarrage au plus tôt de la tâche candidate alors cette duplication est supprimée. Ensuite le processeur produisant la plus petite date de démarrage pour la tâche est sélectionnée pour ordonnancer la tâche candidate.

2.4 Ordonnancement de tâches dépendantes

Comme nous l'avons vu à la section 1.4.3, une contrainte de dépendance entre tâches peut être une contrainte de précédence ou une contrainte de dépendances de données. Dans l'ordonnnancement de tâches avec des contraintes de précédences, il existe deux approches pour résoudre le problème. La première est basée sur les sémaphores [FGPR11] : un sémaphore est alloué à chaque précédence (τ_i, τ_j), et la tâche successeur τ_j doit attendre que la tâche prédécesseur τ_i libère le sémaphore avant de commencer son exécution. La deuxième approche est basées

sur la modification des priorités et des dates de premières activations des tâches [CSB90, FBG+10a]. Dans l'ordonnancement des tâches avec des contraintes de dépendances de données en plus des tenir compte des contraintes de précédences, il faut gérer le transfert et le partage de données entre tâches.

2.4.1 Transfert de données

Soient τ_i et τ_j deux tâches de périodes respectives T_i et T_j liées par une contrainte de dépendances de données telle que τ_i est la tâche productrice et τ_j la tâche consommatrice. Dans le cas général le transfert de données entre ces tâches se fait dans un rapport k relativement à leurs périodes tel que si $T_i < T_j$ alors $T_j = kT_i + r$, sinon $T_i = kT_j + r$ avec $k \in \mathbb{N}^* \; et \; r \in \mathbb{N}$. Dans le cas où τ_i a une période plus petite que τ_j alors τ_j consomme k données produites par τ_i. Dans le cas où τ_i a une période plus grande ou égale à celle de τ_j alors τ_j consomme une ou plusieurs fois la donnée produite par τ_i. Si $r \neq 0$, il peut arriver que la tâche productrice τ_i produise $k+1$ données avant que la tâche consommatrice τ_j ne consomme, sur ces $k+1$ données seules les k données sont consommées par τ_j et la donnée qui reste est perdue. Ce qui fait qu'au total on perdrait une infinité de données. Pour illustrer ce cas prenons l'exemple de deux tâches $\tau_1(0, 1, 3, 3)$ et $\tau_2(3, 1, 7, 7)$ telles que τ_1 est la tâche productrice de données et τ_2 la tâche consommatrice de données. Nous avons $T_2 = 7 = 2 * 3 + 1 = 2T_1 + 1$. Comme le montre la figure 2.1 ci-dessous, avant que la tâche consommatrice τ_2 ne démarre sa $4^{ème}$ exécution, la productrice s'est exécutée trois fois donc a fourni trois données dont deux seulement sont consommée à la $4^{ème}$ exécution de τ_2 et la donnée qui reste est perdue.

Il peut y avoir plusieurs possibilités pour la tâche τ_j de consommer les données produites par τ_i [FBG+10b]. La tâche τ_j peut consommer la totalité des données produites par τ_i ou n'importe quel sous-ensemble des données. Par exemple dans [RCR01], la tâche τ_j ne consomme que la dernière donnée produite par τ_i. Par contre dans [Ker09, KS07] toutes les données produites par τ_i sont consommées par τ_j sans qu'aucune données ne soient perdues.

2.4.2 Partage de ressources

Les dépendances de données entraînent des partages de données entre tâches. Les données produites par les tâches sont sauvegardées dans des mémoires considérées comme des ressources. Ces mémoires sont partagées entre les tâches productrices qui écrivent leurs données dans ces mémoires et les tâches consommatrices qui y lisent des données. Considérons deux taches τ_i et τ_j telles que τ_i produit des données que τ_j consomme. Lorsque τ_j se fait préempter par τ_i, les données dans la mémoire partagée par ces deux tâches peuvent être dans un état incon-

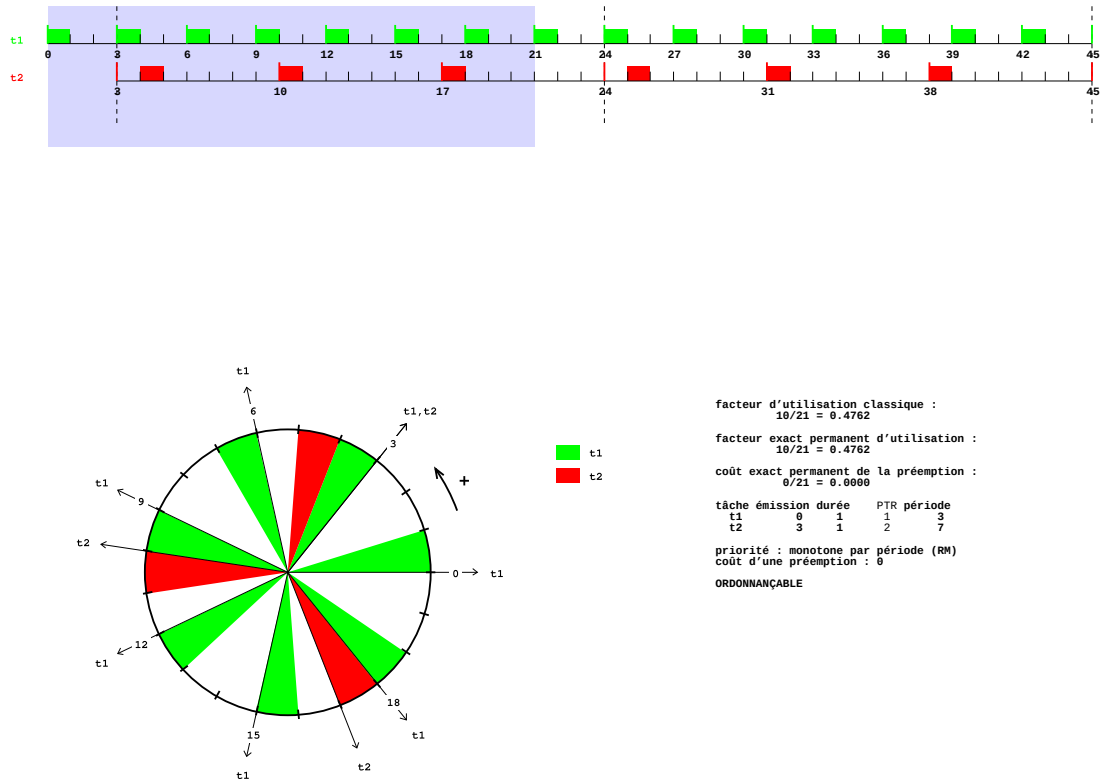

FIGURE 2.1 – Illustration des pertes de données

sistant car τ_i risque de modifier les données qui ne seront plus les mêmes à la reprise de l'exécution de τ_j. Donc τ_i et τ_j ne doivent pas accéder en même temps à la mémoire tampon qu'elles partagent. On dit que les deux tâches ont des contraintes d'exclusions mutuelles [Kai81, SRL90]. Pour traiter ces contraintes d'exclusion mutuelles, les tâches sont exécutés dans des sections critiques [Kai81]. Une tâche qui s'exécute en section critique ne pourra pas être préemptée par une autre tâche. Lorsque la tâche τ_i, la plus prioritaire, attend la fin de la section critique de τ_j pour pouvoir s'exécuter, on parle d'inversion de priorité [SRL90] et on dit que τ_i est bloquée par τ_j. Un exemple d'inversion de priorité est présenté à la figue 2.2.

On dit que deux ou plusieurs tâches sont en inter-blocage si chacune des tâches est bloquée en attente d'une ressource qui est utilisée par une autre tâche parmi ces tâches [Hav68]. Un exemple d'inter-blocage est présenté à la figure 2.3.

Les section critiques doivent être utilisées convenablement afin d'éviter que des tâches moins prioritaires bloquent indéfiniment des tâches plus prioritaires

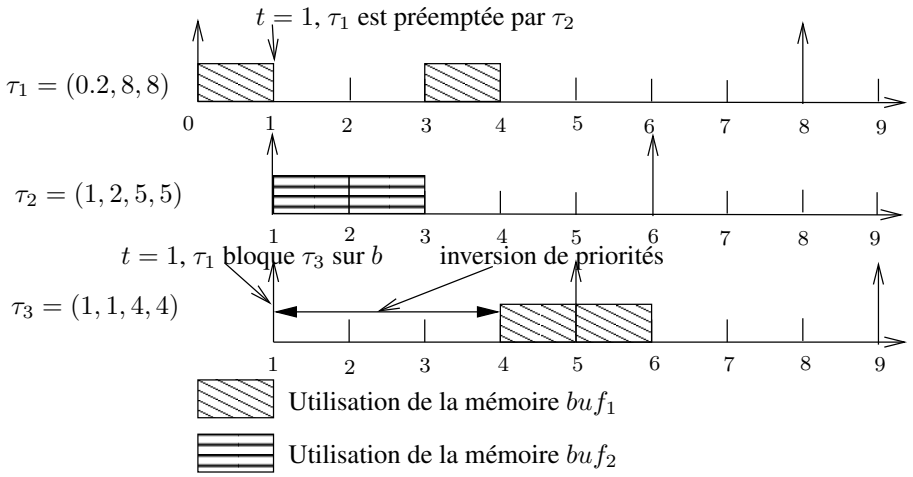

FIGURE 2.2 – Inversion de priorités

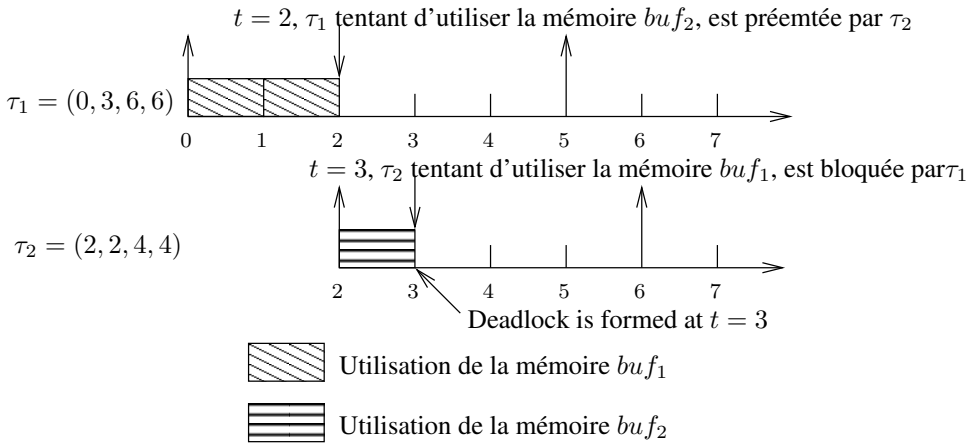

FIGURE 2.3 – Inter-blocage

mais aussi les inter-blocages. Pour cela plusieurs protocoles de synchronisation ont été proposés aussi bien en monoprocesseur qu'en multiprocesseur.

2.4.3 Protocoles de synchronisation monoprocesseur

2.4.3.1 Priority Inheritance Protocol (PIP)

Le protocole de synchronisation PIP (Priority Inheritance Protocol) a été proposé par Sha and al. [SRL90]. Il est basé sur le principe de l'héritage de priorité.

Pour réduire la durée des inversions de priorités, lorsqu'une tâche τ_j bloque une autre tâche τ_i plus prioritaire qu'elle, τ_j hérite de la priorité de τ_i durant toute la durée de son exécution. Après son exécution τ_j reprend sa priorité initiale. Ceci permet d'éviter qu'une tâche de priorité intermédiaire (une tâche plus prioritaire que τ_j et moins prioritaire que τ_i) préempte τ_j ce qui allongerait le temps de blocage de τ_i. Ce protocole permet de réduire le temps de blocage mais ne permet pas d'éviter les inter-blocages. Ce protocole est utilisé dans le cas des algorithmes d'ordonnancement à priorités fixes.

2.4.3.2 Priority Ceiling Protocol (PCP)

Le protocole de synchronisation PCP (Priority Ceiling Protocol) [SRL90] est une extension de PIP pour éviter les inter-blocages. Il est basé sur le principe de la priorité plafond . Une priorité plafond est donnée à chaque ressource. Cette priorité plafond est définie par la plus grande priorité des tâches qui utilisent cette ressource. À chaque instant t, on définit la priorité du système comme étant la plus grande priorité plafond parmi les ressources en cours d'utilisation à t. Une tâche τ_i ne pourra exécuter sa section critique que lorsque sa priorité est plus grande que la "priorité du système" qui est égale à la plus grande priorité parmi les priorités des ressources en cours d'utilisation. Sinon τ_j continue son exécution et hérite de la priorité de τ_i. Ceci permet d'éviter les inter-blocages.

2.4.3.3 Stack Resource Policy (SRP)

Le protocole SRP (Stack Resource Policy MSRP) [Bak91], est la version de PCP avec des priorités dynamiques. Chaque instance d'une tâche possède un plafond de préemption. Une instance de la tâche τ_i ne pourra préempter une instance de la tâche τ_j que lorsque la priorité de l'instance de τ_i est supérieure au plafond de préemption de l'instance de la tâche τ_i. Chaque ressource a un plafond de préemption qui est le maximum des plafonds de préemption des instances pouvant utiliser cette ressource. On définit le plafond de préemption du système comme étant le plus grand plafond de préemption des ressources en cours d'utilisation. Une tâche τ_i ne peut exécuter une section critique que lorsque sa priorité est plus grande que le plafond de préemption de toutes les tâches actives et que son plafond de préemption est supérieur à celui du système. Les instances actives sont placées dans une pile dans l'ordre décroissant de leur priorité.

2.4.4 Protocoles de synchronisation multiprocesseur

2.4.4.1 MPCP

Le protocole MPCP (Multiprocessor Priority Ceiling Protocol) [Raj90] est une extension en multiprocesseur de PCP. Avec MPCP on distingue deux types de ressources. Les ressources locales sont partagées entre tâches sur le même processeur et les ressources globales sont partagées entre tâches allouées sur des processeurs différents. Chaque ressource globale a une priorité plafond égale à la plus grande priorité parmi celles des tâches pouvant utiliser cette ressource et la plus grande priorité du système (la plus grande priorité des ressources en cours d'utilisation sur les processeurs). Une tâche qui utilise une ressource globale est exécutée à la priorité de cette ressource. Dans le cas des ressources locales, MPCP fonctionne de la même manière que PCP.

2.4.4.2 MSRP

Le protocole MSRP (Multiprocessor Stack Resource Policy MSRP) [GLN01] est une extension en multiprocesseur de SRP. Chaque processeur possède son propre plafond de préemption. Chaque processeur attribue à une ressource globale un plafond de préemption supérieur ou égal au plafond de préemption sur ce processeur. Lorsqu'une tâche utilise une ressource globale sur un processeur, ce processeur hérite du plafond de priorité de cette ressources globale. Dans le cas des ressources locales, MSRP fonctionne de la même manière que SRP.

2.5 Ordonnancement avec prise en compte du coût de l'OS

Le coût du système d'exploitation est composé de deux parties. La première est appelée coût de l'ordonnanceur. Elle correspond au coût de sauvegarde de contexte du processeur, au coût de sélection de la tâche à exécuter et au coût de chargement de la tâche sélectionnée [KAS93]. La seconde partie est appelée coût de la préemption dû à l'ensemble des préemptions [Yom09]. Le coût d'une préemption correspond au coût de sauvegarde de contexte du processeur, de sélection de la tâche à exécuter et de restauration de contexte du processeur au moment de chaque préemption. Les travaux que nous avons présentés jusqu'à présent font l'hypothèse que le coût de l'OS est négligeable ou est approximé dans le pire temps d'exécution de chaque tâche.

2.5.1 Coût de l'ordonnanceur

Le coût de l'ordonnanceur est illustré sur la figure 2.4. Ce coût se décompose en :

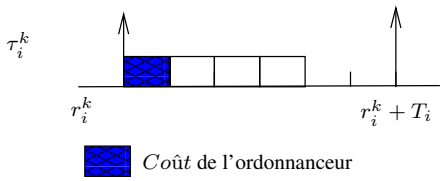

FIGURE 2.4 – Illustration du coût de l'ordonnanceur

- coût de sauvegarde de contexte du processeur,
- coût de sélection de la tâche à exécuter,
- coût de chargement pour exécution de la tâche sélectionnée.

La sauvegarde du contexte de la tâche en cours d'exécution et de chargement de la tâche sélectionnée ont un coût fixe. Dans le cas d'un ordonnancement en ligne la sélection d'une tâche s'effectue pendant l'exécution des tâches. Cette sélection de la tâche à exécuter dépend du nombre de tâches dans l'état "Prêt" et donc son coût varie. Par contre dans le cas d'un ordonnancement hors ligne, la sélection d'une tâche s'effectue à partir d'une table d'ordonnancement qui a été déterminée avant l'exécution des tâches. Le coût de sélection est fixe et égal au coût de lecture dans une table.

Dans la suite comme nous nous intéressons à l'ordonnancement hors ligne, le coût de l'ordonnanceur devient fixe. Pour le prendre en compte dans l'analyse d'ordonnançabilité, il suffira de l'ajouter au pire temps d'exécution de chaque tâche. Dans la suite du manuscrit on expliquera comment prendre en compte le coût exact de la préemption.

2.5.2 Coût de la préemption

Lorsqu'une tâche se fait préempter par une autre tâche plus prioritaire, son contexte d'exécution est sauvegardé et celui-ci doit être restauré lors de sa re-sélection. Le coût de sauvegarde de contexte est pris en compte dans la durée d'exécution de la tâche qui préempte plutôt que dans la durée d'exécution de la tâche préemptée car, le coût de l'ordonnanceur qui est supposé inclus dans la durée d'exécution des tâches contient aussi la sauvegarde de contexte. Par contre la restauration du contexte de la tâche préemptée a lieu uniquement lors de sa reprise d'exécution. Cette restauration du contexte de la tâche préemptée est précédée par

une sélection de tâche et par une sauvegarde de contexte d'exécution de la tâche en cours. Cet ensemble de coût de sélection, de sauvegarde de contexte d'exécution de la tâche en cours et de restauration de contexte de la tâche préemptée, est appelé coût d'une préemption. Ce coût temporel de la préemption rallonge le temps de réponse de l'exécution de la tâche préemptée et peut causer à son tour d'autres préemptions pour d'autres tâches moins prioritaires. Cette accumulation de préemptions peut rendre non ordonnançable un système de tâches qui était initialement déclaré ordonnançable en négligeant le coût de la préemption. La figure 5.5.2 illustre le coût de la préemption.

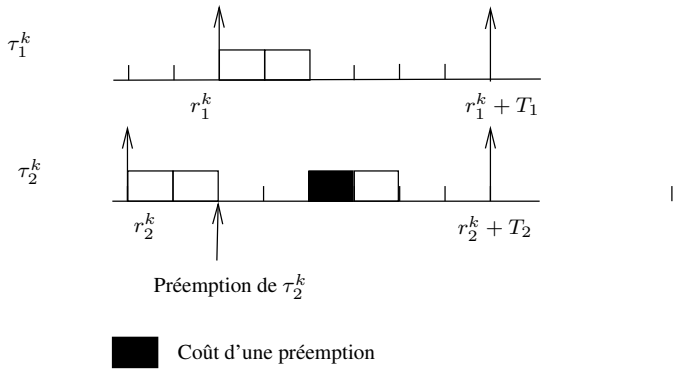

FIGURE 2.5 – Illustration du coût de la préemption

Pour éviter de prendre en compte le coût de la préemption qui a un impact sur l'ordonnançabilité, certains travaux proposent de réduire le nombre de préemption [Bur94, WS99, SW00]. Dans [Bur94], un algorithme d'ordonnancement à priorités fixes et à préemptions différées est proposé. Cet algorithme considère que chaque exécution ou instance d'une tâche est constituée de plusieurs sous-exécutions et une préemption ne peut avoir lieu qu'entre deux sous-exécutions. Ceci permet de réduire le nombre de préemptions d'une tâche au cours de son exécution. Dans [WS99, SW00] un autre algorithme d'ordonnancement avec plafond de priorité est proposé. Cet algorithme assigne à une tâche deux types de priorité : un priorité nominale et une priorité plafond. Avec cet algorithme, une préemption ne peut avoir lieu que lorsque la tâche qui préempte a une priorité nominale plus grande que la priorité de seuil de la tâche qu'elle veut préempter. Donc en choisissant comme il faut les priorités de seuil des tâches on peut réduire le nombre de préemptions. Cependant aucun de ces travaux ne prend en compte dans l'ordonnancement le coût lié à chaque préemption.

En revanche d'autres travaux ont pris en compte le coût de la préemption. Dans le modèle de tâche Liu et Layland [LL73] le coût de la préemption est ap-

proximé dans le WCET. Cette approximation se fait soit en prenant une marge faible, ce qui peut rendre un système de tâches non ordonnançables à l'exécution alors que ce système a été conclu ordonnançable. Ou bien l'approximation est faite en prenant une marge importante ce qui entraîne le gaspillage de la ressource processeur. C'est la raison pour laquelle nous devons prendre en compte le coût exact de la préemption, c'est-à-dire le coût lié à l'occurrence de chaque préemption. A. Burns and al. dans [BTW95] ont essayé de résoudre ce problème en présentant une analyse d'ordonnançabilité basée sur le temps de réponse qui prend en compte le coût global lié au nombre maximum de préemptions. Sauf que dans ce cas on une borne maximum du nombre de préemptions et non pas le nombre exact. Dans [ERC95], Ripoll and al. se sont eux aussi intéressés au nombre exact de préemption en faisant l'ordonnancement tâche après tâche et en comptant le nombre de préemption. Mais ils ne prennent pas en compte les préemption engendrées par d'autres préemptions car ils n'ajoutent pas le coût lié à l'occurrence de chaque préemption.

Dans [Yom09] Meumeu et Sorel ont proposé une analyse d'ordonnançabilité d'un ensemble de tâches indépendantes périodiques qui permet de prendre en compte le coût exact de la préemption. Cette analyse est basée sur une opération binaire d'ordonnancement appelée $\oplus$ qui permet de compter le nombre exact de préemptions de chaque instance d'une tâche puis ajoute le coût associé à ces préemptions à son WCET ce qui permet d'obtenir une nouvelle durée d'exécution de la tâche avec prise en compte du coût exact de la préemption. Cette durée appelée PET ("Préemption Execution Time") est utilisée dans l'analyse d'ordonnançabilité des tâches.

Puisque prendre en compte le coût exact de l'ordonnanceur revient à rajouter son coût fixe dans le WCET de chaque tâche, ceci avant l'exécution des tâches, on supposera que le coût de l'ordonnanceur est inclus dans le WCET des tâches. Donc prendre en compte le coût exact de l'OS revient à prendre en compte le coût exact de la préemption. Donc dans la suite nous concentrerons notre étude sur le coût exact de la préemption.

2.6 Conclusion

Dans ce chapitre nous avons présenté un état de l'art sur l'ordonnancement temps réel en mettant l'accent sur les méthodes d'allocation des tâches dans l'ordonnancement multiprocesseur, sur la prise en compte du coût exact de la préemption et sur les dépendances de données.

Dans la seconde partie de cette thèse nous étudierons l'ordonnancement multiprocesseur des tâches indépendantes avec prise en compte du coût exact de la préemption. Pour cela nous réutiliserons les concepts définis dans cet état de l'art.

Deuxième partie

Ordonnancement temps réel multiprocesseur de tâches indépendantes avec prise en compte du coût exact de la préemption

Chapitre 3

Ordonnancement monoprocesseur de tâches indépendantes

3.1 Introduction

Comme nous l'avons présenté dans l'état de l'art, la prise en compte du coût exact de la préemption dans l'analyse d'ordonnançabilité de tâches temps réel est peu étudiée. Les seuls travaux que nous avons trouvés dans l'état de l'art sont ceux de Meumeu et Sorel [Yom09] qui ont présenté l'opération d'ordonnancement $\oplus$ dans le cas de l'ordonnancement monoprocesseur. Ce chapitre donne une présentation simplifiée de l'opération $\oplus$. Nous allons aussi dans ce chapitre prouver la viabilité ("sustainabilty") de l'analyse d'ordonnançabilité basée sur cette opération dans le cas d'un ordonnancement à priorités fixes.

3.2 Modèle de tâches

Considérons l'ensemble Γ_n de tâches temps réel périodiques indépendantes concrètes. Chaque tâche $\tau_i = (r_i^1, C_i, T_i, D_i)$ est caractérisée par une date de première activation r_i^1, une pire durée d'exécution C_i n'incluant pas les coûts de préemption contrairement au modèle classique de Liu et Layland [LL73], une période T_i et une échéance relative D_i. Les tâches étant périodiques, la date de la $k^{ème}$ activation de τ_i est donnée par $r_i^k = r_i^k + (k-1)T_i$. La tâche τ_i doit terminer son exécution avant la date $d_i^k = r_i^k + D_i$. Le temps qui s'écoule entre la date d'activation et la date de fin d'exécution de la $k^{ème}$ activation de τ_i désigne le temps de réponse R_i^k de la $k^{ème}$ instance de τ_i. On suppose que l'inégalité $C_i \leq D_i = T_i$ est toujours vérifiée.

Nous supposons aussi que le processeur que nous considérons n'a pas de cache, ni "pipeline" ou d'architecture interne complexe.

3.3 Intervalle d'étude de l'ordonnancement

L'opération d'ordonnancement $\oplus$ utilise l'approche par simulation présentée à la section 1.5.3.2 de l'état de l'art pour faire l'analyse d'ordonnançabilité des tâches. Cette simulation est réalisée sur un intervalle d'étude défini. Cet intervalle est déterminé à l'aide du théorème 3.3.1 ci-dessous.

Théorème 3.3.1 *[Yom09] Soit $\Gamma_n = \{\tau_1, \tau_2, \cdots, \tau_n\}$ un ensemble de n tâches périodiques indépendantes concrètes tel que $\tau_i = (r_i^1, C_i, T_i, D_i)$, avec $i = 1, \cdots, n$, rangées par priorités décroissantes relativement à un algorithme d'ordonnancement à priorités fixes noté $Algo$, c'est-à-dire $i < j$ implique que la priorité de τ_i est supérieure à la priorité de τ_j. Soit $(s_i)_{i\in\mathbb{N}^*}$ la suite définie par*

$$\begin{cases} s_1 = r_1^1 \\ \\ s_i = r_i^1 + \left\lceil \frac{(s_{i-1} - r_i^1)^+}{T_i} \right\rceil \cdot T_i, \quad 2 \leq i \leq n \end{cases}$$

S'il existe un ordonnancement valide de Γ_n jusqu'à la date $s_n + H_n$ où $H_n = ppcm\{T_i \,|\, i = 1, \cdots, n\}$ et $x^+ = max(x, 0)$, alors cet ordonnancement est valide et périodique de période H_n à partir de s_n.

D'après le théorème 3.3.1, l'exécution de l'ensemble des tâches Γ_n est périodique à partir de la date s_n et sa période est égale à $H_n = ppcm\{T_i | i = 1, \cdots, n\}$. L'intervalle défini par $[r_{min}, s_n]$ avec $r_{min} = min\{r_i^1\}_{\tau_i \in \Gamma_n}$ est appelé phase transitoire et l'intervalle défini par $[s_n, s_n + H_n]$ est appelé la phase permanente qui se répète indéfiniment. Donc un intervalle d'étude de l'ordonnancement de Γ_n est donné par $I_n = [r_{min}, s_n + H_n]$. En considérant uniquement l'ordonnancement des i premières tâches on parle d'ordonnancement de niveau i et un intervalle d'étude de l'ordonnancement des i tâches est donné par $I_i = [r_{min}, s_i + H_i]$ avec $H_i = ppcm\{T_j \,|\, j = 1, \cdots, i\}$.

3.4 Opération binaire d'ordonnancement $\oplus$

3.4.1 Principe

L'opération d'ordonnancement $\oplus$ a été proposée par Meumeu [Yom09]. C'est une opération binaire, c'est-à-dire qu'elle ne s'applique qu'entre deux opérandes : un opérande de gauche op_1 appelé tâche exécutée et un opérande de droite op_2 appelé tâche exécutable telles que le résultat $\mathcal{R}$ est donné par $\mathcal{R} = op_1 \oplus op_2$. Par convention l'opérande de gauche est toujours plus prioritaire que l'opérande de droite. On représente une instance de la tâche exécutable op_2 par une séquence

de symboles de "$\mathbf{e}_2$" en gras suivi par une séquence de symbole de "a". Chaque symbole de "$\mathbf{e}_2$" de la tâche exécutable représente une unité de temps exécutable, c'est-à-dire une unité de temps que op_2, la tâche à exécuter, doit exécuter. Chaque symbole "a" représente une unité de temps libre. Nous représentons une instance de la tâche exécutée op_1 par de séquences de symbole de "e" suivi par de séquences de symboles de "a" qui ne se suivent pas forcément et qui peuvent se répéter indéfiniment. Chaque symbole "e_1" de la tâche exécutée op_1 représente une unité exécutée par op_1, c'est-à-dire une unité de temps que op_1 a déjà exécutée.

Le principe de l'opération $\oplus$ est de remplacer les unités de temps "a" libres de la tâche exécutée op_1 par les unités de temps exécutables "$\mathbf{e}_2$" de la tâche op_2. Pour cela, puisque les deux opérandes ne sont pas comparables, on les référence par rapport à la même origine de temps, puis on répète l'exécution de op_1 autant de fois qu'il y a d'instances de op_1 sur l'intervalle d'étude I_2. Ensuite on réécrit le premier opérande op_1 en fonction du nombre d'instances du second opérande op_2 sur l'intervalle d'étude I_2. Le résultat $\mathcal{R} = op_1 \oplus op_2$ est une tâche exécutée, composée de séquences de symboles de "e_1", de séquences de symboles de "e_2", suivies de séquences de symboles de "a" qui ne se suivent pas forcément et qui peuvent se répéter indéfiniment.

Soient deux tâches périodiques concrètes $\tau_i = (r_i^1, C_i, T_i, D_i)$ et $\tau_{i+1} = (r_{i+1}^1, C_{i+1}, T_{i+1}, D_{i+1})$. On suppose que τ_i est plus prioritaire que τ_{i+1}. On veut ordonnancer τ_i et τ_{i+1} en utilisant $\oplus$. τ_i étant la plus prioritaire on a $\mathcal{R} = \tau_i \oplus \tau_{i+1}$. On étudie l'ordonnançabilité de τ_i et τ_{i+1} sur l'intervalle $I_{i+1} = [r_{min}, S_{i+1} + H_{i+1}]$ avec $r_{min} = min\{r_j^1\}_{\tau_j \in \{\tau_i, \tau_{i+1}\}}$ et S_{i+1}, déterminé à l'aide du théorème 3.3.1. L'exécution de τ_i est périodique de période T_i et se répète indéfiniment à partir de r_i^1. Le coût d'une préemption est supposé être une constante donnée par α.

Définition 3.4.1 *La durée d'exécution de la tâche τ_{i+1} sur son instance k avec prise en compte du coût exact de la préemption appelée "Preemption Execution Time" (PET) est donnée par :*

$$C_{i+1}^k = C_{i+1} + np_{i+1}^k * \alpha$$

np_{i+1}^k est le nombre exact de préemption de τ_{i+1} sur son instance k.

La figure 3.1 donne une illustration du PET.

Le nombre np_{i+1}^k de préemptions de la tâche τ_{i+1} sur son instance k est calculé selon le principe suivant :

– np_{i+1}^k est initialisé à zéro au début de l'instance k,

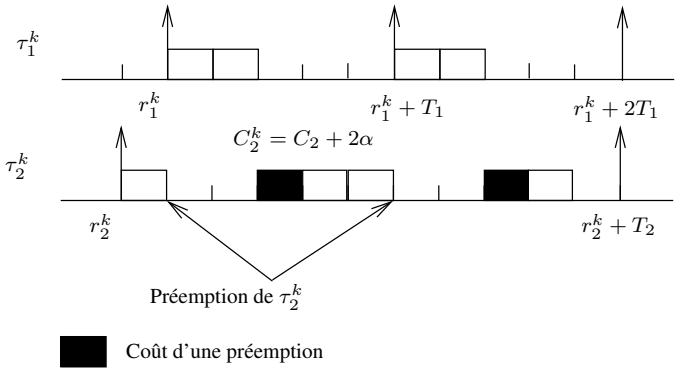

FIGURE 3.1 – Illustration du PET de l'instance k de τ_2

- à chaque fois que τ_{i+1} est préemptée par τ_i, np_{i+1}^k est incrémenté de 1 et on ajoute α unités de temps à la durée d'exécution qui reste à τ_{i+1} pour finir son exécution. Ceci permet de prendre en compte les préemptions engendrées par d'autres préemptions;
- à la fin de l'instance k, np_{i+1}^k correspond au nombre exact de préemptions de τ_{i+1}.

L'algorithme 1 ci-dessous présente les différentes étapes de l'ordonnancement de τ_i et τ_{i+1} utilisant $\oplus$.

3.4.2 Application

Pour illustrer l'ordonnancement avec $\oplus$ on considère les tâches $\tau_1 = (1, 2, 4; 4)$ et $\tau_2 = (0, 2, 6, 6)$ avec τ_1 plus prioritaire que τ_2. L'intervalle d'étude de l'ordonnancement de τ_1 et τ_2 est calculé à l'aide du théorème 3.3.1. Cet intervalle est donné par $I_2 = [0, 18]$. La figure 3.2 ci-dessous donne les différentes étapes de l'ordonnancement avec $\oplus$.

3.5 Analyse d'ordonnançabilité avec $\oplus$

Soit un ensemble de tâches périodiques concrètes $\Gamma_n = \{\tau_1, \tau_2 \cdots, \tau_n\}$ où les tâches sont triées dans l'orde décroissant de leurs priorités selon l'algorithme à priorités fixes noté $Algo$. Puisque l'opération d'ordonnancement $\oplus$ ne s'applique qu'entre deux opérandes à la fois, alors $\oplus$ est appliquée $n - 1$ fois et de façon itérative pour faire l'analyse d'ordonnancement de l'ensemble Γ_n. Ainsi si $\mathcal{R}_n$ est

Algorithme 1 $\mathcal{R} = \tau_i \oplus \tau_{i+1}$

1: Calculer l'intervalle d'étude de l'ordonnancement de τ_i et τ_{i+1} noté I_{i+1}
2: Référencer τ_i et τ_j par rapport à la même origine de temps $r_{min} = min\{r_j^1\}_{\tau_j \in \{\tau_i, \tau_{i+1}\}}$
3: Répéter l'exécution de τ_i autant de fois qu'il y a d'instances de τ_i sur l'intervalle I_{i+1}. Ceci est possible car τ_i est la plus prioritaire donc elle peut s'exécuter sans être préemptée
4: Délimiter toutes les instances de τ_{i+1} sur l'intervalle I_{i+1}. Soit n_{i+1} le nombre d'instances de τ_{i+1} sur I_{i+1}
5: $ordonnançable \leftarrow vrai$
6: $k \leftarrow 1$
7: **tant que** $(k \leq n_{i+1})$ et $(ordonnançable = vrai)$ **faire**
8: Calculer C_{i+1}^k, la durée d'exécution de τ_{i+1} sur son instance k avec le coût exact de la préemption
9: Compter le nombre d'unités de temps libres "a" sur l'instance k de τ_i. On note par $idle^k$ ce nombre.
10: **si** $idle^k < C_{i+1}^k$ **alors**
11: $ordonnançable \leftarrow faux$
12: **sinon**
13: τ_{i+1} ne rate pas son échéance sur l'instance k
14: Remplacer les C_{i+1}^k premières unités de temps libres "a" par des unités de temps e_{i+1} exécutées par τ_{i+1}
15: **fin si**
16: **fin tant que**
17: **si** $ordonnançable = vrai$ **alors**
18: τ_i et τ_{i+1} sont ordonnançables et $\mathcal{R}$ le résultat de l'ordonnancement de τ_i et τ_{i+1} est donné par I_{i+1} avec des exécutions de τ_i et de τ_{i+1}. Cet ordonnancement est périodique de période $H_{i+1} = ppcm(T_i, T_{i+1})$ et se répète indéfiniment à partir de s_{i+1} déterminé à l'aide du théorème 3.3.1
19: **sinon**
20: τ_i et τ_{i+1} ne sont pas ordonnançables car il existe une instance de τ_{i+1} dans laquelle τ_{i+1} n'est pas ordonnançable.
21: **fin si**

le résultat de l'ordonnancement de Γ_n, alors $\mathcal{R}_n$ est obtenu par :

$$\begin{cases} \mathcal{R}_1 = \tau_1 \\ \mathcal{R}_{i+1} = \mathcal{R}_i \oplus \tau_{i+1}, \quad 1 \leq i < n \end{cases}$$

À la fin de l'analyse d'ordonnançabilité, nous pouvons produire la table d'ordonnancement des tâches qui contient les dates de début et de fin d'exécution de chaque tâche.

Il faut noter que c'est le choix de priorité qui détermine l'ordre d'application de $\oplus$, de la tâche la plus prioritaire à la moins prioritaire.

La complexité de l'analyse d'ordonnançabilité de Γ_n basée $\oplus$ est évaluée à $O(n.l)$ avec $l = ppcm\{T_i : \tau_i \in \Gamma_n\}$.

3.6 Facteur d'utilisation du processeur avec coût de la préemption

Le facteur d'utilisation classique du processeur pour l'ensemble de tâches $\Gamma_n = \{\tau_1, \tau_2, \cdots, , \tau_n\}$ avec $\tau_i = (r_i^1, C_i, T_i, D_i)$ est donné par :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Lorsque le coût de la préemption est prise en compte, la durée d'exécution d'une tâche τ_i avec le coût de la préemption sur une instance k est donnée par $C_i^k \geq C_i$. C_i^k est obtenu en ajoutant sur la durée d'exécution C_i le coût associé au nombre de préemptions de τ_i sur l'instance k. On note par n_i le nombre d'instances de la tâche τ_i sur les deux phases (transitoire et permanente) de l'ordonnancement.

Définition 3.6.1 *Le facteur d'utilisation du processeur avec prise en compte du coût exact de la préemption noté U^* est défini par :*

$$U^* = \sum_{i=1}^{n} \frac{C_i^*}{T_i} \; avec \; C_i^* = \sum_{k=1}^{n_i} \frac{C_i^k}{n_i} \tag{3.1}$$

3.7 Impact du coût de la préemption dans l'analyse d'ordonnançabilité

Dans cette section nous présentons l'impact du coût de la préemption dans l'analyse d'ordonnançabilité des tâches. Pour cela nous considérons deux cas.

Dans le cas 1 on suppose que le coût de la préemption est négligeable ($\alpha = 0$). Dans le cas 2 on suppose que le coût de la préemption n'est pas approximé dans la pire durée d'exécution des tâches ($\alpha = 1$). Pour montrer l'impact du coût de la préemption, nous comparons les taux de succès de l'analyse d'ordonnançabilité dans les deux cas. Soit un ensemble E composé de plusieurs ensembles de tâches, le taux de succès d'un algorithme A sur E est défini par :

$$\frac{nombre\ d'ensembles\ de\ tâches\ ordonnançables\ avec\ A}{nombre\ total\ d'ensembles\ de\ tâches\ dans\ E} \tag{3.2}$$

Afin de comparer les taux de succès de l'analyse d'ordonnançabilité avec $\oplus$, nous générons aléatoirement, en utilisant l'outil FORTAS [CG11] fondé sur l'algorithme de Bini [BB05], 15 ensembles chacun composé de 10 ensembles de tâches. Chaque ensemble de tâches est composé de 10 tâches. On calcule le taux de succès de $\oplus$ dans les deux cas et sur chaque ensemble d'ensembles de tâches. La courbe des taux de succès dans chacun des deux cas et les résultats sont présentés par la figure 3.3. Sur l'axe des ordonnées nous avons les taux de succès et sur l'axe des abscisses, les moyenne des facteurs d'utilisation des ensembles de tâches. En ce qui concerne les conditions techniques de test, l'ordinateur utilisé est un DELL muni d'un processeur $IntelCore^{TM}$ 2 Duo (2,66 GHz, mémoire RAM 4Go).

Dans la figure 3.3, on observe que jusqu'à un facteur d'utilisation moyen inférieur à 0.8, dans les deux cas nous avons des taux de succès de 1. Ce qui signifie que le coût des préemptions éventuelles dans le cas 2 n'affecte pas l'ordonnançabilité des ensembles de tâches. À partir d'un facteur d'utilisation moyen supérieur à 0.77, le taux de succès dans le cas 2 décroît jusqu'à 0. Par contre dans le cas 1, pour les ensembles d'ensembles de tâches avec des facteurs d'utilisation moyens entre 0.8 et 0.9, le taux de succès reste égal à 1 avant de décroître à 0.8. Ceci signifie que parmi les ensembles d'ensembles de tâches de facteurs d'utilisation moyen supérieur à 0.9 aucun ensemble de tâches n'est ordonnançable avec la prise en compte du coût exact de la préemption par contre sans prise en compte du coût de la préemption certains ensembles de tâches sont ordonnançables. Cela confirme le fait qu'en prenant en compte le coût exact de la préemption dans l'analyse d'ordonnançabilité des tâches, un ensemble de tâches peut être non ordonnançable alors qu'il a été déclaré ordonnançable en négligeant le coût de la préemption. Ceci dépend des facteurs d'utilisation des tâches et aussi du coût de chaque préemption. Dans notre exemple ceci semble être vérifié uniquement pour des tâches qui chargent beaucoup le processeur, mais même dans le cas contraire si le coût d'une préemption est élevé ce fait reste vérifié.

3.8 Viabilité

Dans cette section nous allons montrer que l'analyse d'ordonnançabilité basée sur $\oplus$ est viable. Cette viabilité est prouvée si un ensemble de tâches est ordonnançable avec $\oplus$, alors cet ensemble de tâches reste ordonnançable, même si une ou plusieurs tâches ont des durées d'exécution plus petites que celles considérées lors de l'analyse d'ordonnançabilité.

Théorème 3.8.1 *Soit un ensemble de tâches périodiques concrètes* $\Gamma_n = \{\tau_1, \cdots, \tau_i, \cdots, \tau_n\}$ *où les tâches sont triées dans l'ordre décroissant de leurs priorités selon l'algorithme à priorités fixes noté* $Algo$ *avec* $\tau_i = (r_i^1, C_i, T_i, D_i)$. *Si* Γ_n *est ordonnançable avec* $\oplus$, *lors de l'exécution en remplaçant* $\tau_i^{'} = (r_i^1, C_i^{'}, T_i, D_i)$ *par* τ_i *avec* $C_i^{'} \leq C_i$ *alors* $\Gamma_n^{'} = \{\tau_1, \cdots, \tau_i^{'}, \cdots, \tau_n\}$ *est aussi ordonnançable.*

Preuve on suppose que $\Gamma_n = \{\tau_1, \cdots, \tau_i, \cdots, \tau_n\}$ est ordonnançable avec $\oplus$. Nous allons montrer par contradiction que $\Gamma_n^{'} = \{\tau_1, \cdots, \tau_i^{'}, \cdots, \tau_n\}$ est aussi ordonnançable avec $\oplus$. Pour cela supposons que $\Gamma_n^{'}$ n'est pas ordonnançable avec $\oplus$. Ce qui veut dire que $\exists \tau_l \in \Gamma_n^{'}$ tel que τ_l n'est pas ordonnançable. Puisque les $i-1$ premières tâches de Γ_n sont les plus prioritaires et sont les mêmes que dans Γ_n alors leur ordonnancement ne change pas donc $i \leq l \leq n$.

Supposons que $l = i$ ce qui veut dire que c'est la tâche $\tau_i^{'}$ qui n'est pas ordonnançable, alors il existe une instance k de $\tau_i^{'}$ dans laquelle on ne peut exécuter toutes les $C_i^{'k}$ unités de temps de la tâche $\tau_i^{'}$. Puisque nous avons des tâches indépendantes avec des priorités fixes et que de plus τ_i et $\tau_i^{'}$ ont la même priorité donc si $C_i^{'} \leq C_i$ alors $C_i^{'k} \leq C_i^k \ \forall k \geq 1$. Il existe alors une instance k de la tâche τ_i dans laquelle on ne peut pas exécuter toutes les C_i^k unités de temps de τ_i. Ce qui veux dire que τ_i n'est pas ordonnançable donc Γ_n n'est pas ordonnançable. Ce qui est absurde d'après notre hypothèse de départ disant que Γ_n est ordonnançable.

De la même manière on montre que si $i < l \leq n$ alors Γ_n n'est pas ordonnançable. Donc on a prouvé que si $\Gamma_n^{'}$ n'est pas ordonnançable alors Γ_n ne l'est pas. D'où Γ_n ordonnançable entraîne que $\Gamma_n^{'}$ soit aussi ordonnançable.□

3.9 Conclusion

Dans ce chapitre nous avons présenté le principe de l'analyse d'ordonnançabilité basée sur l'opération binaire d'ordonnancement $\oplus$ qui permet de prendre en compte le coût exact de la préemption en monoprocesseur. Ensuite nous avons montré que cette analyse d'ordonnançabilité est viable c'est dire si la durée d'exécution des tâches diminue au cours de leur exécution alors l'ensemble des tâches reste toujours ordonnançable. Dans le chapitre suivant, nous allons utiliser cette

analyse d'ordonnançabilité basée sur $\oplus$ dans l'étude de l'ordonnancement multi-processeur de tâches indépendantes.

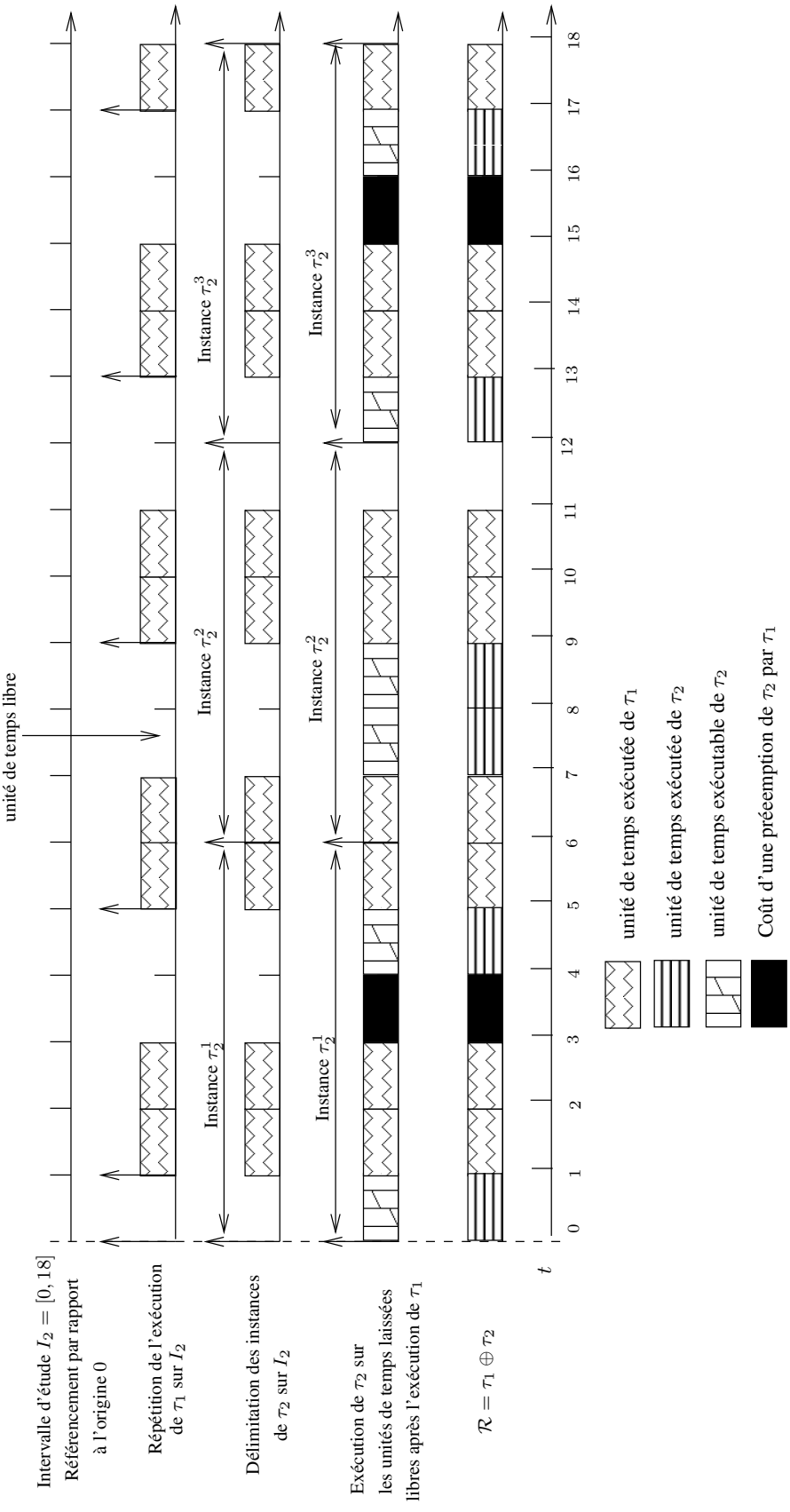

FIGURE 3.2 – Opération $\oplus$ entre deux tâches

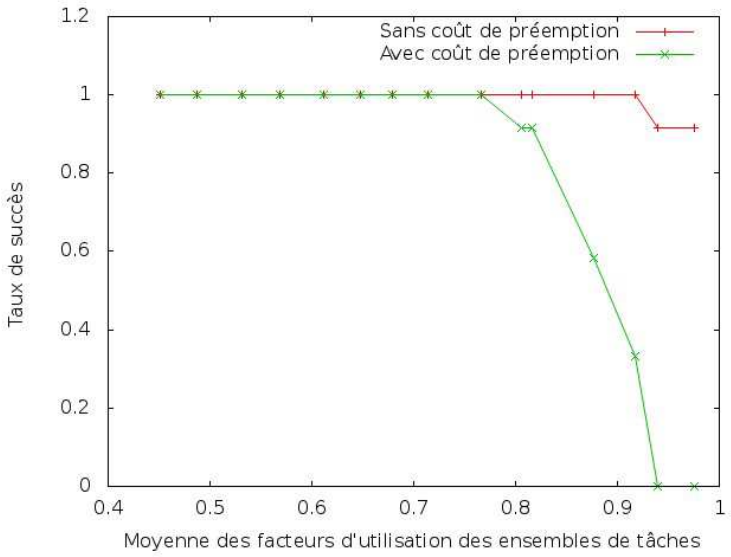

FIGURE 3.3 – Impact du coût de la préemption

Chapitre 4

Ordonnancement multiprocesseur de tâches indépendantes

4.1 Introduction

Dans ce chapitre nous allons étudier l'ordonnancement multiprocesseur de tâches temps réel préemptives indépendantes avec la prise en compte du coût exact de la préemption. Pour cela nous avons choisi la stratégie d'ordonnancement multiprocesseur par partitionnement. Ce choix évite les coûts de migration élevés que l'on a avec la stratégie globale et par semi-partitionnement et nous permet d'utiliser l'analyse d'ordonnancement basée sur $\oplus$ présentée dans le chapitre précédent. Afin d'obtenir des résultats dans un temps raisonnable pour des applications industrielles réalistes, nous optons pour une heuristique plutôt qu'un algorithme exact très coûteux en terme de temps d'exécution. Ainsi nous proposons dans ce chapitre une heuristique d'ordonnancement qui alloue et qui ordonnance les tâches sur les différents processeurs tout en équilibrant l'utilisation des processeurs.

Ce chapitre est organisé comme suit : la section 4.2 présente le modèle de tâches et d'architecture multiprocesseur considérés, la section 4.3 présente l'heuristique d'ordonnancement multiprocesseur proposée et enfin la section 4.4 présente l'étude de performances de notre heuristique. Dans cette analyse de performances nous comparons notre heuristique à celles de BF et WF et à l'algorithme exact $B\&B$.

4.2 Modèle de tâches et d'architecture

Soit $\Gamma_n = \{\tau_1, \tau_2, \cdots, \tau_n\}$ un ensemble de tâches périodiques concrètes selon le modèle de tâches défini dans le chapitre précédent. Soit Π_m un ensemble de

m processeurs identiques. On suppose que les processeurs sont sans cache, ni "pipeline" ou d'architecture interne complexe. Notre objectif est d'ordonnancer l'ensemble des tâches Γ_n sur les m processeurs en prenant en compte le coût exact de la préemption et en équilibrant le facteur d'utilisation des processeurs.

4.3 Heuristique d'ordonnancement

Dans cette section on présente l'heuristique d'ordonnancement proposée pour résoudre le problème d'ordonnancement multiprocesseur de tâches temps réel indépendantes. Cette heuristique présentée dans l'algorithme 2 permet d'ordonnancer et d'allouer les tâches en même temps. L'allocation d'une tâche consiste à choisir le processeur sur lequel va s'exécuter cette tâche. Le critère d'allocation des tâches sur les différents processeurs est défini à l'aide de la fonction de coût présentée à la section suivante.

4.3.1 Fonction de coût

La fonction de coût permet de décider le choix du processeur sur lequel va être allouée une tâche donnée. Nous définissons cette fonction de coût sur chaque processeur comme étant égale au facteur d'utilisation du processeur considéré avec prise en compte du coût exact de la préemption. Le facteur d'utilisation du processeur avec coût de la préemption est donné par la relation 3.1 à la section 3.6. Puisque nous sommes dans le cas de l'ordonnancement multiprocesseur nous changeons la notation de la relation 3.1 pour préciser le processeur que l'on considère. Ainsi on note par $U^*_{p_j}$ le facteur d'utilisation du processeur p_j avec prise en compte du coût exact de la préemption. Par la suite on utilisera le terme facteur d'utilisation tout court pour désigner le facteur d'utilisation d'un processeur avec prise en compte du coût exact de la préemption.

Notre objectif est d'optimiser l'utilisation des processeurs afin d'éviter de surcharger un processeur alors que d'autres processeurs restent disponibles. Pour ce faire nous maximisons le facteur d'utilisation restant, appelé aussi capacité restante [LCGG10] de chaque processeur p_j. Cette capacité restante pour un processeur p_j est donnée par $1 - U^*_{p_j}$ puisque le facteur d'utilisation maximal de p_j est de 1. Cette maximisation $1 - U^*_{p_j}$ est équivalente à la minimisation de $U^*_{p_j}$. Ainsi dans notre heuristique on utilisera comme fonction de coût $F(p_j, \tau_i)$ donnée par la relation 4.1.

$$F(\tau_i, p_j) = U^*_{p_j} \tag{4.1}$$

avec τ_i allouée à p_j.

Lorsque τ_i peut être allouée à plusieurs processeurs, on choisira le processeur qui minimise $F(\tau_i, p_j)$. Ce choix d'allocation permet de rendre robuste l'exécution des tâches, c'est à dire augmente les chances des tâches de rester ordonnançables lors de leur exécution même si certaines tâches ont vu leurs durées d'exécution augmentées. Il permet aussi une meilleure répartition de l'exécution des tâches sur tous les processeurs, par rapport aux heuristiques de BF et WF, ce qui réduit le pire temps de réponse des tâches.

4.3.2 Principe de l'heuristique

L'heuristique d'ordonnancement et d'allocation qu'on propose est de type gloutonne. Une fois qu'une tâche est allouée à un processeur, cette allocation est définitive, il n'y a pas de retour en arrière. Cette heuristique ne prend en compte que les algorithmes d'ordonnancement à priorités fixes tels que RM, DM, etc., car elle utilise l'opération $\oplus$ pour faire l'analyse d'ordonnançabilité et cette opération ne s'applique qu'à des tâches à priorités fixes. Les tâches sont triées dans l'ordre décroissant de leurs priorités. Ce tri permet d'appliquer l'opération d'ordonnancement $\oplus$ toujours entre la tâche la plus prioritaire et la tâche la moins prioritaire comme l'exige le principe de $\oplus$. Tant qu'il reste une tâche non encore allouée dans Γ_n, on sélectionne la tâche la plus prioritaire puis on teste son ordonnançabilité sur chaque processeur en faisant l'analyse d'ordonnançabilité avec $\oplus$. Si la tâche sélectionnée est ordonnançable sur plusieurs processeurs, on alloue cette tâche au processeur p_j sur lequel elle est ordonnançable et qui a la plus petite valeur de $F(\tau_i, p_j)$. Si la tâche sélectionnée ne peut être ordonnancée par aucun processeur, donc le système de tâches Γ_n n'est pas ordonnançable selon notre heuristique.

L'algorithme 2 donne les différentes étapes de notre heuristique. Après avoir ordonnancé les tâches avec notre heuristique, nous pouvons produire une table d'ordonnancement pour chaque processeur qui contient les dates de début et de fin d'exécution des tâches allouées à ce processeur. La complexité de notre heuristique dépend de celle de l'opération $\oplus$. Pour un ensemble de n tâches à ordonnancer sur m processeurs, la complexité au pire cas de notre heuristique est évaluée à $O(n.m.l)$ avec $l = ppcm\{T_i : \tau_i \in \Gamma_n\}$.

4.4 Étude de performances

L'heuristique d'ordonnancement que nous avons présentée dans l'algorithme 2 a été implantée en OCaml afin de pouvoir la comparer aux heuristiques de "bin-packing" BF et WF et à l'algorithme exact $B\&B$ puisqu'on ne dispose pas de "benchmark" sur la prise en compte du coût exact de la préemption. Pour que cette

Algorithme 2 Heuristique d'ordonnancement multiprocesseur

1: Trier les tâches dans Γ_n suivant l'ordre décroissant de leurs priorités selon l'algorithme d'ordonnancement à priorités fixes considéré
2: Initialiser la variable $ordonnançable$ à $vrai$
3: **tant que** il reste des tâches non encore allouées dans Γ_n et $ordonnançable = vrai$ **faire**
4: Sélectionner dans Γ_n la tâche τ_i la plus prioritaire et non encore allouée à un processeur
5: *% boucle pour vérifier l'ordonnançabilité de la tâche τ_i sur chacun des processeurs avec $\oplus$.%*
6: **pour** $j = 1$ à m **faire**
7: **si** la tâche τ_i est ordonnançable sur p_j avec prise en compte du coût exact de la préemption selon l'analyse d'ordonnançabilité basée sur $\oplus$ **alors**
8: Calculer $F(\tau_i, p_j)$ pour τ_i allouée à p_j.
9: **fin si**
10: **fin pour**
11: **si** τ_i est ordonnançable au moins sur un processeur **alors**
12: Allouer τ_i au processeur p_j qui minimise $F(\tau_i, p_j)$
13: **sinon**
14: $ordonnançable = faux$
15: **fin si**
16: **fin tant que**

comparaison ait un sens nous avons utilisé la même condition d'ordonnançabilité basée sur $\oplus$ pour toutes les heuristiques et pour l'algorithme exact $B\&B$. Le choix de BF et WF parmi les heuristiques de "bin-packing" repose sur le fait qu'elles peut permettent d'introduire un critère d'optimisation. Avant de présenter les résultats obtenus, on présente d'abord les algorithmes des heuristiques BF et WF et de l'algorithme exact $B\&B$. En ce qui concerne les détails techniques , l'ordinateur utilisé pour les tests est un DELL muni d'un processeur $IntelCore^{TM}$ 2 Duo (2,66 GHz, mémoire RAM 4Go). Tous les algorithmes sont implémentés en OCaml.

4.4.1 Heuristiques BF et WF

Les heuristiques BF et WF utilisent l'analyse d'ordonnançabilité basée sur $\oplus$ pour prendre en compte le coût exact de la préemption. C'est pour cette raison que nous avons trié les tâches dans l'ordre décroissant de leurs priorités au lieu de l'ordre de leur facteur d'utilisation comme dans la plupart des travaux de l'état de l'art [DD86, OS95].

Heuristique BF

L'heuristique BF maximise le facteur d'utilisation des processeurs. Elle sélectionne la tâche la plus prioritaire et non encore allouée dans Γ_n qu'on note par τ_i puis tente d'allouer cette tâche au premier processeur dans l'ordre croissant des indices des processeurs déjà utilisés (processeurs auxquels on a déjà alloué au moins une tâche) et qui maximise $F(\tau_i, p_j)$. Si aucun processeur n'est encore utilisé ou que parmi les processeurs déjà utilisés aucun ne peut ordonnancer τ_i alors BF ajoute un nouveau processeur et alloue τ_i à ce nouveau processeur si le nombre de processeurs déjà utilisé est inférieur à m sinon Γ_n n'est pas ordonnançable. L'heuristique BF tente de charger au maximum un processeur avant d'en ajouter un nouveau. La fonction de coût et le parcours des processeurs sont les principales différences de BF et notre heuristique qui minimise $F(\tau_i, p_j)$ et qui parcourt tous les processeurs en partant du premier afin de trouver une allocation pour la tâche sélectionnée τ_i. L'algorithme 3 présente les étapes de l'allocation de BF.

Heuristique WF

La différence entre les heuristiques BF et WF est que la première maximise $F(\tau_i, p_j)$ alors que la deuxième heuristique la minimise. c'est-à-dire que dans le choix du processeur parmi les processeurs déjà utilisés, WF choisit celui qui minimise $F(\tau_i, p_j)$ pour ne pas trop surcharger un processeur alors que

Algorithme 3 Heuristique BF avec coût de la préemption

1: Trier les tâches dans Γ_n suivant l'ordre décroissant de leurs priorités selon l'algorithme d'ordonnancement à priorités fixes considéré
2: Initialiser la variable $nb\_proc\_utilisés = 1$
3: Initialiser la variable $ordonnançable$ à $vrai$
4: **tant que** (il reste des tâches non encore allouées dans Γ_n) et $(nb\_proc\_utilisés \leq m)$ et $(ordonnançable = vrai)$ **faire**
5: Sélectionner dans Γ_n la tâche τ_i la plus prioritaire et non encore allouée à un processeur
6: **pour** $j = 1$ à $nb\_proc\_utilisés$ **faire**
7: **si** la tâche τ_i est ordonnançable sur p_j avec prise en compte du coût exact de la préemption selon l'analyse d'ordonnançabilité basée sur $\oplus$ **alors**
8: Calculer $F(\tau_i, p_j)$, le facteur d'utilisation du processeur p_j en supposant que τ_i est allouée à p_j.
9: **fin si**
10: **fin pour**
11: **si** τ_i est ordonnançable au moins sur un des $nb\_proc\_utilisés$ premiers processeurs **alors**
12: Allouer τ_i au processeur p_j qui maximise $F(\tau_i, p_j)$
13: **sinon**
14: **si** $(nb\_proc\_utilisés < m)$ **alors**
15: Allouer τ_i au processeur p_{j+1}
16: $nb\_proc\_utilisés \leftarrow nb\_proc\_utilisés + 1$
17: **sinon**
18: $ordonnançable \leftarrow faux$
19: **fin si**
20: **fin si**
21: **fin tant que**

d'autre processeurs sont disponibles. WF sélectionne la tâche la plus prioritaire et non encore allouée dans Γ_n qu'on note par τ_i puis tente d'allouer τ_i au premier processeur dans l'ordre croissant des indices des processeurs déjà utilisés et qui minimise $F(\tau_i, p_j)$. Si aucun processeur n'est encore utilisé ou que parmi les processeurs déjà utilisés aucun ne peut ordonnancer τ_i alors WF ajoute un nouveau processeur et alloue τ_i à ce nouveau processeur si le nombre de processeur déjà utilisé est inférieur à m sinon Γ_n n'est pas ordonnançable. Le parcours des processeurs est la principale différence de WF et notre heuristique qui parcourt tous les processeurs en partant du premier afin de trouver le meilleur processeur pour allouer la tâche sélectionnée τ_i. L'algorithme 4 présente les étapes de l'allocation de WF.

Algorithme 4 Heuristique WF avec coût de la préemption

Trier les tâches dans Γ_n suivant l'ordre décroissant de leurs priorités selon l'algorithme d'ordonnancement à priorités fixe considéré
Initialiser la variable $nb\_proc\_utilisés = 1$
Initialiser la variable $ordonnançable$ à $vrai$
tant que (il reste des tâches non encore allouées dans Γ_n)et $(nb\_proc\_utilisés \leq m)$ et $(ordonnançable = vrai)$ **faire**
  Sélectionner dans Γ_n la tâche τ_i la plus prioritaire et non encore allouée à un processeur
  pour $j = 1$ à $nb\_proc\_utilisés$ **faire**
    si la tâche τ_i est ordonnançable sur p_j avec prise en compte du coût exact de la préemption selon l'analyse d'ordonnançabilité basée sur $\oplus$ **alors**
      Calculer $F(\tau_i, p_j)$, le facteur d'utilisation du processeur p_j en supposant que τ_i est allouée à p_j.
    fin si
  fin pour
  si τ_i est ordonnançable au moins sur un des $nb\_proc\_utilisés$ premiers processeurs **alors**
    Allouer τ_i au processeur p_j qui minimise $F(\tau_i, p_j)$
  sinon
    si $(nb\_proc\_utilisés < m)$ **alors**
      Allouer τ_i au processeur p_{j+1}
      $nb\_proc\_utilisés \leftarrow nb\_proc\_utilisés + 1$
    sinon
      $ordonnançable \leftarrow faux$
    fin si
  fin si
fin tant que

4.4.2 Algorithme exact $B\&B$

Contrairement aux heuristiques présentées précédemment, l'algorithme exact $B\&B$ explore toutes les solutions possibles pour déterminer la meilleure solution. Puisque notre objectif est d'équilibrer l'utilisation des processeurs pour ne pas surcharger un processeur alors que d'autres sont disponibles alors la fonction de coût est définie pour chaque solution S par $H(S)$ donnée par la relation 4.2.

$$H(S) = max\{U^*_{pj}\}_{p_j \in S} \tag{4.2}$$

Une solution S est composée d'un ensemble des processeurs pour lesquels un ensemble de tâches a été alloué à chaque processeur. La meilleure solution parmi toutes les solutions est celle qui minimise la fonction de coût H. L'algorithme 5 présente les étapes de l'allocation selon $B\&B$.

4.4.3 Résultats

Pour tous les algorithmes, nous avons utilisé le choix de priorités des tâches selon RM (la tâche la plus prioritaire est celle qui a la plus petite valeur de sa période). Nous allons comparer les temps d'exécution des algorithmes, leur taux de succès, les temps de réponse et les moyennes des facteurs d'utilisation restants sur les processeur. Pour la génération des tâches nous avons utilisé l'outil FORTAS [CG11] fondé sur l'algorithme de Bini [BB05].

4.4.3.1 Comparaison des temps d'exécution

Pour comparer le temps d'exécution entre les heuristiques et l'algorithme exact $B\&B$ nous avons dans un premier temps fixé le nombre de processeurs à 10 et généré 10 ensembles de tâches. Chaque ensemble de tâches est composé entre 100 et 1000 tâches générées aléatoirement avec l'outil FORTAS. On exécute successivement les 4 algorithmes (BF, WF, B&B et notre heuristique) sur tous les ensembles de tâches puis on note le temps d'exécution de chaque algorithme. Seuls les ensembles de tâches ordonnançables avec les 4 algorithmes ont été considérés. La figure 4.1 montre les résultats de ces tests. Dans un deuxième temps nous avons considéré un seul ensemble de tâches composé de 1000 tâches et nous avons fait varier le nombre de processeurs entre 3 et 20 processeurs. Les résultats de ces tests sont donnés par la figure 4.2.

Dans les deux cas on remarque que le temps d'exécution de l'algorithme exact $B\&B$ explose très vite par contre les heuristiques ont des temps d'exécution raisonnables. On note aussi que sur la figure 4.1 notre heuristique jusqu'à 1000 tâches donne des temps d'exécution proches de BF et WF, mais au delà, elle a des temps d'exécution plus importants. Ceci est dû au fait que pour allouer une

Algorithme 5 Algorithme exact de $B\&B$ avec coût de la préemption

Initialiser la variable $liste\_solutions$ à l'ensemble vide
Initialiser la variable $ordonnançable$ à $vrai$
tant que (il reste des tâches non encore allouées dans Γ_n) et ($ordonnançable = vrai$) **faire**
  Sélectionner dans Γ_n la tâche τ_i non encore allouée
  si $liste\_solutions = \{\emptyset\}$ **alors**
    Créer la solution S_1 composée d'un processeur avec τ_i allouée à ce processeur
    Ajouter S_1 à la liste des solutions $liste\_solutions$
  sinon
    $l\_sol \leftarrow \{\emptyset\}$
    pour $k = 1$ à $nb\_solutions$ (nombre de solution de $liste\_solutions$) **faire**
      $S_k \leftarrow$ solution k de $liste\_solutions$
      pour $j = 1$ à $nb\_proc\_s_k$ ($nb\_proc\_s_k$ est le nombre de processeurs dans S_k) **faire**
        si τ_i est ordonnançable sur p_j en utilisant $\oplus$ **alors**
          Créer la solution $S_{(k,j)}$ qui est la solution extraite de S_k en allouant la tâche τ_i au processeur p_j
          Ajouter $S_{(k,j)}$ à la liste des solutions l_sol
        fin si
      fin pour
      si $nb\_proc\_s_k < m$ **alors**
        Créer la solution $S_{(k,j+1)}$ qui est extraite de S_k en ajoutant un $j+1$ ème processeur et en allouant la tâche τ_i au processeur p_{j+1}
        Ajouter $S_{(k,j+1)}$ à la liste des solutions l_sol
      fin si
    fin pour
    si $l\_sol = \emptyset$ **alors**
      $ordonnançable = faux$
    sinon
      $liste\_solutions \leftarrow l\_sol$
    fin si
  fin si
fin tant que
si $ordonnançable = vrai$ **alors**
  $meilleure\_solution \leftarrow$ première solution dans $liste\_solutions$
  pour $k = 2$ à $nb\_solutions$ **faire**
    Calculer $H(S_k)$
    si $H(S_k) < H(meilleure\_solution)$ **alors**
      $meilleure\_solution \leftarrow S_k$
    fin si
  fin pour
  Γ_n est ordonnançable et $meilleure\_solution$ est la meilleure solution
sinon
  L'ensemble Γ_n n'est pas ordonnançable
fin si

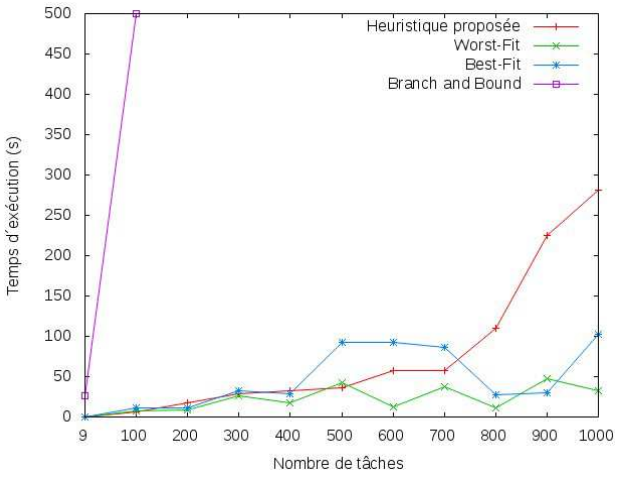

FIGURE 4.1 – Comparaison des temps d'exécution avec variation du nombre de tâches

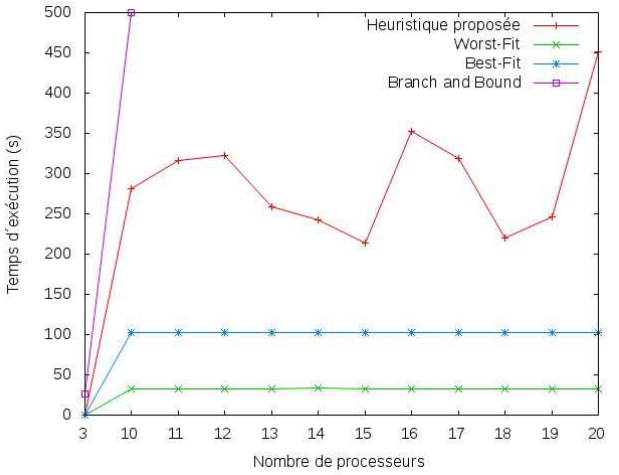

FIGURE 4.2 – Comparaison des temps d'exécution avec variation du nombre de processeurs

tâche, notre heuristique parcourt tous les processeurs afin de trouver une allocation à cette tâche qui donne une meilleure optimisation de l'utilisation des processeurs alors que les heuristiques BF et WF optimise uniquement sur les processeurs déjà utilisés.

Sur la figure 4.2, on remarque que lorsque le nombre de processeurs varie, les temps d'exécution des heuristiques BF et WF restent constants, ceci est dû

au fait que ces heuristiques n'utilisent que le nombre de processeurs minimum nécessaire pour ordonnancer un ensembles de tâches. Elles n'optimisent pas l'ordonnancement sur l'ensemble des processeurs, ce qui permet de les exécuter plus rapidement. Sur cette même figure 4.2, on constate que le temps d'exécution de notre heuristique ne croît pas de façon monotone avec le nombre de processeurs, ceci parce qu'elle répartit les tâches sur tous les processeurs disponibles et cette répartition entraîne sur certains processeurs une diminution de la valeur du $ppcm$ des tâches sur ces processeurs, ce qui réduit le temps d'exécution de $\oplus$. Ceci diminue le temps d'exécution de notre heuristique.

4.4.3.2 Comparaison des taux de succès

Dans cette partie nous comparons le taux de succès des différents algorithmes. l'algorithme exact $B\&B$ nous avons limité les tests à un petit nombre d'ensembles de tâches. Ainsi nous avons fixé le nombre de processeurs à 2 puis nous avons généré 6 ensembles chacun composé de 6 ensembles de tâches. Chaque ensemble de tâches est composé de 10 tâches. On exécute les 4 algorithmes sur chaque ensemble de tâches et on calcule le taux de succès de chaque algorithme sur chaque ensemble d'ensembles de tâches selon l'équation 3.2. Les résultats de ces tests sont donnés par la figure 4.3.

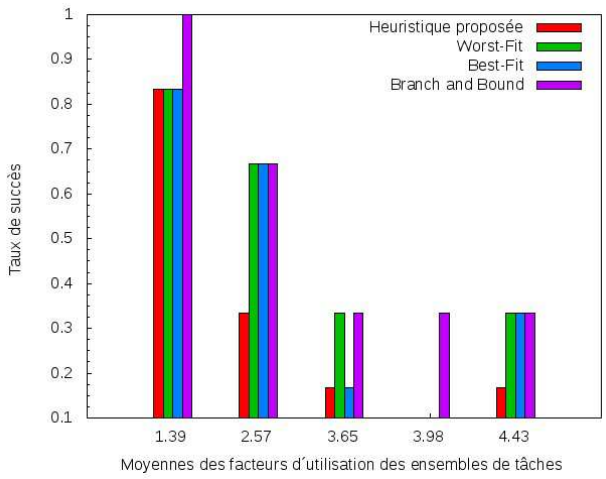

FIGURE 4.3 – Comparaison des taux de succès

On remarque, comme c'était prévu, que l'algorithme exact $B\&B$ donne des taux de succès meilleurs que les heuristiques. De plus on remarque que les heuristiques BF et WF donnent aussi des taux de succès meilleurs que notre heuristique. Cette contre performance en terme de taux de succès de notre heuristique

par rapport aux heuristiques BF et WF est compensée par le fait que notre heuristique donne de meilleurs temps réponse et optimise la capacité restante sur les différent processeurs.

4.4.3.3 Comparaison des temps de réponse de l'ordonnancement des tâches

Dans cette partie nous comparons les temps de réponse de l'ordonnancement de l'allocation trouvés par les différentes heuristiques (BF, WF et notre heuristique). Nous avons limité ces tests aux heuristiques à cause de la complexité de l'algorithme exact $B\&B$. On considère 10 ensembles de tâches. Chaque ensemble de tâches contient entre 100 et 1000 tâches générées aléatoirement et on a fixé le nombre de processeurs à 10. Pour chaque algorithme on détermine l'allocation et on calcule le pire temps de réponse de l'ordonnancement des tâches, c'est-à-dire le pire temps total pour exécuter l'ensemble des tâches sur les différents processeurs sur lesquels elles ont été allouées. Les résultats de ces tests sont donnés par la figure 4.4.

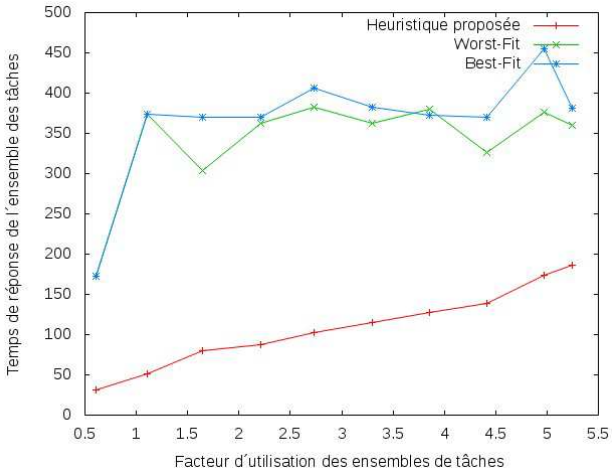

FIGURE 4.4 – Comparaison des temps de réponse de l'ordonnancement des tâches

On remarque que notre heuristique trouve des allocations qui donnent des temps de réponse meilleurs que ceux trouvés par les heuristiques BF et WF, ceci parce que notre heuristique répartit l'exécution des tâches sur l'ensemble des processeurs ce qui permet une exécution rapide des tâches.

4.4.3.4 Comparaison des moyennes des facteurs d'utilisation restants

Dans cette partie nous comparons les moyennes des facteurs d'utilisation restants ou capacité restantes sur les processeurs entre les allocations trouvées par les heuristiques. Ces moyennes permettent de déterminer le niveau d'utilisation des processeurs. On considère les mêmes données de test qu'à la section 4.4.3.3. Après exécution de chaque heuristique sur chaque ensemble de tâches, on calcule la moyenne des $1 - U_{p_j}$ de l'allocation trouvée. Les résultats de ces tests sont donnés sur la figure 4.5.

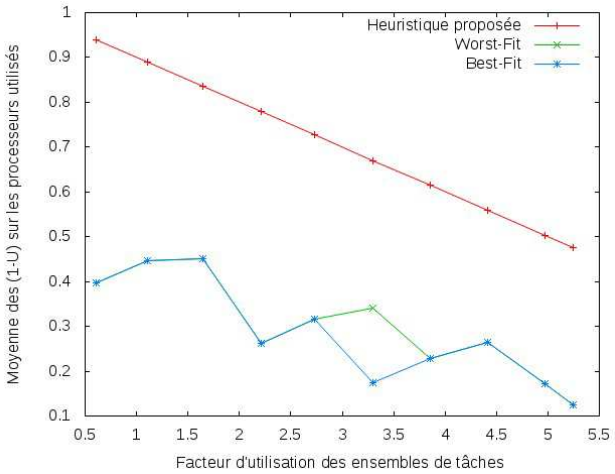

FIGURE 4.5 – Comparaison des moyennes des facteurs d'utilisations restants sur les processeurs

Sur la figure 4.5 on remarque que, les allocations trouvées par notre heuristique donnent de plus grande moyennes de $1 - U_{p_j}$. Ceci montre une meilleure répartition de l'utilisation des processeurs. En effet les heuristique BF et WF n'utilisent que le nombre de processeurs nécessaires pour allouer les tâches. C'est ce qui fait qu'elle peuvent utiliser un plus petit nombre de processeurs par rapport au processeurs disponibles. Ceci a pour conséquence de mettre le maximum de tâches sur ces processeurs.

4.5 Conclusion

Dans ce chapitre nous avons traité le problème de l'ordonnancement multiprocesseur de tâches temps réel indépendantes avec la prise en compte du coût exact de la préemption. Nous avons proposé une heuristique d'ordonnancement

multiprocesseur qui permet d'équilibrer l'utilisation des différents processeurs. Cette contribution fait partie des premiers travaux réalisés dans l'ordonnancement temps réel multiprocesseur avec prise en compte du coût exact du coût exact de la préemption. Dans la partie qui suit nous allons étudier l'ordonnancement multiprocesseur des tâches temps réel avec des dépendances de données, toujours avec prise en compte du coût exact de la préemption.

Troisième partie

Ordonnancement temps réel multiprocesseur de tâches dépendantes avec prise en compte du coût exact de la préemption

Chapitre 5

Ordonnancement monoprocesseur de tâches dépendantes

Dans ce chapitre on étudie l'ordonnancement de tâches temps réel avec des contraintes de dépendances de données et avec prise en compte du coût exact de la préemption. Nous allons dans ce chapitre étudier ce problème d'ordonnancement en monoprocesseur ensuite nous allons l'étendre en multiprocesseur en utilisant la stratégie par partitionnement. On propose dans ce chapitre une analyse d'ordonnançabilité des tâches dépendantes qui prend en compte le coût exact lié à la préemption. Contrairement à l'analyse d'ordonnancement basée sur l'opération $\oplus$ (voir chapitre 3), qui ne s'applique qu'à des tâches à priorités fixes aux tâches, notre analyse d'ordonnançabilité est applicable aussi bien à des tâches à priorités fixes qu'à des tâches à priorités dynamiques. Ainsi elle permet d'ordonnancer des tâches dépendantes qui entraînent des inversions de priorités dues aux partages de ressources (dans notre cas partage des mémoires de données échangées). Notre analyse d'ordonnançabilité permet aussi de gérer les transferts de données entre tâches.

Ce chapitre est organisé comme suit : la section 5.1 présente le modèle de tâches et les notations utilisées, la section 5.2 présente l'intervalle d'étude utilisé pour faire l'analyse d'ordonnançabilité, la section 5.3 présente le mécanisme de transfert de données entre les tâches, la section 5.4 présente la synchronisation de l'accès aux mémoires tampons, la section 5.5 présente l'analyse d'ordonnançabilité proposée et enfin la section 5.6 présente la viabilité de cette analyse d'ordonnançabilité.

5.1 Modèle de tâches et notations

5.1.1 Modèle de tâches

On considère un ensemble de n tâches périodiques concrètes $\Gamma_n = \{\tau_1, \tau_2, \cdots, \tau_n\}$ avec des dépendances de données représentées par un graphe acyclique de dépendances noté par G où les sommets représentent les tâches et l'ensemble des arcs la relation de dépendances de données. Une tâche $\tau_i = (r_i^1, C_i, T_i, D_i)$ est caractérisée par une date de première activation r_i^1, une pire durée d'exécution C_i n'incluant pas les coûts de préemptions, une période T_i et une échéance relative D_i (durée maximale pour l'exécution de τ_i depuis sa dernière activation). On suppose que le coût lié à une préemption est une constante connue notée α.

Pour éviter les pertes de données on suppose que les tâches qui ont des dépendances de données ont des périodes multiples [Ker09]. Un exemple de graphe tâches dépendantes noté G_5 composé de 5 tâches est présenté à la figure 5.1.

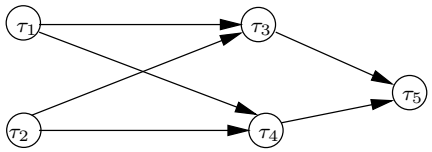

FIGURE 5.1 – Graphe G_5 de dépendances de données

L'ensemble des tâches Γ_n est ordonnancé hors ligne selon un choix de priorité basé sur un algorithme noté $Algo$ (les priorités peuvent être fixes ou dynamiques). Nous supposons aussi que chaque tâche τ_i produit des données qu'elle sauvegarde dans une mémoire tampon ("buffer" en anglais) notée buf_i. La figure 5.2 illustre les mémoires de données des tâches du graphe G_5.

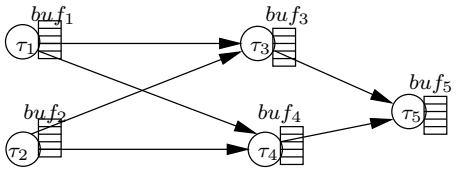

FIGURE 5.2 – Illustration des mémoires de données

Dans la figure 5.2, la mémoire buf_1 est partagée entre la tâche τ_1 qui y écrit les données qu'elle produit, les tâches τ_3 et τ_4 qui y lisent les données produites par τ_1. Lorsque la tâche τ_3, consommant les données produites par τ_1 sauvegardées dans buf_1, est préemptée par τ_1 au moment où elle commence à lire dans buf_1

Notation	Description
$H_n = ppcm\{T_i\}_{1 \leq i \leq n}$	hyperpériode des tâches dans Γ_n,
$r_{min} = min\{r_i^1\}_{1 \leq i \leq n}$	La plus petite date de première activation parmi les tâches dans Γ_n
$r_{max} = max\{r_i^1\}_{1 \leq i \leq n}$	La plus grande date de première activation parmi les tâches dans Γ_n
$pred(\tau_i)$	L'ensemble des prédécesseurs de la tâche τ_i
$succ(\tau_i)$	L'ensemble des successeurs de la tâcheτ_i
G	L'ensemble des dates d'activation et de fin d'exécution sur l'intervalle d'étude I_n
$r^-(t)$	La dernière date d'appel de l'ordonnanceur avant t
$r^+(t)$	La prochaine date d'appel de l'ordonnanceur après t
$\Gamma_r(t)$	L'ensemble des tâches prêtes à la date t
$\phi_i(t)$	La tâche sélectionnée à l'exécution à la date t
$c_i(t)$	La durée restant à exécuter par la tâche τ_i à la date t
$d_i(t)$	L'échéance relative à la date t de la tâche τ_i
p_i	La priorité de la tâche τ_i selon l'algorithme de choix de priorités $Algo$

TABLE 5.1 – Résumé des notations

alors les données dans buf_1 peuvent être dans un état inconsistant car la tâche τ_1 peut modifier ces données en même temps que τ_3 les lit. Donc τ_1 et τ_3 ne doivent pas accéder à la mémoire buf_1 simultanément. Par contre les tâches τ_3 et τ_4 peuvent lire à la mémoire buf_1 simultanément puisqu'elles ne modifient pas les données dans buf_1. En règle générale, deux tâches ne peuvent pas écrire et lire simultanément dans une mémoire partagée, on dit qu'elles ont des contraintes d'exclusions mutuelles dans l'accès à cette mémoire. En revanche, elle peuvent lire la même donnée dans la même mémoire.

Nous supposons aussi que le processeur que nous considérons n'a pas de cache, ni pipeline ou d'architecture interne complexe. On retrouve ces hypothèses dans le contexte des applications critiques. Finalement on suppose aussi que le coût de transfert de données entre les tâches est négligeable puisque nous considérons un seul processeur et qu'il correspond donc à une écriture et une lecture dans une mémoire.

5.1.2 Notations

Nous introduisons au tableau 5.1 ci-dessous quelques notations utilisées par la suite.

5.2 Intervalle d'étude de l'ordonnancement

Comme dans le cas des tâches indépendantes nous utilisons l'approche par simulation pour faire l'analyse d'ordonnançabilité des tâches dépendantes. La simulation de l'ordonnancement des tâches permet de compter le nombre exact de préemptions et d'ajouter leur coût au fur et à mesure qu'on ordonnance les tâches. Dans [CGG04], Choquet et Grolleau ont proposé un intervalle d'étude pour un ensemble de n tâches dépendantes périodiques concrètes donné par :

$$I_n = [r_{min}, t_c + H_n] \tag{5.1}$$

où t_c est la date à partir de laquelle l'ordonnancement se répète et est calculée itérativement par un algorithme donné dans cet article. D'après cet algorithme qui calcule t_c, on a $tc \leq r_{max} + H_n$. Donc pour simplifier le calcul de I_n, nous utiliserons l'intervalle d'étude donné par la relation 5.1 avec $tc = r_{max} + H_n$. Mais il faut vérifier que l'ordonnancement à la date t_c est égal à celui qu'on obtient à la date $t_c + H_n$, car dans [CGG04], on considère que le WCET des tâches reste fixe sur une instance d'une tâche. L'intervalle d'étude que nous utiliserons dans notre analyse d'ordonnançabilité est alors donné par la relation 5.2.

$$I_n = [r_{min}, r_{max} + 2 * H_n] \tag{5.2}$$

5.3 Mécanisme de transfert de données

Soient deux tâches dépendantes τ_i et τ_j avec une contrainte de dépendances de données telle que τ_i est la tâche productrice de données et τ_j est la tâche consommatrice de données. On suppose que toutes les données produites par τ_i seront consommées par τ_j sans aucune perte de données. Si T_i est la période de τ_i et T_j celle de τ_j alors nous avons les cas suivants :

- $T_i < T_j$ alors la tâche τ_i s'exécute $\frac{T_j}{T_i}$ fois et produit $\frac{T_j}{T_i}$ données avant une seule exécution de τ_j qui consomme ces données durant son exécution,
- $T_i > T_j$ alors la tâche τ_i s'exécute une seule fois et produit une donnée avant $\frac{T_i}{T_j}$ exécutions de τ_j qui consomme durant chacune de ses exécutions la donnée produite par τ_i,
- $T_i = T_j$ alors la tâche productrice τ_i s'exécute une fois et produit une donnée avant une exécution τ_j qui consomme cette donnée durant cette exécution. Ce cas correspond à une contrainte de précédence simple.

Puisque les tâches dépendantes ont des périodes multiples, ces trois cas peuvent être résumés en un seul cas donné par : τ_i s'exécute $k_{ij} = \lceil \frac{T_j}{T_i} \rceil$ fois et produit k_{ij}

donnée(s) pour τ_j puis τ_j s'exécute $k_{ji} = \lceil \frac{T_i}{T_j} \rceil$ fois et durant chaque exécution, τ_j consomme la ou les k_{ij} données produites par τ_i. $\lceil x \rceil$ est la partie entière supérieure de x.

- $k_{ij} = \lceil \frac{T_j}{T_i} \rceil$ est le nombre de données que doit produire la tâche τ_i pour la tâche τ_j. Ce nombre correspond aussi au nombre de fois que τ_i doit s'exécuter avant que τ_j ne commence son exécution. τ_i doit attendre de recevoir la ou les k_{ij} donnée(s) de τ_i avant de commencer son exécution.
- $k_{ji} = \lceil \frac{T_i}{T_j} \rceil$ est le nombre de fois que τ_j doit consommer la ou les k_{ij} donnée(s) produite(s) par τ_i. Ce nombre correspond aussi au nombre de fois que τ_j doit s'exécuter avant que τ_i ne recommence à produire d'autres données. Pour éviter les pertes de données, τ_i doit attendre que τ_j consomme toutes les données qu'elle a déjà produites avant qu'elle n'en produise d'autres.

Les tâches τ_i et τ_j étant périodiques, leurs exécutions se répètent indéfiniment de même que la dépendance de données entre ces deux tâches. On appelle motif d'exécution de (τ_i, τ_j), une exécution de τ_i et τ_j telle que le mécanisme de transfert de données est respecté. C'est-à-dire une exécution composée de k_{ij} instances de τ_i et de k_{ji} instances de τ_j.

Un exemple d'ordonnancement sans pertes de données est présenté à la figure 5.4. Dans cette figure nous avons $\Gamma_3 = \{\tau_1, \tau_2, \tau_3\}$ un ensemble composé de 3 tâches caractérisées par : $\tau_1 = (0, 2, 6, 6)$, $\tau_2 = (1, 1, 3, 3)$ et $\tau_3 = (2, 1, 6, 6)$. Le graphe G_3 donné à la figure 5.3, représente le graphe des dépendances de données des tâches dans Γ_3.

FIGURE 5.3 – Graphe G_3

La tâche τ_1 s'exécute $k_{12} = \lceil \frac{T_2}{T_1} \rceil = \lceil \frac{3}{6} \rceil = 1$ et produit durant cette exécution $k_{12} = 1$ donnée avant que la tâche τ_2 ne s'exécute $k_{21} = \lceil \frac{T_1}{T_2} \rceil = \lceil \frac{6}{3} \rceil = 2$ fois et consomme durant chacune des ses exécutions la seule donnée produite par τ_1. La tâche τ_2 s'exécute $k_{23} = \lceil \frac{T_3}{T_2} \rceil = \lceil \frac{6}{3} \rceil = 2$ fois et produit $k_{23} = 2$ données avant que τ_3 ne s'exécute $k_{32} = \lceil \frac{T_2}{T_3} \rceil = \lceil \frac{3}{6} \rceil = 1$ et consomme durant son exécution les k_{23} données produites pour la tâche τ_2.

La figure 5.5 donne des exemples de motifs d'exécution sans perte de données de l'ensemble Γ_3. Dans cette figure, τ_2^2 représente la deuxième instance de la tâche τ_2 et sa deuxième instance dans le premier motif d'exécution de Γ_3, τ_2^3 est la troisième instance de τ_2 et sa première instance dans le deuxième motif d'exécution de Γ_3.

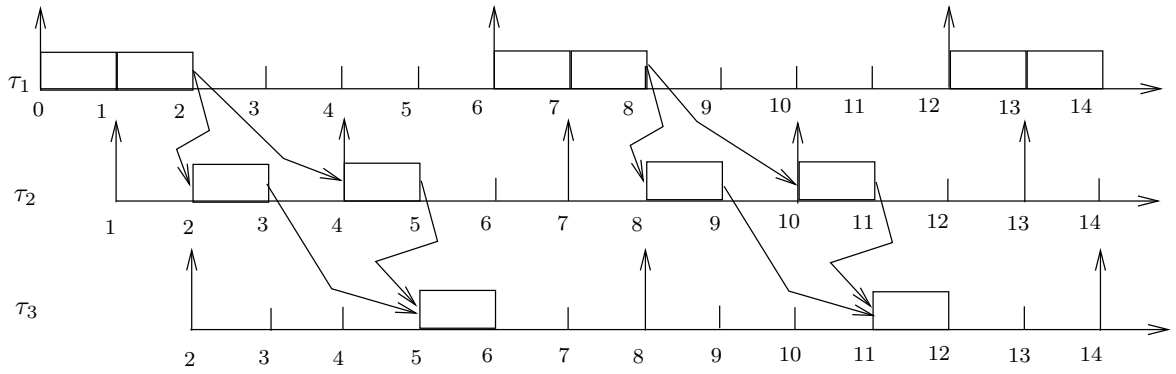

FIGURE 5.4 – Illustration du transfert de données des tâches dans G_3

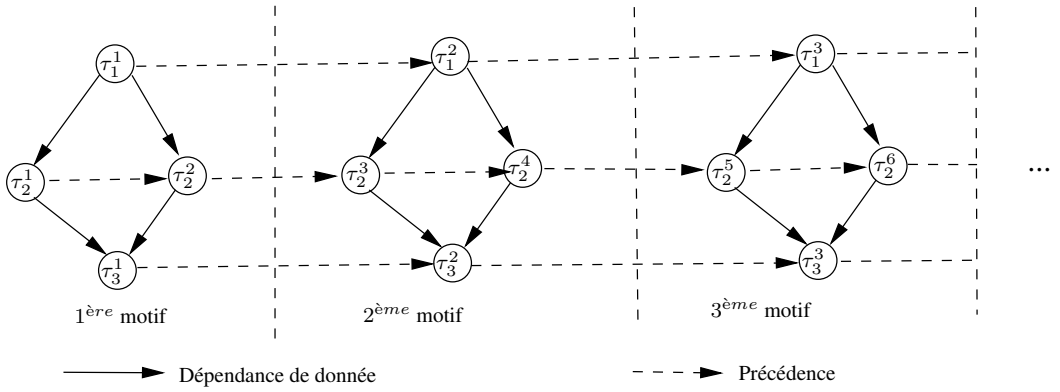

FIGURE 5.5 – Motifs d'exécution des tâches de Γ_3

Définition 5.1 *Le makespan est défini pour chaque motif d'exécution i d'un ensemble de tâches Γ_n par le temps qui s'écoule entre la date de début d'exécution de la première instance de la première tâche d'entrée exécutée parmi les tâches sans prédécesseur dans le motif i et la date de fin d'exécution de la dernière instance de la dernière tâche de sortie exécutée parmi les tâches sans successeur dans le motif i .*

Définition 5.2 *Le makesepan peut varier d'un motif à un autre à cause de la variation des temps de réponse des tâches. On définit alors le pire $makespan$ d'un ensemble de tâches Γ_n par le maximum des makespans des motifs d'exécution contenu dans l'intervalle d'étude de Γ_n.*

Par exemple dans la figure 5.4, le premier motif de Γ_3 débute à $t = 0$ et se termine à la date $t = 6$. Son deuxième motif d'exécution commence à la date

$t = 6$ et se termine à $t = 12$. Le makespan de Γ_3 est le même dans les deux motifs d'exécution de l'intervalle d'étude. Donc le pire makespan de Γ_3 est de 6

5.4 Synchronisation de l'accès aux mémoires de données

Le mécanisme de transfert de données vu dans la section précédente permet aussi d'éviter les inter-blocages entre les tâches car la tâche τ_j ne pourra commencer son exécution que lorsqu'elle dispose de toutes les k_{ij} données produite par τ_i et ceci durant toute la durée de son exécution, ce qui veut dire que τ_j ne peut jamais être arrêtée à cause d'attente de données de τ_i donc d'indisponibilité de la mémoire buf_i. Par contre ce mécanisme ne permet pas d'éviter les inversions de priorités qui rallongent le makespan de l'exécution des tâches. Pour éviter cela, on utilise le principe de l'héritage de priorités qu'on retrouve dans les protocoles de synchronisation PIP et PCP présenté respectivement aux sections 2.4.3.1 et 2.4.3.2 du chapitre 2. Lorsqu'une tâche τ_j commence son exécution, elle hérite de la plus grande priorité de ses tâches prédécesseurs qui lui fournissent des données. Une fois que la tâche τ_j a fini son exécution elle retrouve sa priorité initiale. Ceci permet d'éviter que τ_j soit préemptée par une tâche de priorité intermédiaire par rapport à ses prédécesseurs, c'est-à-dire les tâches plus prioritaires que τ_i et moins prioritaires que les prédécesseurs de τ_j. Ainsi on évite le rallongement du makespan des tâches.

On note par $p_i(t)$, la priorité de τ_i à la date t. $p_i(t)$ est déterminée par l'équation 5.3.

$$p_i(t) = \begin{cases} p_i \text{ si } ((t = r_{min}) \vee (\phi(r^-(t)) = \tau_i) \wedge ((r^-(t) + c_i(r^-(t))) = t)) \text{ sinon} \\ max\{p_j\}_{\tau_j \in pred(\tau_i)} \quad \text{si } \phi(r^-(t)) = \tau_i \text{ sinon} \\ p_i(r^-(t)) \end{cases} \tag{5.3}$$

$\phi(r^-(t))$ est la tâche qui a été sélectionnée pour s'exécuter à la dernière date d'appel de l'ordonnanceur notée $r^-(t)$ et $c_i(r^-(t))$ est la durée restant à exécuter par cette tâche à cette date.

La figure 5.6 donne une illustration de cet héritage de priorités.

Dans cette figure nous avons trois tâches τ_1, τ_2 et τ_3 telles que τ_1 est plus prioritaire que τ_3 qui est plus prioritaire que τ_2. Les tâches τ_2 et τ_3 consomment les données produites par τ_1. Dans un premier temps, nous avons ordonnancé les

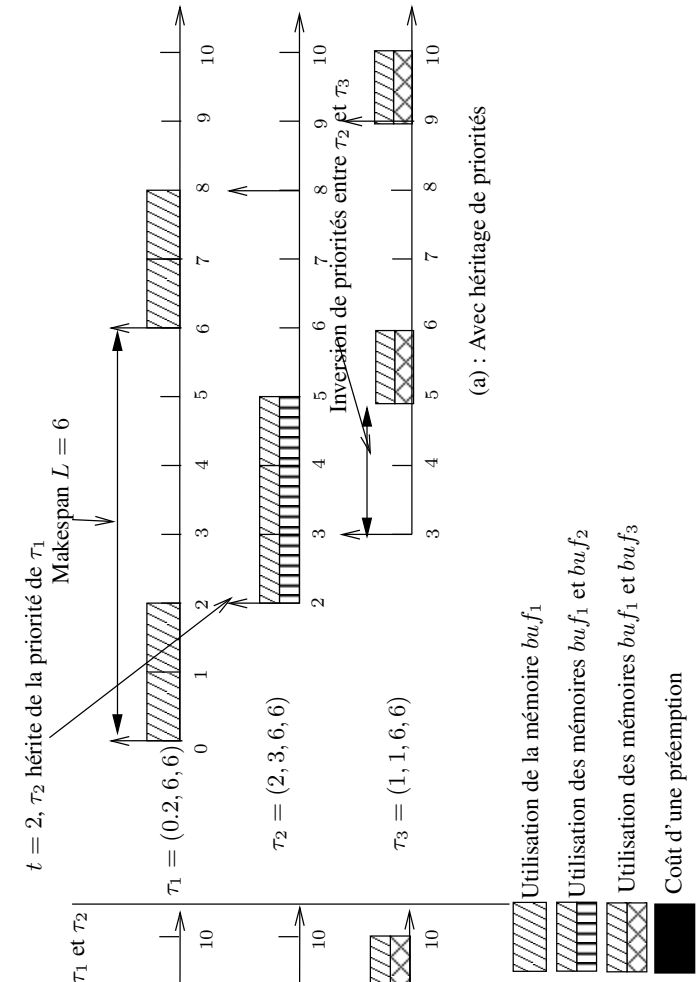

FIGURE 5.6 – Illustration du principe d'héritage de priorités

tâches sans héritage de priorités correspondant à la figure 5.6.(a) puis dans un second temps nous les avons ordonnancé avec héritage de priorités correspondant à la figure 5.6.(b). Dans le cas sans héritage de priorités, à la date $t = 3$ lorsque τ_3 est activée elle préempte la tâche τ_2 qui avait commencé son exécution à $t = 2$ ce qui rallonge le temps de réponse de la tâche τ_2 à cause du coût de la préemption avec un makespan d'exécution des tâches égale à $L = 7$. Cette préemption entraîne aussi le blocage d'une unité de temps de la tâche τ_1 par τ_2 car à $t = 6$, τ_1 est activée mais elle attend la libération de buf_1 utilisée par τ_2 pour s'exécuter. Par contre dans le cas avec héritage de priorités, τ_2 hérite à $t = 3$ de la priorité de τ_1 pour toute la durée de son exécution ce qui lui permet de ne pas être préemptée par τ_3. Même si cet héritage de priorités a entraîné une inversion de priorités entre τ_2 et τ_3, il permet d'éviter un blocage de τ_1 par τ_2 ou par τ_2 et donne une meilleur makespan des tâches, $L = 6$, par rapport à l'ordonnancement sans héritage.

5.5 Analyse d'ordonnançabilité hors ligne

On considère Γ_n, un ensemble de n tâches périodiques avec des contraintes de dépendances de données. Γ_n est défini selon le modèle de tâches présenté dans la section 5.1.1. Nous allons faire l'analyse d'ordonnançabilité hors ligne de Γ_n sur un seul processeur avec la prise en compte du coût exact de la préemption. Pour cela, en partant de la plus petite des dates de première activation qui est r_{min}, on appelle notre ordonnanceur hors ligne à cette date qui sélectionne la tâche à exécuter. Puis en utilisant une condition d'ordonnançabilité nécessaire et suffisante on vérifie si cette tâche sélectionnée à r_{min} ainsi que toutes les autres tâches, sont ordonnançables à cette date. Cette condition d'ordonnançabilité utilise la durée restant à exécuter pour chaque tâche et l'échéance de chaque tâche relative à la date r_{min}. La durée restant à exécuter prend en compte le coût exact de la préemption. Si toutes les tâches sont ordonnançables à la date r_{min}, on détermine la prochaine date d'appel de notre ordonnanceur. Sinon l'ensemble Γ_n n'est pas ordonnançable. Au prochain appel de l'ordonnanceur, on répète le même traitement jusqu'à la fin de l'intervalle d'étude I_n présenté à la section 5.2. Lorsqu'on arrive à la fin de l'intervalle I_n, si Γ_n est ordonnançable, le résultat de l'ordonnancement de Γ_n est donné par la table d'ordonnancement T. Cette table donne l'ordre total d'exécution des tâches sous la forme de leur date de début et de fin d'exécution dans l'intervalle d'étude I_n.

Dans la suite de cette section, on détaille la sélection de la tâche à exécuter, le calcul de la durée restant à exécuter de chaque tâche, le calcul de l'échéance de chaque tâche relative à la date d'appel de l'ordonnanceur, la condition d'ordonnançabilité d'une tâche et enfin le calcul de la prochaine date d'appel de l'ordonnanceur. Ces calculs sont faits chaque fois que l'ordonnanceur est appelé. L'ana-

lyse d'ordonnançabilité est donnée dans l'algorithme 5.5.6.

5.5.1 Sélection de la tâche à exécuter

À chaque fois que l'ordonnanceur est appelé, il fait la mise à jour de la liste des tâches prêtes afin de sélectionner la tâche à exécuter. Cette liste contient toutes les tâches activées et qui ont reçu toutes les données nécessaires à leurs exécution.

Lemme 5.1 *Le nombre de données produites par une tâche τ_p sur l'intervalle de temps $[r_{min}, t]$ est donnée par :*

$$b_p(t) = \begin{cases} 0 & \textit{si } (t < r_p^1 + C_p) \textit{ sinon} \\ b_p(r^-(t)) + 1 & \textit{si } ((\phi(r^-(t)) = \tau_p) \wedge \\ & \quad (r^-(t) + c_p(r^-(t)) \leq t)) \textit{ sinon} \\ b_p(r^-(t)) & \end{cases} \tag{5.4}$$

$r^-(t)$, $\phi(r^-(t))$ et $c_p(r^-(t))$ représentent respectivement la dernière date d'appel de l'ordonnanceur, la tâche sélectionnée pour s'exécuter à la date $r^-(t)$ et la durée d'exécution que doit sexécuter τ_p à la date $r^-(t)$ pour terminer son exécution.

Preuve
Le nombre de données produites par une tâche τ_p à t est égale à 0 si τ_p n'est pas encore activée à t ou qu'à cette date t, τ_p est en cours d'exécution de sa toute première instance. Si elle à été sélectionnée pour s'exécuter à la date $r^-(t)$ ($\phi(r^-(t)) = \tau_p$), correspondant à la dernière date où l'ordonnanceur a été appelé, et qu'elle finit son exécution avant ou à t ($r^-(t) + c_p(r^-(t)) \leq t$) donc une nouvelle donnée est produite par τ_p. Dans ce cas on incrémente son nombre de données produites. Sinon, aucune donnée n'est produite à t donc le nombre de donnée de τ_i ne change pas par rapport à la date $r^-(t)$. ∎

Lemme 5.2 *Soit (τ_p, τ_i) un couple de tâches avec une contrainte de dépendance de données telle que τ_p est la tâche productrice de données et τ_i la tâche consommatrice de données.*

Le nombre de données produites dans l'intervalle de temps $[r_{min}, t]$ par la tâche τ_p et qui ne sont pas encore consommées par la tâche τ_i est donné par :

$$d_{pi}(t) = b_p(t) * k_{ip} - b_i(t) * k_{pi} \tag{5.5}$$

avec $k_{pi} = \lceil \frac{T_i}{T_p} \rceil$ et $k_{ip} = \lceil \frac{T_p}{T_i} \rceil$.

Preuve
Lorsque la tâche τ_p produit 1 donnée, cette donnée va être consommée k_{ip} fois par τ_i alors le nombre de données de τ_p disponibles pour la tâche τ_i est de k_{ip} données au lieu de 1. En effet la donnée produite par la tâche τ_p représente k_{ip} données pour la tâche τ_i. Ce qui veut dire que si τ_p a produit $b_p(t)$ données dans l'intervalle de temps $[r_{min}, t]$ alors le nombre total de données de τ_p disponibles pour τ_i est de $b_p(t) * k_{ip}$ données. De l'autre coté, 1 donnée produite par τ_i entraîne la consommation de k_{pi} données de τ_p. Donc si τ_i a produit $b_i(t)$ dans l'intervalle de temps $[r_{min}, t]$ alors τ_i a consommées $b_i(t) * k_{pi}$ données de τ_p. Pour obtenir $d_{pi}(t)$, il suffit de faire la différence entre le nombre de données de τ_p disponibles pour τ_i et le nombre de données que τ_i a consommé parmi ces dernières. D'où la relation 5.5. ∎

À la date t (date d'appel de l'ordonnanceur), pour qu'une tâche τ_i activée ($r_i^1 \leq t$) puisse être à l'état prêt, donc dans la liste des tâches prêtes, elle doit vérifier certaines conditions par rapport à l'ensemble de ses tâches prédécesseurs $pred(\tau_i)$ qui lui fournissent les données qu'elle consomme et par rapport à ses tâches successeurs $succ(\tau_i)$ qu'elle fournit des données. Par rapport à chacun des ses prédécesseurs $\tau_p \in pred(\tau_i)$, τ_i doit avoir reçu au plus tard à t toutes les $k_{pi} = \lceil \frac{T_i}{T_p} \rceil$ données produites par τ_p mais aussi toutes les données $k_{is} = \lceil \frac{T_s}{T_i} \rceil$, qu'elle a déjà produites pour chacun des ses successeurs $\tau_s \in succ(\tau_i)$, doivent être consommées toutes par τ_s au plus tard à t.

Théorème 5.1 *Une tâche τ_i est prête (à l'état prêt d'exécution voir section 1.7.2) à l'exécution à la date t si et seulement si :*

1. $t \geq r_i^1$,
2. $(r^-(t) + c_i(r^-(t)) > t) \vee ((c_i(r^-(t)) > 0) \wedge (\phi(r^-(t)) \neq \tau_i)) \vee ((t - r_i^1) mod T_i = 0)$,
3. $\forall \tau_p \in pred(\tau_i), d_{pi}(t) \geq k_{pi}$,
4. $\forall \tau_s \in succ(\tau_i), d_{is}(t) < k_{is}$.

Preuve
Supposons qu'une de ces quatre conditions n'est pas vérifiée. Si c'est la condition 1) qui est fausse c'est-à-dire que $t < r_i^1$ alors la tâche τ_i n'est pas encore activée donc ne peut pas être prête à l'exécution. Sinon, si c'est la condition 2) qui est fausse c'est à dire que $(r^-(t) + c_i(r^-(t)) \leq t) \wedge (c_i(r^-(t)) \leq 0) \vee (\phi(r^-(t)) = \tau_i)) \wedge ((t - r_i^1) mod T_i \neq 0)$ alors la tâche τ_i a fini son exécution à la date $r^-(t)$ et ne commence pas une nouvelle instance à t donc τ_i est à l'état passif à la date t. Sinon, si la 3) qui est fausse c'est-à-dire que $\exists \tau_p \in pred(\tau_i) / d_{pi}(t) < k_{pi}$

alors la tâche τ_i n'a pas encore reçu toutes les k_{pi} données de son prédécesseur τ_p, donc elle ne peut pas démarrer son exécution. Sinon, si c'est la condition $4)$ qui est fausse c'est à dire que $\exists \tau_s \in succ(\tau_i)/d_{is}(t) \geq k_{is}$ alors la tâche τ_i a déjà produit k_{is} données qui ne sont pas encore toutes consommées par son successeur τ_s donc elle ne peut pas en produire d'autres avant que τ_s ne consomme celles déjà produites. Donc si l'une de ces conditions est fausse, alors la tâche τ_i ne peut pas être prête à l'exécution.

Si toutes les quatre conditions son vérifiées alors : τ_i est activée avant ou à la date t d'après la condition $1)$, τ_i n'a pas fini son exécution avant ou à t, c'est-à-dire qu'elle n'est pas à l'état passif d'après la condition $2)$; τ_i a reçu toutes les données de ses prédécesseurs avant ou à t, d'après la condition $3)$; τ_i ne produit pas plus de k_{is} données à la fois pour chacun de ses successeurs τ_s. Donc la tâche τ_i est prête à l'exécution. Donc si les quatre conditions sont vérifiées alors la tâche τ_i alors est prête à l'exécution. ∎

Définition 5.5.1 *On définit par $\Gamma_r(t)$ l'ensemble des tâches prêtes à l'exécution à la date t. $\Gamma_r^l(t)$ est donné par : $\Gamma_r(t) = \{\tau_i \in \Gamma_n / \tau_i$ est prête à l'exécution à t selon le théorème 5.1.*

Pour ordonnancer les tâches sans pertes de données, à chaque appel de l'ordonnanceur, la tâche sélectionnée pour s'exécuter doit être dans l'ensemble des tâches prêtes.

Définition 5.5.2 *On note par $\phi(t)$ la tâche sélectionnée pour s'exécuter à la date t. Elle correspond à la tâche la plus prioritaire à l'instant t dans $\Gamma_r(t)$.*

$\forall t \in I(t)$, $\phi(t)$ est donnée par :

$$\phi(t) = \tau_k / p_k(t) = max\{p_i(t)\}_{\tau_i \in \Gamma_r(t)} \tag{5.6}$$

avec $p_k(t)$ déterminée par la relation 5.3.

À la date t, c'est la tâche sélectionnée $\phi(t)$ qui s'exécute jusqu'à la prochaine date d'appel de l'ordonnanceur qui sélectionnera à cette date soit la même tâche qu'à t ou une autre tâche. Si aucune tâche n'est sélectionnée pour s'exécuter à une date t alors le processeur est inoccupé jusqu'au prochain appel de l'ordonnanceur et on note $\phi(t) = idle$.

5.5.2 Durée restant à exécuter avec coût exact de la préemption

Lors de l'analyse d'ordonnançabilité, le coût exact de la préemption est pris en compte dans la durée restant à exécuter par la tâche préemptée. En effet à chaque fois qu'une tâche τ_i se fait préempter à la date t on ajoute le coût associé à une préemption à la durée d'exécution qui reste à cette tâche. Ce qui rallonge le

temps de réponse de la tâche τ_i et permet de prendre en compte les préemptions potentielles engendrées par cette préemption. Ainsi toutes les préemptions sont prises en compte de manière exacte, sans faire aucune approximation comme c'est le cas dans les approches classiques.

On note par $c_i(t)$ la durée restant à exécuter par la tâche τ_i à la date t. $c_i(t)$ correspond au nombre d'unités de temps que doit compléter la tâche τ_i à partir de t pour finir son exécution. La figure 5.7, ci-dessous, donne une illustration de cette durée d'exécution. Dans cette figure la tâche τ_i est préemptée à la date t et sa durée restant à exécuter à t notée par $c_i(t)$ est rallongée d'une unité de temps sur cet exemple. Car on a fait l'hypothèse que le coût associé à une préemption, représenté ici en noir, était d'une unité de temps.

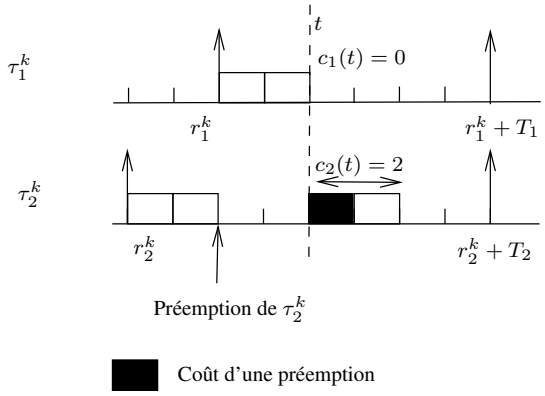

FIGURE 5.7 – Illustration de $c_1(t)$ et $c_2(t)$

À la date t, le calcul de $c_i(t)$ se fait en fonction de l'ordonnancement obtenu aux dates $r^-(t)$ et t. Ainsi nous avons les cas suivants :

- si τ_i est activée à t alors $c_i(t) = C_i$ sinon;
- si τ_i n'était pas la tâche sélectionnée pour s'exécuter lors de la dernière date d'appel de l'ordonnanceur $r^-(t)$ ou si τ_i a déjà terminé son exécution à cette date alors $c_i(t) = c_i(r^-(t))$ sinon;
- si τ_i était la tâche sélectionnée pour s'exécuter à $r^-(t)$ et qu'elle n'est pas préemptée à t alors $c_i(t) = (r^-(t) + c_i(r^-(t)) - t)$ sinon;
- si τ_i était la tâche sélectionnée pour s'exécuter à $r^-(t)$ et qu'elle est préemptée à t alors $c_i(t) = (r^-(t) + c_i(r^-(t)) - t) + \alpha$, avec α égal au coût d'une préemption.

$\forall t \in I(t)/t \geq r_i^1$, $c_i(t)$ est donnée par :

$$c_i(t) = \begin{cases} C_i & \text{si } (\frac{t-r_i^1}{T_i}) \in \mathbb{N} \quad \text{sinon} \\ c_i(r^-(t)) & \text{si } (\phi(r^-(t)) \neq \tau_i) \quad \text{sinon} \\ r^-(t) + c_i(r^-(t)) - t & \text{si } (\phi(t) = \tau_i) \vee \\ & \quad ((\phi(t) \neq \tau_i) \wedge (r^-(t) + c_i(r^-(t)) = t)) \quad \text{sinon} \\ (r^-(t) + c_i(r^-(t)) - t) + \alpha & \end{cases}$$

5.5.3 Échéance relative à une date d'appel de l'ordonnanceur

L'échéance d'une tâche τ_i relative à la date d'appel de l'ordonnanceur t est la durée $d_i(t)$ telle que τ_i doit obligatoirement terminer son exécution avant $t + d_i(t)$ sous peine de rater son échéance relative à sa dernière activation. La figure 5.8 donne une illustration de l'échéance relative de τ_i par rapport à t.

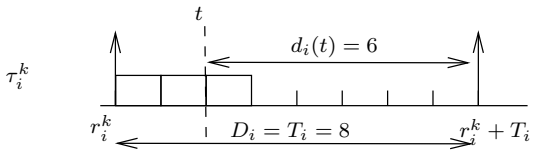

FIGURE 5.8 – Illustration de $d_i(t)$

Si t est une activation de la tâche τ_i, on a $d_i(t) = D_i$ et $d_i(t)$ décroît jusqu'à la prochaine activation de τ_i.

$\forall t \in I(t) \geq r_i^1$, $d_i(t)$ est donnée par :

$$d_i(t) = \begin{cases} D_i & \text{si } (\frac{t-r_i^1}{T_i}) \in \mathbb{N} \quad \text{sinon} \\ r^-(t) + d_i(r^-(t)) - t & \text{si } r^-(t) + d_i(r^-(t)) > t \text{ sinon} \\ 0 & \end{cases}$$

5.5.4 Condition d'ordonnançabilité

Dans cette section on présente la condition d'ordonnançabilité d'une tâche τ_i. Cette condition est nécessaire et suffisante.

Théorème 5.5.1 *Soit t une date d'appel de l'ordonnanceur. Une tâche $\tau_i \in \Gamma_n$ est ordonnançable à t avec prise en compte du exact de la préemption si et seulement si :*

$$\begin{array}{l}(c_i(t) \leq d_i(t)) \wedge \\ ((t \leq r_i^1) \vee (c_i(r^-(t)) = 0) \vee \\ (\phi(r^-(t)) = \tau_i) \vee ((t - r_i^1) mod T_i \neq 0))\end{array} \quad (5.7)$$

Preuve
Montrons que si la tâche τ_i est ordonnançable alors on a la condition 5.7 (condition nécessaire). Pour cela, supposons que la condition 5.7 est fausse, alors nous avons:

$$\begin{array}{l}((c_i(t) > d_i(t)) \vee \\ ((t > r_i^1) \wedge (c_i(r^-(t)) > 0) \wedge \\ (\phi(r^-(t)) \neq \tau_i) \wedge ((t - r_i^1) mod T_i = 0))\end{array}$$

Si $c_i(t) > d_i(t))$ alors la tâche τ_i ne pourra pas finir son exécution à $t + d_i(t)$ donc τ_i va dépasser son échéance à la date $t + d_i(t)$. Dans ce cas τ_i n'est pas ordonnançable. Si $((t > r_i^1) \wedge (\phi(r^-(t)) \neq \tau_i) \wedge (c_i(r^-(t)) > 0) \wedge (t - r_i^1) mod T_i = 0))$ cela signifie qu'à la dernière date d'appel de l'ordonnanceur $r^-(t)$, τ_i n'a pas été sélectionnée pour s'exécuter et que τ_i était déjà activée à cette date. Puisque $((t - r_i^1) mod T_i = 0))$, τ_i débute l'exécution d'une nouvelle instance à t alors qu'elle n'a pas fini son exécution sur sa dernière instance ($c_i(r^-(t)) > 0$), donc τ_i a dépassé l'échéance de sa dernière instance, ce qui veut dire que τ_i n'est pas ordonnançable. Donc si la condition 5.7 est fausse alors τ_i n'est pas ordonnançable. Ou bien, si $\tau_i \in \Gamma_n$ est ordonnançable alors la condition 5.7 est vérifiée.

Montrons que si la condition 5.7 est vérifiée alors la tâche τ_i est ordonnançable (condition suffisante). Pour cela, supposons que la condition 5.7 est vraie. Si $((c_i(t) \leq d_i(t)) \wedge ((c_i(r^-(t)) = 0) \vee (t \leq r_i^1)))$ alors τ_i a fini son exécution à $r^-(t)$, ou τ_i n'est pas encore activée à t ou bien τ_i vient d'être activée à t. Dans chacun de ces cas, la tâche τ_i ne rate pas son échéance donc est ordonnançable jusqu'à la date t. Sinon, si $((c_i(t) \leq d_i(t)) \wedge (\phi(r^-(t)) = \tau_i)$ alors τ_i était la tâche sélectionné à $r^-(t)$ et après son exécution nous avons $c_i(t) \leq d_i(t)$ donc τ_i est ordonnançable jusqu'à la date t. Sinon, si $((c_i(t) \leq d_i(t)) \wedge ((t - r_i^1) mod T_i \neq 0))$ alors τ_i n'était pas la tâche sélectionnée à la date $r^-(t)$ mais ne dépasse pas son échéance à t ($c_i(t) \leq d_i(t)$) et la date t ne coïncide pas au début d'exécution d'une nouvelle instance de τ_i ($(t - r_i^1) mod T_i \neq 0)$) dons τ_i est toujours ordonnançable à t. Donc si la condition 5.7 est vraie alors τ_i est ordonnançable.

Donc la tâche τ_i est ordonnançable si et seulement si la relation 5.7 est vraie. ■

Corollaire 5.5.1 *Une tâche τ_i est ordonnançable avec prise en compte du coût exact de la préemption si et seulement si elle est ordonnançable lors de chaque appel de l'ordonnanceur.*

Preuve
Montrons que si la tâche τ_i n'est pas ordonnançable lors de chaque appel de l'ordonnanceur alors elle n'est pas ordonnançable. Pour cela supposons qu'il existe une date d'appel de l'ordonnanceur à laquelle τ_i n'est pas ordonnançable. Donc τ_i rate son échéance dans une de ses instances dans l'intervalle d'étude donc n'est pas ordonnançable.

Montrons que si la tâche τ_i est ordonnançable lors de chaque appel de l'ordonnanceur alors elle est ordonnançable. Pour cela supposons que τ_i est ordonnançable lors de chaque appel de l'ordonnanceur. Donc τ_i ne rate aucune de ses échéances dans l'intervalle d'étude donc elle est ordonnançable. ∎

Corollaire 5.5.2 *Un ensemble de tâches dépendantes Γ_n est ordonnançable avec la prise en compte du coût exact de la préemption si et seulement si pour chaque appel de l'ordonnanceur à t, $\forall \tau_i \in \Gamma_n$, τ_i est ordonnançable selon le théorème 5.5.1.*

Preuve
Si Γ_n est ordonnançable alors aucune tâche ne rate son échéance donc toutes les tâches sont ordonnançables. $\forall t$ telle que t est une date d'appel de l'ordonnanceur, $\forall \tau_i \in \Gamma_n$, τ_i est ordonnançable donc τ_i vérifie la condition la 6.6 d'où τ_i est ordonnançable selon le théorème 5.5.1. Donc si Γ_n est odonnançable alors $\forall \tau_i \in \Gamma_n$ est ordonnançable selon le théorème 5.5.1.

Si $\forall t$ est une date d'appel de l'ordonnanceur, $\forall \tau_i \in \Gamma_n$, τ_i est ordonnançable selon le théorème 5.5.1 alors Γ_n ordonnançable. ∎

5.5.5 Prochaine date d'appel de l'ordonnanceur hors ligne

L'ordonnanceur hors ligne est appelé aux dates d'activation et de fin d'exécution des tâches afin de sélectionner la tâche à exécuter. Les dates de fin d'exécution des tâches sont déterminées en supposant que hors ligne, les tâches ont des durées d'exécution égales à leur pire temps d'exécution. Dans ce qui suit lorsqu'on parlera d'exécution d'une tâche on entend par là une exécution hors ligne.

On note par F l'ensemble ordonné des dates d'activation incluses dans I_n. L'ensemble F peut être déterminé avant l'exécution des tâches à l'aide des dates de première activation et des périodes des tâches. Cet ensemble nous permet de déterminer la prochaine activation qui arrive après un appel de l'ordonnanceur. F est donné par la relation 5.8. On suppose que les éléments de F sont dans l'ordre croissant des dates d'activation.

$$F = \{t \in I_n / \exists (\tau_i, k) \in (\Gamma_n, \mathbb{N}), t = r_i^1 + kT_i\} \tag{5.8}$$

On note par G, l'ensemble des dates d'activation et de fin d'exécution des tâches qui correspondent aux dates d'appel de notre ordonnanceur. Avant l'exécution des tâches on initialise G à l'ensemble F puisqu'on ne connaît pas encore les dates de fin d'exécution des tâches. Nous allons mettre à jour l'ensemble G au fur et à mesure de l'analyse d'ordonnançabilité des tâches en ajoutant les dates de fin d'exécution. Pour cela en partant de la plus petite des dates de première activation r_{min}, on détermine la prochaine date d'appel de l'ordonnanceur notée $r^+(r_{min})$ que l'on ajoute dans G. Si la tâche sélectionnée à la date r_{min}, termine son exécution avant la prochaine date d'activation, quelle que soit la tâche, alors $r^+(r_{min})$ est égale à la date de fin d'exécution de cette tâche sélectionnée à r_{min}. Sinon, $r^+(r_{min})$ correspond à la prochaine date d'activation. À la date $r^+(r_{min})$ on effectue le même traitement jusqu'à la fin de l'intervalle d'étude.

Définition 5.3 *Soit t une date d'appel de l'ordonnanceur qui ne peut être qu'une date d'activation ou de fin d'exécution d'une tâche. La prochaine date d'appel de l'ordonnanceur après t est donnée par la relation 5.9.*

$$r^+(t) = \begin{cases} t + c_k(t) & \textit{si } (t + c_k(t)) < a(t) \textit{ sinon} \\ \\ a(t) \end{cases} \tag{5.9}$$

avec $c_k(t)$ est la durée restant à exécuter par la sélectionnée à t et $a(t)$ est la date de l'activation d'une tâche qui arrive après t, $a(t)$ est l'élément qui succède t dans F.

La figure 5.9 donne une illustration du calcul de $I(t)$.

À la date t qui correspond à une date d'appel de l'ordonnanceur, après avoir calculé $r^+(t)$ on a $G = G \cup \{r^+(t)\}$.

5.5.6 Algorithme d'analyse d'ordonnançabilité monoprocesseur

L'analyse d'ordonnançabilité de l'ensemble Γ_n est réalisé par l'algorithme 6.

Avec l'algorithme 6 on parcourt l'intervalle d'étude I_n en se limitant aux dates d'activation et de fin d'exécution des tâches qui correspondent aux date d'appel de l'ordonnanceur. À chacune de ces dates on sélectionne la tâche à exécuter puis on vérifie s'il existe une tâche non ordonnançable en utilisant la condition 5.7 du théorème 5.5.1. Dès qu'il existe une tâche non ordonnançable, c'est-à-dire qu'une tâche qui ne vérifie pas la condition 5.7, alors Γ_n n'est pas ordonnançable. Sinon si toutes les tâches vérifient la condition 5.7 alors Γ_n est ordonnançable. À la fin de l'intervalle d'étude, si Γ_n est ordonnançable alors une table d'ordonnancement de Γ_n est produite qui indique pour chacune des dates d'appel de l'ordonnanceur

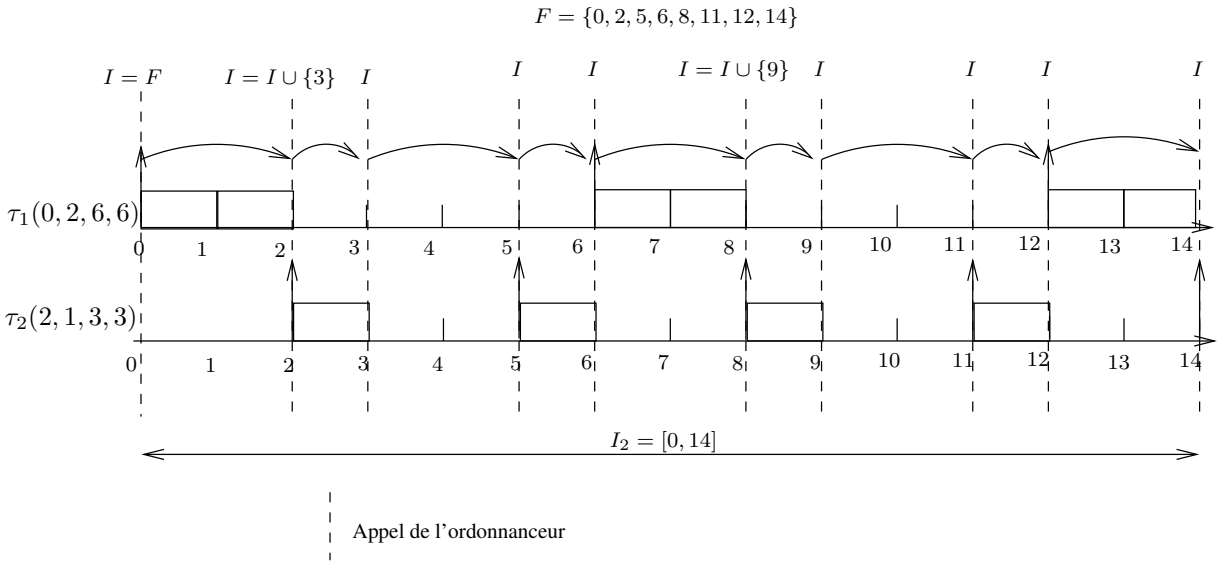

FIGURE 5.9 – Illustration des dates d'appel de l'ordonnanceur

(date d'activation et de fin d'exécution des tâches), la tâche qui s'exécute à cette date.

La complexité au pire cas de cette analyse d'ordonnançabilité est de $O(n.l)$, n est le nombre de tâches et l est la longueur de l'intervalle d'étude I_n.

5.5.7 Application

Considérons l'ensemble Γ_3 constitué de 3 tâches périodiques avec des contraintes de dépendances de données. Le graphe G_3' de la figure 5.10 représente les dépendances de données des tâches dans Γ_3. Nous allons faire l'analyse d'ordonnançabilité de Γ_3 sur un seul processeur.

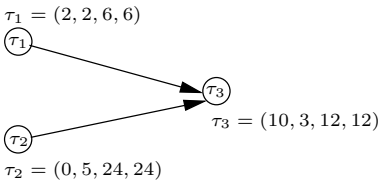

FIGURE 5.10 – Graphe G_3' représentant Γ_3

Selon le mécanisme de transfert de données présenté à la section 5.3, la tâche τ_1 doit produire 2 données à la tâche τ_3 et la tâche τ_2 doit produire 1 donnée à la

Algorithme 6 Analyse d'ordonnançabilité

1: $t \leftarrow r_{min}$
2: $G \leftarrow F$
3: $ordonnançable \leftarrow vrai$
4: **tant que** $(t < (tc + H_n)) \wedge (ordonnançable = vrai)$ **faire**
5: Calculer $\phi(t)$
6: $i \leftarrow 1$
7: **tant que** $(i \leq n) \wedge (ordonnançable = vrai)$ **faire**
8: **si** $(t \geq r_i^1)$ **alors**
9: Calculer $c_i(t)$
10: Calculer $d_i(t)$
11: **si** $((c_i(t) > d_i(t)) \vee$
$((t > r_i^1) \wedge (c_i(r^-(t)) > 0) \wedge (\phi(r^-(t)) \neq \tau_i) \wedge ((t - r_i^1) mod T_i = 0))$
alors
12: $ordonnançable \leftarrow faux$
13: **fin si**
14: **fin si**
15: $i \leftarrow i + 1$
16: **fin tant que**
17: $t \leftarrow r^+(t)$
18: $G \leftarrow G \cup \{r^+(t)\}$
19: **fin tant que**

tâche τ_3. On suppose que le coût d'une préemption est égal à $\alpha = 1$ unité de temps et que les priorités des tâches sont calculées selon l'algorithme Rate Monotonic présenté dans l'état de l'art. L'analyse d'ordonnançabilité de Γ_3 est réalisée sur l'intervalle $I_3 = [r_{min}, t_c + H_3]$ avec $H_3 = ppcm(T_1, T_2, T_3) = T_2 = 24$ et $t_c = r_{max} + H_3 = 10 + 24 = 34$. Donc $I_3 = [0, 34 + 24] = [0, 58]$.

Selon l'algorithme 6, Γ_3 est ordonnançable avec la prise en compte du coût exact de la préemption. La table d'ordonnancement de Γ_3 est présentée au tableau 5.2.

Dans ce tableau, $\phi(t) = idle$ correspond au moment où le processeur est inoccupé. En utilisant la table d'ordonnancement 5.2 on réalise le diagramme de Gantt de l'ordonnancement de l'ensemble des tâches Γ_3 présenté à la figure 5.11.

Dans la figure 5.11, nous observons que la tâche τ_2 est préemptée aux dates $t = 2, 26, 32, 50, 56$. Pour la tâche τ_2, c'est sa préemption à $t = 26$ qui a engendré sa préemption à $t = 32$. En effet si à la date $t = 26$ la tâche τ_2 n'était pas préemptée alors on aurait $c_2(26) = 4$ au lieu de $c_2(26) = 5$ donc τ_2 finirait son exécution à $t = 32$. Nous observons aussi que le mécanisme de transfert de don-

t	$\phi(t) = \tau_i$	$c_i(t)$
0	τ_2	5
2	τ_1	2
4	τ_2	4
8	τ_1	2
10	τ_3	3
13	$idle$	1
14	τ_1	2
16	$idle$	4
20	τ_1	2
22	τ_3	3
24	τ_3	1
25	τ_2	5
26	τ_1	2
28	τ_2	5
32	τ_1	2
34	τ_2	2
36	τ_3	3
38	τ_3	1
39	τ_1	2
41	$idle$	3
44	τ_1	2
46	τ_3	3
48	τ_3	1
49	τ_2	5
50	τ_1	2
52	τ_2	5
56	τ_1	2
58	τ_2	2

TABLE 5.2 – Table d'ordonnancement de Γ_3

nées présenté à la section et les contraintes d'exclusion mutuelles sont bien pris en compte dans l'ordonnancement des tâches. La tâche τ_3 consomme pour chacune de ses exécutions 2 données produites par τ_1 et 1 donnée produite par τ_2. A la date $t = 38$, même si la tâche τ_1 est plus prioritaire que la tâche τ_3 c'est cette dernière qui s'exécute à cette date car τ_1 est bloquée par τ_3 qui lit sur la mémoire de donnée de τ_i.

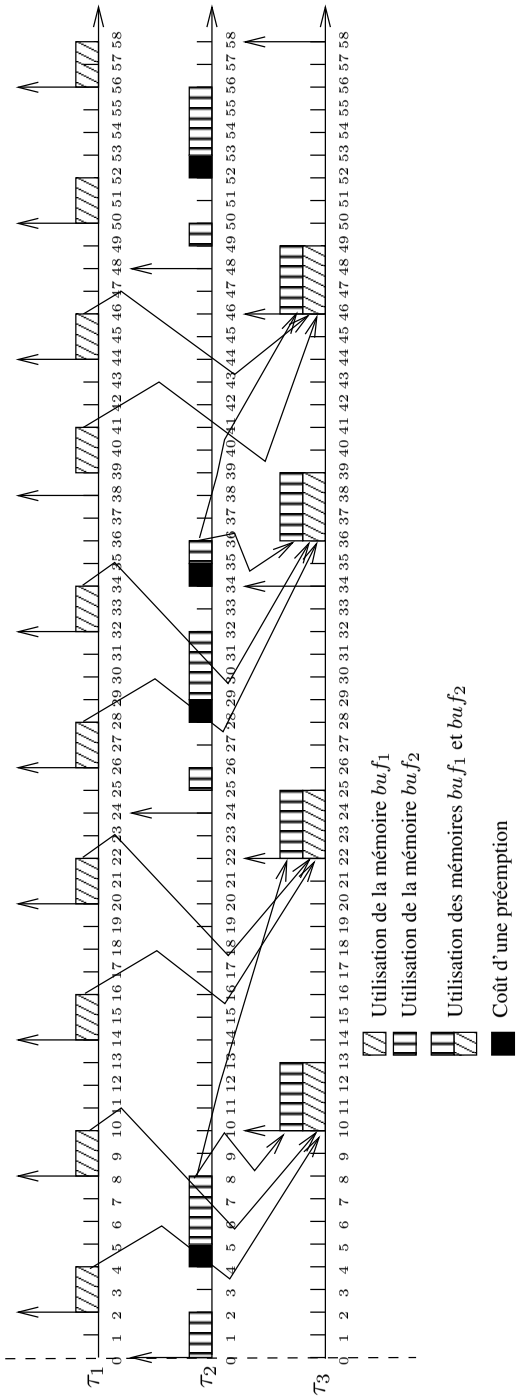

FIGURE 5.11 – Résultat de l'ordonnancement de Γ_3

5.6 Viabilité

La viabilité de la condition d'ordonnançabilité présentée à la section précédente section est montrée par le théorème 5.6.1.

Théorème 5.6.1 *Soit $\Gamma_n = \{\tau_1, \tau_2, \cdots, \tau_i, \cdots, \tau_{n-1}, \tau_n\}$, un ensemble de tâches dépendantes où les tâches sont triées dans l'ordre décroissant de leurs priorités selon l'algorithme à priorités fixes noté $Algo$ avec $\tau_i = (r_i^1, C_i, D_i, T_i)$.*

Si Γ_n est ordonnançable alors lors de l'exécution, $\Gamma_n^{'} = \{\tau_1, \tau_2, \cdots, \tau_i^{'}, \cdots, \tau_{n-1}, \tau_n\}$, avec $\tau_i^{'} = (r_i^1, C_i^{'}, D_i, T_i)$ et $C_i^{'} < C_i$ est aussi ordonnançable.

Preuve
Les tâches dans Γ_n ont les mêmes dates de première activation et les mêmes périodes que celles dans $\Gamma_n^{'}$. Donc Γ_n et $\Gamma_n^{'}$ ont le même intervalle d'étude.

Supposons que Γ_n est ordonnançable et que T est une table d'ordonnancement de Γ_n. Dans la table d'ordonnancement T, remplaçons la tâche τ_i par la tâche $\tau_i^{'}$, on obtient la table $T^{'}$. Montrons que dans la table d'ordonnancement $T^{'}$ toutes les tâches de $\Gamma_n^{'}$ respectent leurs échéances.

Avec l'ordonnancement obtenu avec $T^{'}$, les tâches $\tau_j \in \Gamma_n^{'}/i \neq j$ respectent leurs échéances puisque ce sont les mêmes tâches que celles dans Γ_n ($\{\tau_j \in \Gamma_n/i \neq j\} = \{\tau_j \in \Gamma_n^{'}/i \neq j\}$) et qu'elles ont les mêmes plages d'exécution que dans T. Supposons que la tâche $\tau_i^{'}$ n'est pas ordonnançable, c'est-à-dire qu'il existe au moins une instance de $\tau_i^{'}$ dans laquelle $\tau_i^{'}$ ne respecte pas son échéance. Cette instance correspond à la même instance que celle τ_i dans T. Hors nous avons $C_i^{'} < C_i$, ce qui veut dire que τ_i ne respecte pas son échéance sur une instance donc n'est pas aussi ordonnançable, ce qui est absurde d'après notre supposition que Γ_n était ordonnançable. Donc $\tau_i^{'}$ est ordonnançable de même que toutes les tâches dans $\Gamma_n^{'}$ sont ordonnançables et que $T^{'}$ est une table d'ordonnancement de $\Gamma_n^{'}$. ∎

5.7 Conclusion

Dans ce chapitre nous avons étudié l'ordonnancement des tâches dépendantes. Nous avons présenté une condition d'ordonnançabilité de ces tâches avec la prise en compte du coût exact de la préemption sans perte de données. Nous allons utiliser cette condition d'ordonnançabilité dans l'ordonnancement multiprocesseur des tâches en appliquant la stratégie par partitionnement.

Chapitre 6

Ordonnancement multiprocesseur de tâches dépendantes

6.1 Introduction

Dans ce chapitre nous traitons le problème d'ordonnancement multiprocesseur de tâches dépendantes avec la prise en compte du coût exact de la préemption. Nous proposons une heuristique d'ordonnancement multiprocesseur qui alloue les tâches aux processeurs puis les ordonnance en prenant en compte le coût exact de la préemption sans migration de tâches. Afin de prendre en compte les coûts de communication inter-processeurs, nous allons étendre en multiprocesseur l'analyse d'ordonnançabilité présentée à la section 5.5 du chapitre précédent.

Ce chapitre est organisé comme suit : la section 6.2 présente le modèle de tâches, la section 6.3 présente le modèle d'architecture, la section 6.4 présente le principe d'allocation des tâches aux processeurs, la section 6.5 présente l'analyse d'ordonnançabilité multiprocesseur, la section 6.6 présente l'heuristique d'ordonnancement multiprocesseur qui alloue et ordonnance les tâches, la section 6.7 présente l'application de notre heuristique d'ordonnancement multiprocesseur et enfin la section 6.8 présente l'analyse de performance de notre heuristique que l'on compare avec l'algorithme exact $B\&B$ appliqué à des tâches dépendantes.

6.2 Modèle de tâches

Soit Γ_n un ensemble de n tâches dépendantes selon le modèle de tâches défini au chapitre précédent. Nous utilisons aussi le même mécanisme de transfert de données que celui présenté à section 5.3 du chapitre précédent.

6.3 Modèle d'architecture

On considère une architecture multiprocesseur distribuée composée de m processeurs identiques. Cette architecture est représentée par un graphe non orienté où chaque sommet est un processeur ou un médium de communication et chaque arc est une connexion entre un processeur et un médium de communication [Gra00]. Un médium de communication peut être une liaison $point$-à-$point$, une liaison $multi$-$point$ (bus) ou une mémoire partagée. Afin d'éviter qu'une communication bloque une tâche sur un processeur on suppose que chaque processeur est doté d'une unité de communication appelé $communicateur$ qui permet d'avoir au sein de chaque processeur un parallélisme entre les exécutions des tâches, et les communications entre processeurs. Les tâches sont exécutées par les opérateurs notés OPR_i et les communications par les communicateurs notés COM_i.

Un exemple de graphe d'architecture est présenté à la figure 6.1 ci-dessous. Dans cette figure, M_1 etM_2 représentent les médiums de communication et RAM représente la mémoire programme d'un processeur.

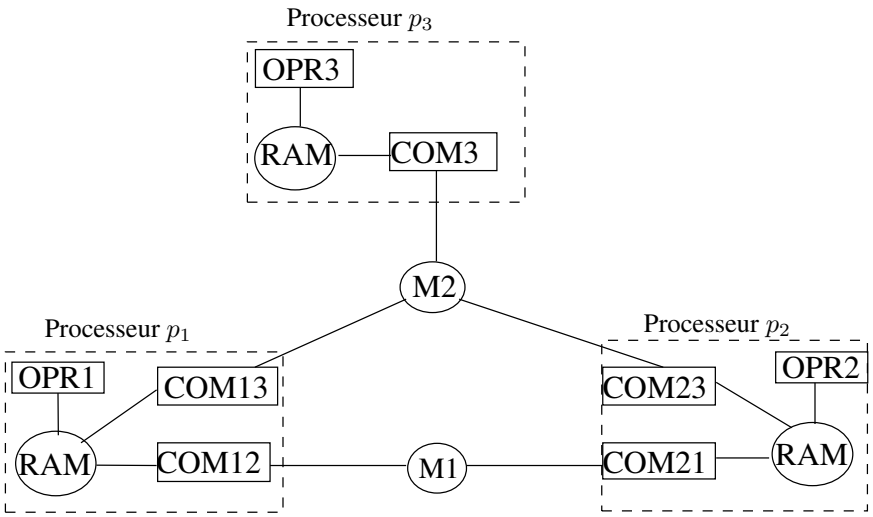

FIGURE 6.1 – Illustration d'un graphe d'architecture

Soient deux tâches τ_i et τ_j, telles que τ_i est la tâche productrice des données de τ_j et τ_i est la tâche consommatrice des données de τ_i, exécutées respectivement sur les processeurs différents p_l et p_k. Le coût de transfert d'une donnée de τ_i du processeur p_l au processeur p_k, appelé coût de communication entre p_l et p_k, est considéré comme étant une constante. On note par com_{lk} ce coût .

On suppose que tous les processeurs ont la même horloge et donc la même origine des temps [PBAF10].

6.4 Allocation des tâches aux processeurs

Contrairement aux tâches indépendantes où on effectue en même temps l'allocation et l'ordonnancement des tâches avec un condition suffisante d'ordonnançabilité, pour les tâches dépendantes on effectue en même temps l'allocation et un pré-ordonnancement des tâches avec une condition d'ordonnançabilité nécessaire qui permet de décider pour chaque tâche le processeur où elle est allouée. Puis on raffine ce pré-ordonnancement avec notre analyse d'ordonnançabilité qui permet de prendre en compte, en plus du coût de la préemption comme dans la section 5.5 du chapitre précédent, le coût des communications inter-processeur. En effet à cause des dépendances de données, l'analyse d'ordonnaçabilité de chaque tâche doit être effectuée avec ses prédécesseurs et ses successeurs. Ceci afin de pouvoir assurer que chaque tâche a reçu toutes les données de ses prédécesseurs et aussi qu'elle a produit le bon nombre de données pour ses successeurs. Ainsi après avoir réalisé l'allocation des tâches sur les différents processeurs, on peut les ordonnancer plus finement avec une condition d'ordonnançabilité suffisante en prenant en compte le coût exact de la préemption et le coût des communications inter-processeur. Dans la suite on présente le principe de l'allocation et du pré-ordonnancement des tâches.

6.4.1 Condition nécessaire d'ordonnançabilité

Pour un processeur donné si la condition nécessaire d'ordonnançabilité n'est pas vérifiée par une tâche, alors cette tâche n'est pas ordonnançable sur ce processeur, donc elle ne peut pas lui être alloué. Cependant une tâche peut vérifier cette condition sur un processeur sans pour autant être ordonnançable sur ce processeur. Dans le cas de l'ordonnancement des tâches indépendantes la condition d'ordonnançabilité utilisée est fondée sur l'opération $\oplus$.

À cause des contraintes de dépendances de données, une tâche τ_j en attente de recevoir des données de ses prédécesseurs voit sa date de début d'exécution au plus tôt retardée. Donc il existe une date avant laquelle τ_j ne pourra pas démarrer son exécution, c'est cette date qui correspond en réalité à sa date de première activation. Cette date dépend du processeur sur lequel la tâche τ_j est allouée et des processeurs sur lesquels ses prédécesseurs sont allouées.

Soit $pred(\tau_j)$ l'ensemble des tâches prédécesseurs de τ_j et $k_{ij} = \lceil \frac{T_j}{T_i} \rceil$ le nombre de données que doit produire $\tau_i \in pred(\tau_j)$ pour τ_j, k_{ij} correspond aussi au nombre de fois que τ_i doit s'exécuter avant le début d'exécution de τ_j.

Lemme 6.1 *La date de début d'exécution au plus tôt de la première instance de la tâche τ_i sur le processeur p_k est donnée par :*

$$r_j^{(1,k)} = max(r_j^1, max(r_i^{(1,l)} + (k_{ij} - 1) * T_i + C_i + com_{lk})_{\tau_i \in pred(\tau_j)}) \quad (6.1)$$

com_{lk} est le coût de communication entre le processeur p_l où est allouée la tâche $\tau_i \in pred(\tau_j)$ et le processeur p_k. Si $p_l = p_k$ alors $com_{lk} = 0$.

Preuve
La tâche τ_j activée à la date r_j^1 doit attendre que chacun des ses prédécesseurs τ_i s'exécute k_{ij} fois afin de lui transférer k_{ij} données. $\tau_i \in pred(\tau_j)$ commence son exécution au plus tôt à la date $r_i^{(1,l)}$ alors τ_i finira de produire les k_{ij} premières données au plus tôt à la date $r_i^{(1,l)} + (k_{ij} - 1) * T_i + C_i$. Donc les k_{ij} premières données produites par τ_i seront disponibles au plus tôt à la date $r_i^{(1,l)} + (k_{ij} - 1) * T_i + C_i + com_{lk}$ car il faut prendre en compte le temps de transfert de la dernière donnée produite du processeur p_l au processeur p_k. D'où la date de début d'exécution au plus tôt de τ_j est donnée par la relation 6.1.

■

Lemme 6.2 *Soit la tâche τ_j sur le processeur k. L'échéance de la tâche τ_j relative à la date $r_j^{(1,k)}$ est donnée par :*

$$D_j^k = D_j - (r_j^{(1,k)} - r_j^1) \quad (6.2)$$

Preuve
La tâche τ_j est activée à la date r_j^1 donc elle doit finir son exécution au plus tard à la date $r_j^1 + D_j$. Donc l'échéance de τ_j relative à la date $r_j^{(1,k)}$ est égale à $(r_i^1 + D_j) - r_j^{(1,k)} = D_i - (r_j^{(1,k)} - r_j^1)$. D'où la relation 6.2. ■

Si la tâche τ_j n'a pas de prédécesseur alors $r_j^{(1,k)} = r_j^1$ et $D_j^k = D_j$ quel que soit le processeur k sur lequel τ_j est allouée. $r_j^{(1,k)}$ et D_j^k sont illustrées par la figure 6.2.

Théorème 6.1 *Soit τ_j une tâche allouée à un processeur p_k. Si τ_j est ordonnançable sur p_k alors elle doit vérifier nécessairement la condition donnée à la relation 6.3.*

$$C_j \leq D_j^k \quad (6.3)$$

Preuve
Supposons que la condition donnée à la relation 6.3 n'est pas vérifiée donc on a $C_j > D_j^k$. D'après les lemmes 6.1 et 6.2, la tâche τ_j ne peut pas démarrer

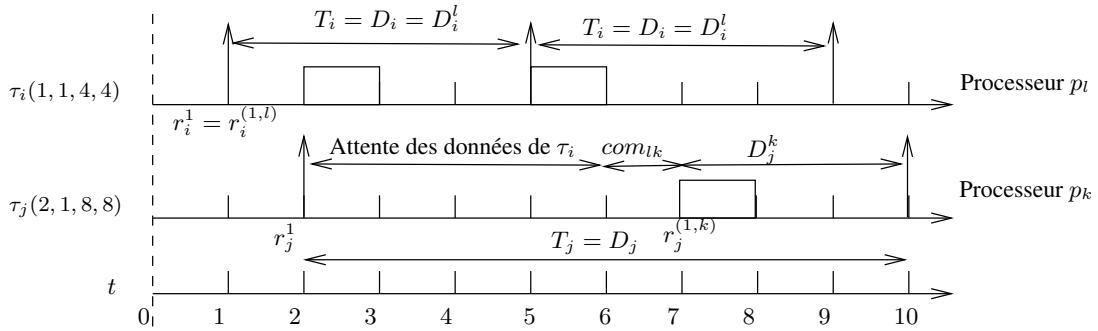

FIGURE 6.2 – Illustration de $r_j^{(1,k)}$ et D_j^k

l'exécution de sa première instance avant la date $r_j^{(1,k))}$ avec D_j^k l'échéance de τ_j relative à la date $r_j^{(1,k))}$. Donc la tâche τ_j doit terminer l'exécution de sa première instance au plus tard à la date $r_j^{(1,k))} + D_j^k$. Puisque τ_j ne peut commencer son exécution avant $r_j^{(1,k))}$ donc τ_j terminera au plus tôt l'exécution de sa première instance à la date $r_j^{(1,k))} + C_j$. Et comme nous $C_j > D_j^k$ alors $r_j^{(1,k))} + C_j > r_j^{(1,k))} + D_j^k$ et τ_j dépassera son échéance dans sa toute première instance.

Donc si τ_j est ordonnançable sur p_k alors elle doit vérifier la condition donnée à la relation 6.3. ∎

Durant l'allocation des tâches nous utiliserons la condition 6.3 pour allouer les tâches aux processeurs. C'est à dire qu'une tâche ne peut être allouée à un processeur que si la condition 6.3 est vérifiée sur ce processeur.

Remarque 6.4.1 *La condition 6.3 est une condition nécessaire d'ordonnançabilité, une tâche qui est allouée à un processeur n'est que potentiellement ordonnançable. Mais cette condition permet de prévoir lors de l'allocation des tâches les coûts de transfert de données. Ceci permet d'augmenter les chances des tâches d'être ordonnançables lors de l'étape de l'analyse d'ordonnançabilité et d'ordonnancement.*

6.4.2 Fonction de coût

Durant l'allocation des tâches nous allons maximiser la fonction de coût correspondant à la marge d'exécution des tâches, donnée par la définition 6.4.1. c'est-à-dire que si une tâche peut être allouée à plusieurs processeurs, nous choisirons le processeur qui lui donne la plus grande marge d'exécution. Ce choix de maximiser la marge d'exécution permet de prévoir lors de l'allocation les éventuels coûts

de préemptions des tâches. Il permet aussi de minimiser le makespan des tâches (voir définition 5.1) car en maximisant la marge d'exécution des tâches on réduit leur temps de réponse.

Définition 6.4.1 *La fonction de coût de la tâche τ_i sur le processeur p_k est définie par la marge d'exécution de cette tâche sur sa première instance. Cette marge est donnée par :*

$$M_i^k = \begin{cases} (r_i^1 + D_i) - (r_i^{(1,k)} + C_i) \textit{ si } pred(\tau_i) \neq \{\emptyset\} \textit{ sinon} \\ D_i - (\sum_{(\tau_j \in \Gamma^k)} C_j + C_i) \end{cases} \quad (6.4)$$

avec Γ^k l'ensemble des tâches déjà allouées au processeur p_k qui sont plus prioritaires que τ_i.

La marge d'exécution d'une tâche sur une instance correspond à la durée entre la date de fin d'exécution de cette tâche et son échéance absolue [AH98]. Donc plus tôt une tâche termine son exécution plus sa marge d'exécution est importante. La marge d'exécution d'une tâche τ_i varie en fonction du processeur sur lequel τ_i est allouée. Ainsi si une tâches est allouée sur le même processeur que ses prédécesseurs sa marge d'exécution est plus importante puisque les coûts de transferts de données sont nuls. Si une tâche n'a pas de prédécesseur, sa marge d'exécution sur un processeur est déterminée en fonction des tâches qui sont déjà allouées sur ce processeur et qui sont plus prioritaires qu'elle.

La figure 6.3 donne une illustration de la marge d'exécution d'une tâche.

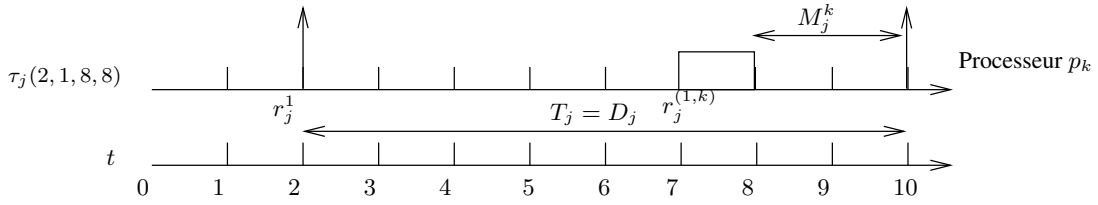

FIGURE 6.3 – Illustration de la marge d'exécution de τ_j

6.4.3 Heuristique d'allocation

Notre heuristique d'allocation donnée dans l'algorithme 7 prend en entrée un ensemble Γ_n composé de n tâches dépendantes représenté par un graphe de dépendances de tâches G_n et un ensemble Π_m de m processeurs identiques représenté

par un graphe de processeurs P_n. Si l'allocation réussit, l'heuristique fournit en sortie m sous-ensembles distincts de l'ensemble Γ_n et chaque sous-ensemble noté Γ_{n_l} avec $1 \leq l \leq m$ composé de n_l tâches est alloué au processeur p_l. Ce résultat est appelé une solution de l'allocation des tâches de Γ_n au processeur de Π_m et on le note par S. C'est une heuristique de type liste, on initialise la liste des tâches candidates W à l'ensemble des tâches sans prédécesseurs dans le graphe G_n. À chaque étape de l'allocation, la tâche τ_i la plus prioritaire est sélectionnée dans W pour être allouée à son meilleur processeur selon la fonction de coût définie à la section 6.4.2. Pour cela on parcourt l'ensemble des m processeurs pour vérifier si τ_i vérifie la condition 6.3. Si c'est le cas on calcule la marge d'exécution de cette tâche sur ce processeur. Après avoir parcouru tous les processeurs, parmi les processeurs sur lesquels la tâche sélectionnée τ_i vérifie la condition 6.3 on alloue τ_i au processeur p_j qui lui donne la plus grande marge d'exécution et cette allocation est définitive, c'est-à-dire qu'elle ne sera pas remise en cause. Une fois que la tâche sélectionnée τ_i est allouée à un processeur, la liste des tâches candidate W est mis à jour par suppression de τ_i et par ajout des successeurs de τ_i qui n'ont pas de prédécesseur dans W. Si on arrive à allouer toutes les tâches de Γ_n aux processeurs de Π_m alors l'allocation a réussi et on a en sortie l'ensemble des processeurs Π_m et sur chaque processeurs p_j les tâches allouées sur ce processeur.

La complexité au pire cas de cette heuristique d'allocation est $O(n.m)$.

6.5 Analyse d'ordonnançabilité multiprocesseur

Après avoir alloué les tâches aux différents processeurs on procède à l'analyse d'ordonnançabilité des tâches sur ces processeurs. Puisque nous appliquons la stratégie d'ordonnancement par partitionnement, chaque processeur p_l a son propre ordonnanceur, donc il faut faire l'analyse d'ordonnançabilité pour chaque processeur afin de garantir l'ordonnançabilité de l'ensemble des tâches. Pour cela nous allons étendre l'analyse d'ordonnançabilité monoprocesseur présentée au chapitre précédent pour prendre en compte les coûts de communication inter-processeurs afin de pouvoir l'appliquer sur chaque processeur.

Dans la suite de cette section on détaille le principe des communications inter-processeurs, l'analyse d'ordonnançabilité des tâches allouées à un processeur qui tient compte de ces communications inter-processeurs, la condition d'ordonnançabilté multiprocesseur et l'algorithme d'analyse d'ordonnançabilité multiprocesseur.

Algorithme 7 Heuristique d'allocation

1: Initialiser la liste des tâches candidates W à l'ensemble des tâches sans prédécesseurs dans G_n
2: $arrêt \leftarrow faux$
3: **tant que** $(W \neq \{\emptyset\} et (arrêt = faux)$ **faire**
4: Sélectionner la tâche la plus prioritaire dans W notée τ_i
5: **pour** $k = 1$ à m **faire**
6: Calculer D_i^k
7: **si** $(C_i \leq D_i^k)$ **alors**
8: Calculer la marge d'exécution M_i^k de la tâche τ_i sur le processeur p_k
9: **fin si**
10: **fin pour**
11: **si** il existe au moins un processeur sur lequel $(C_i \leq D_i^k)$ **alors**
12: Allouer τ_i au processeur p_k qui maximise M_i^k
13: **sinon**
14: $arrêt \leftarrow vrai$
15: **fin si**
16: Supprimer τ_i de l'ensemble W
17: Ajouter dans W toutes les successeurs de τ_i qui n'ont pas de prédécesseurs dans W
18: **fin tant que**
19: **si** $W = \{\emptyset\}$ **alors**
20: L'allocation a réussi et on a en sortie m sous-ensemble distincts de Γ_n et chaque sous-ensemble Γ_{n_l} (avec $1 \leq l \leq m$) composé de n_l tâches est alloué au processeur p_l.
21: **sinon**
22: L'allocation n'a pas réussie
23: **fin si**

6.5.1 Communication inter-processeurs

L'allocation des tâches aux processeurs peut entraîner des transferts de données entre des tâches allouées à des processeurs différents, dans ce cas on parle de communications inter-processeurs. On les appellera par la suite communications. Par exemple soit le couple de tâches (τ_i, τ_j) tel que τ_i est la tâche productrice de données allouée au processeur p_l et τ_j est la tâche consommatrice de données allouée au processeur p_k. Le transfert d'une donnée produite par τ_i, du processeur p_l au processeur p_k, entraîne une communication entre les processeurs p_l et p_k. On suppose que sur le processeur p_l, si la tâche τ_i produit à la date t une donnée, celle-ci sera au plus tard disponible pour la tâche τ_j, qui est allouée à p_k,

à la date $t + com_{lk}$. com_{lk} représente le pire temps de transfert d'une donnée, produite par une tâche, du processeur p_l au processeur p_k. À la date $t + com_{lk}$, la tâche τ_j peut consommer la donnée produite à t par la tâche τ_i. Ainsi nous n'avons pas besoin de synchroniser l'exécution des tâches sur les processeurs p_l et p_k puisque la tâche consommatrice τ_j connaît exactement la date à laquelle elle peut consommer la donnée de τ_i car, tous les processeurs ont la même horloge et donc la même origine des temps. La communication entre ces deux processeurs est exécutée sur ce qu'on a appelé un processeur de communication noté par $pcom = (COM_l, M, COM_k) = (COM_k, M, COM_l)$ (COM_l est le communicateur de p_l, M le médium de communication entre p_l et p_k, et COM_k est le communicateur de p_k). Une communication est formée d'un $send$ suivi d'un $receive$ dans le cas d'un médium de communication par passage de messages. Une communication est formée d'un $write$ suivi d'un $read$ dans le cas d'un médium de communication par mémoire partagée. Les processeurs de communication et les processeurs de calculs ont tous la même base de temps. Les figures 6.4 et 6.5 illustrent respectivement un processeur de communication et une communication entre deux processeurs.

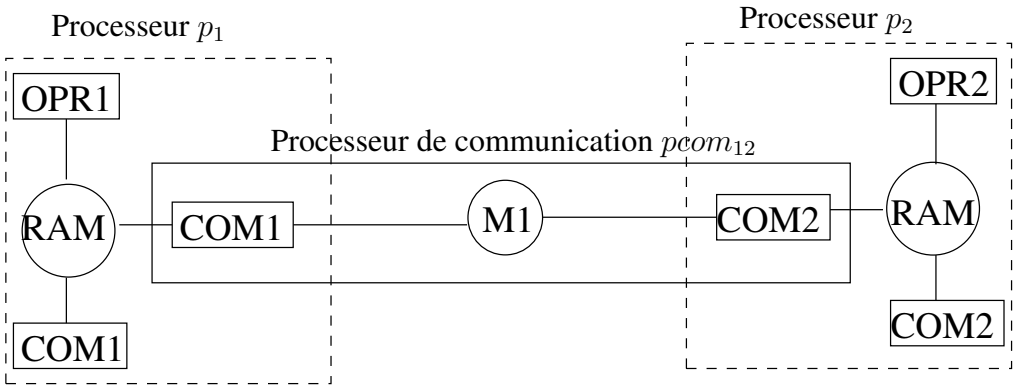

FIGURE 6.4 – Illustration d'un processeur de communication

On suppose que les communications exécutée sur les processeurs de communication sont non préemptibles. C'est-à-dire que plusieurs communications ne peuvent pas avoir lieu simultanément sur un même processeur de communication. Donc si lors de l'analyse d'ordonnançabilité plusieurs communications doivent s'exécuter dans le même intervalle de temps notée com_{lk} plus haut, on les ordonnance en fonction d'une priorité. Celle-ci est égale à la priorité de la tâche recevant les données par cette communication.

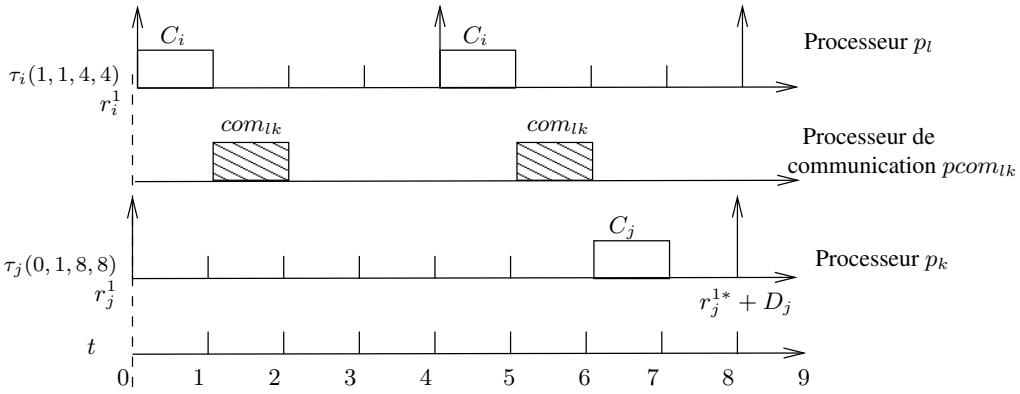

FIGURE 6.5 – Illustration d'une communication entre deux processeurs

6.5.2 Analyse d'ordonnançabilité des tâches allouées à un processeur

L'analyse d'ordonnançabilité des tâches allouées à un processeur p_l est réalisée selon le même principe que celle présentée à la section 5.5 du chapitre précédent. En effet sur un processeur p_l, on part de la plus petite des dates de première activation des tâches allouées à ce processeur que l'on note r^l_{min}. Ensuite on appelle notre ordonnanceur hors ligne à cette date qui sélectionne la tâche à exécuter. Puis en utilisant une condition d'ordonnançabilité nécessaire et suffisante on vérifie si cette tâche sélectionnée à r^l_{min} ainsi que toutes les autres tâches, sont ordonnançables à cette date. Cette condition d'ordonnançabilité utilise la durée restant à exécuter pour chaque tâche et l'échéance de chaque tâche relative à la date r^l_{min}. La durée restant à exécuter prend en compte le coût exact de la préemption. Si toutes les tâches sont ordonnançables à la date r^l_{min}, on détermine la prochaine date d'appel de notre ordonnanceur. Sinon l'ensemble des tâches allouées à p_l n'est pas ordonnançable. La prochaine date d'appel de l'ordonnanceur peut être une date d'activation, une date de fin d'exécution d'une tâche mais aussi une date de réception de données. Au prochain appel de l'ordonnanceur, on répète le même traitement jusqu'à la fin de l'intervalle d'étude I_n présenté à la section 5.2. Il faut noter que la sélection de la tâche à exécuter s'effectue en tenant compte des coûts de communication, c'est-à-dire que la tâche sélectionnée doit nécessairement recevoir au plus tard à r^l_{min} toutes les données produites par ses prédécesseurs. Lorsqu'on arrive à la fin de l'intervalle I_n, si l'ensemble des tâches allouées à p_l est ordonnançable, le résultat de l'ordonnancement sur p_l est donné par la table d'ordonnancement T^l. Cette table donne l'ordre total d'exécution des tâches sous la forme de leur date de début et de fin d'exécution dans l'intervalle

d'étude.

Dans cette section, on détaille pour un processeur p_l la sélection de la tâche à exécuter en tenant compte des coût de communication, le calcul de la durée restant à exécuter de chaque tâche, le calcul de l'échéance de chaque tâche relative à la date d'appel de l'ordonnanceur, la condition d'ordonnançabilité d'une tâche et enfin le calcul de la prochaine date d'appel de l'ordonnanceur qui peut être aussi une date de réception de données. Ces calculs sont faits chaque fois que l'ordonnanceur du processeur p_l est appelé.

Sélection de la tâche à exécuter sur p_l

Dans la mise à jour de la liste des tâches prêtes du processeur p_l, il faut tenir compte des coûts de communication. Lorsqu'une tâche, allouée à un processeur différent de p_l, produit une donnée à la date t, cette donnée n'est pas immédiatement disponible sur le processeur p_l car le transfert de la donnée produite par l'autre processeur prend du temps. Le théorème 6.2 ci-dessous donne les conditions que doit vérifier une tâche sur p_l pour être prête à l'exécution.

Théorème 6.2 *Soit Γ_{n_l} l'ensemble des tâches allouées au processeur p_l. Une tâche $\tau_i \in \Gamma_{n_l}$ est prête à l'exécution à la date t si et seulement si :*

1. $t \geq r_i^1$,
2. $(r^{l-}(t) + c_i(r^{l-}(t)) > t) \vee ((c_i(r^{l-}(t)) > 0) \wedge (\phi(r^{l-}(t)) \neq \tau_i)) \vee ((t - r_i^1) mod T_i = 0)$,
3. $\forall \tau_p \in pred(\tau_i), ((d_{pi}(r^{k-}(t)) \geq k_{pi}) \wedge (t - r^{k-}(t) \geq com_{kl}))$,
4. $\forall \tau_s \in succ(\tau_i), d_{is}(t) < k_{is}$.
   Avec τ_p allouée au processeur p_k avec $r^{l-}(t)$ est la dernière date d'appel de l'ordonnanceur du processeur p_l.

Preuve

Par rapport au théorème 5.1, c'est uniquement la condition $3)$ qui change. Cette condition garantit que la tâche τ_i reçoit toutes données nécessaires avant de commencer son exécution. Supposons que la condition 3 est fausse, c'est-à-dire que $\exists \tau_p \in pred(\tau_i)/(d_{pi}(r^{k-}(t)) - 1 < k_{pi}) \vee (t - r^{k-}(t) < com_{kl})$. Si $(d_{pi}(r^{k-}(t)) - 1 < k_{pi}) \vee (t - r^{k-}(t) < com_{kl}))$ alors $(d_{pi}(r^{k-}(t)) \leq k_{pi} + 2)$, qui veut dire que pour que la tâche τ_i puisse être prête à la date t il faut que la tâche τ_p produise entre $r^{k-}(t)$ et t au minimum deux données. Or ceci n'est pas possible car entre ces date, τ_p ne peut s'exécuter au maximum une seule fois donc ne peut pas produire deux données. Donc la tâche τ_i ne peut pas être prête à l'exécution car elle n'a pas encore reçu toutes les données de son prédécesseur τ_p. De même que si $(t - r^{k-}(t) < com_{kl})$ alors la dernière donnée produite à la date $r^{k-}(t)$ ne pas

être reçu à t. Donc la tâche τ_i ne peut pas être prête à l'exécution car elle n'a pas encore reçu toutes les données de son prédécesseur τ_p.

Supposons que la conditions 3) est vérifiée, c'est-à-dire que $\forall \tau_p \in pred(\tau_i)$, $((d_{pi}(r^{k-}(t)) \geq k_{pi}) \wedge (t - r^{k-}(t) \geq com_{kl}))$ alors la $k_{pi}^{ème}$ donnée produite par $\tau_p \in pred(\tau_i)$ et non encore consommée par τ_j a été produite avant ou à la date $r^{k-}(t)$. Et puisque $(t - r^{k-}(t) \geq com_{kl})$ donc cette donnée est reçue sur p_l avant ou à la date t car $r^{k-}(t) + com_{kl} \leq t$. Donc à la date t, la tâche τ_j a reçu toutes les données de ses prédécesseurs.

■

Définition 6.5.1 *On définit par $\Gamma_r^l(t)$ l'ensemble des tâches prêtes à l'exécution sur le processeur p_l à la date t. $\Gamma_r^l(t)$ est donné par : $\Gamma_r^l(t) = \{\tau_i \in \Gamma_{n_l} / \tau_i$ est prête à l'exécution à t selon le théorème 6.2.*

Définition 6.5.2 *On note par $\phi^l(t)$ la tâche sélectionnée pour s'exécuter sur le processeur p_l à la date t. $\phi^l(t)$ est donnée par :*

$$\phi^l(t) = \tau_i / p_i(t) = max\{p_j(t)\}_{\tau_j \in \Gamma_r^l(t)} \tag{6.5}$$

avec $p_i(t)$ déterminée par la relation 5.3.

Durée restant à exécuter avec prise en compte du coût exact de la préemption sur p_l

Dans l'analyse d'ordonnançabilité sur p_l, le coût de la préemption d'une tâche est pris en compte dans le calcul de la durée restant à exécuter par cette tâche comme à la section 5.5.2.

$\forall t \in I^l(t) / t \geq r_i^1$, $c_i(t)$ est donnée par :

$$c_i(t) = \begin{cases} C_i & \text{si } (\frac{t-r_i^1}{T_i}) \in \mathbb{N} \text{ sinon} \\ c_i(r^{l-}(t)) & \text{si } (\phi^l(r^{l-}(t)) \neq \tau_i) \text{ sinon} \\ r^-(t) + c_i(r^{l-}(t)) - t & \text{si } (\phi^l(t) = \tau_i) \vee \\ & ((\phi^l(t) \neq \tau_i) \wedge (r^{l-}(t) + c_i(r^{l-}(t)) = t)) \text{ sinon} \\ (r^{l-}(t) + c_i(r^{l-}(t)) - t) + \alpha \end{cases}$$

Échéance relative aux dates d'appel sur p_l

À chaque appel de l'ordonnanceur, l'échéance relative d'une tâche par rapport à cette date d'appel est déterminée selon le même principe que dans la section 5.5.3.

$\forall t \in I^l(t) \geq r_i^1$, $d_i(t)$ est donnée par :

$$d_i(t) = \begin{cases} D_i & \text{si } (\frac{t-r_i^1}{T_i}) \in \mathbb{N} \text{ sinon} \\ r^-(t) + d_i(r^{l-}(t)) - t \text{ si } r^{l-}(t) + d_i(r^{l-}(t)) > t \text{ sinon} \\ 0 \end{cases}$$

Condition d'ordonnançabilité sur p_l

Dans cette section on présente la condition d'ordonnançabilité d'une tâche τ_i allouée au processeur p_l.

Théorème 6.5.1 *Soit t une date d'appel de l'ordonnanceur de p_l. Une tâche $\tau_i \in \Gamma_{n_l}$ est ordonnançable à t avec prise en compte du exact de la préemption si et seulement si :*

$$\begin{array}{l} (c_i(t) \leq d_i(t)) \wedge \\ ((t \leq r_i^1) \vee (c_i(r^{l-}(t)) = 0) \vee \\ (\phi(r^{l-}(t)) = \tau_i) \vee ((t - r_i^1) mod T_i \neq 0)) \end{array} \quad (6.6)$$

Preuve
La preuve de ce théorème est similaire à celle du théorème 5.5.1. ∎

Corollaire 6.5.1 *Une tâche τ_i allouée au processeur p_l est ordonnançable avec prise en compte du coût exact de la préemption et de communication inter-processeur si et seulement si elle est ordonnançable lors de chaque appel de l'ordonnanceur de p_l.*

Preuve
La preuve de ce corollaire est similaire à celui du corollaire 5.5.1. ∎

Corollaire 6.5.2 *L'ensemble des tâches Γ_{n_l} alloué à p_l avec des contraintes de dépendances de données est ordonnançable avec la prise en compte du coût exact de la préemption si et seulement si $\forall \tau_i \in \Gamma_{n_l}$, τ_i est ordonnançable selon le théorème 6.5.1.*

Preuve
La preuve de ce corollaire est similaire à celui du corollaire 5.5.2. ∎

Prochaine date d'appel de l'ordonnanceur du processeur p_l

L'ordonnanceur du processeur p_l est appelé aux dates d'activation et de fin d'exécution des tâches, et aux dates de réception de données. On note par F^l l'ensemble des dates d'activation, incluses dans I_n, des tâches allouées au processeur p_l. On note aussi par G^l l'ensemble des dates d'activation et de fin d'exécution des tâches sur p_l, et des dates de réception de données. On suppose que les éléments de F^l et ceux de G^l sont dans l'ordre croissant des dates. Avant l'exécution des tâches on initialise G^l à F^l. Nous allons mettre à jour G^l en ajoutant les dates de fin d'exécution des tâches et les dates de réception de données au fur et à mesure de l'analyse d'ordonnançabilité. Pour cela en partant de la plus petite des dates de première activation des tâches sur p_l qu'on note par r^l_{min}, on détermine la prochaine date d'appel de l'ordonnanceur de p_l qu'on note par r^{l+}. Si la tâche sélectionnée à r^l_{min} termine son exécution avant la prochaine date d'activation, quelle que soit la tâche, alors on note par f la date de fin d'exécution de la tâche sélectionnée. Sinon, on note par f la prochaine date d'activation. Ensuite on détermine l'ensemble H des dates de réception de données qui sont envoyées par les tâches allouées aux processeurs qui communiquent avec p_l. C'est-à-dire que, si une tâche τ_i allouée à un processeur p_k produit à la date r^l_{min} une donnée destinée à une tâche τ_j allouée au processeur p_l, cette donnée sera reçue sur p_l à la date $r^l_{min} + com_{kl}$, avec com_{kl} le pire temps de transfert d'une donnée, produite par τ_i, du processeur p_k au processeur p_l. La date $t + com_{kl}$ est une date de réception de donnée pour la tâche τ_j à la quelle l'ordonnanceur est appelé. Il peut y avoir d'autres tâches que τ_i, allouées à d'autres processeurs, qui envoient des données à τ_j. Ainsi τ_j peut recevoir les données de ses prédécesseurs à des dates différentes. La prochaine date d'appel de l'ordonnanceur r^{l+} est alors la plus petite date de l'ensemble $\{f\} \cup H$. On fait alors la mise à jour $G^l = G^l \cup \{f\} \cup H$. À la date r^{l+} on effectue le même traitement jusqu'à la fin de l'intervalle d'étude I_n.

Définition 6.5.3 *Soit t une date d'appel de l'ordonnanceur du processeur p_l qui ne peut être qu'une date d'activation, de fin d'exécution d'une tâche ou une date de réception de données. La prochaine date d'appel de l'ordonnanceur après t est donnée par la relation 6.7.*

$$r^{l+}(t) = min(t' \in \{f\} \cup H) \tag{6.7}$$

f et H sont respectivement déterminés par les relations 6.8 et 6.9.

$$f = \begin{cases} t + c_k(t) & \textit{si } (t + c_k(t)) < a(t) \textit{ sinon} \\ \\ a(t) \end{cases} \tag{6.8}$$

$$H = \begin{cases} \{\emptyset\} & \textit{si } t = r_{min} \textit{ sinon} \\ \\ \{t + com_{kl}\}_{((\tau_i \in pred(\tau_j)) \wedge ((\phi(r^{k-}(t)) = \tau_i) \wedge (r^{k-}(t) + c_i(r^{k-}(t)) = t)))} \end{cases} \tag{6.9}$$

avec τ_j allouée à p_l, $\tau_i \in pred(\tau_j)$ allouée au processeur p_k différent de p_l et les dates $r^{l-}(t)$ et $r^{k-}(t)$ sont respectivement les dernières dates d'appel des ordonnanceurs de p_l et de p_k.

6.5.3 Condition d'ordonnançabilité multiprocesseur

La condition d'ordonnançabilité multiprocesseur est donnée par le théorème 6.5.2.

Théorème 6.5.2 *Soit un ensemble Γ_n de tâches dépendantes et un ensemble Π_m de processeurs identiques.*

S'il existe, selon l'heuristique d'allocation donnée par l'algorithme 7, au moins une solution d'allocation S de l'ensemble des tâches Γ_n aux différents processeurs de Π_m et que chaque sous-ensemble Γ_{n_l} (avec $1 \leq l \leq m$) de Γ_n alloué au processeur p_l est ordonnançable sur ce processeur alors l'ensemble Γ_n est ordonnançable sur Π_m.

Preuve
S'il existe, selon l'heuristique d'allocation donnée par l'algorithme 7, au moins une solution d'allocation S de l'ensemble des tâches Γ_n aux différents processeurs de Π_m et que chaque sous-ensemble Γ_{n_l} (avec $1 \leq l \leq m$) de Γ_n alloué au processeur p_l est ordonnançable sur ce processeur alors chaque tâche de Γ_n respecte son échéance donc l'ensemble Γ_n est ordonnançable. ∎

6.5.4 Algorithme d'analyse d'ordonnançabilité multiprocesseur

L'algorithme 8 donne l'analyse d'ordonnançabilité de l'ensemble Γ_n des tâches dépendantes sur l'architecture multiprocesseur Π_m.

Avec l'algorithme 8 on parcourt l'intervalle d'étude I_n en se limitant aux dates d'appel de l'ordonnanceur des différents processeurs. Si l'ordonnanceur d'un processeur p_l est appelé à une date t, on vérifie s'il existe une tâche non ordonnançable sur ce processeur. Dès qu'il existe une tâche non ordonnançable sur un processeur p_l alors l'ensemble des tâches Γ_{n_l} sur ce processeur n'est pas ordonnançable, donc Γ_n n'est pas ordonnançable. Sinon si chaque ensemble de tâches Γ_{n_l} alloué à chaque processeur p_l est ordonnançable alors Γ_n est ordonnançable. Si

Algorithme 8 Analyse d'ordonnançabilité multiprocesseur

1: $t = r_{min}$
2: $ordonnançable \leftarrow vrai$
3: Initialiser $suivant$ à la prochaine date d'activation après t
4: **tant que** $(t < (tc + H_n)) \wedge (ordonnançable = vrai)$ **faire**
5: % Cette boucle permet de parcourir tous les processeurs afin de vérifier l'ordonnançabilité des tâches sur chaque processeur
6: **pour** $l = 1$ to m **faire**
7: % On vérifie si l'ordonnanceur du processeur p_l est appelé à t
8: **si** $(t \in G^l)$ **alors**
9: Calculer $\phi^l(t)$
10: $i \leftarrow 1$
11: % On vérifie si les n_l tâches allouées à p_l sont ordonnançables à t.
12: **tant que** $(i \leq n_l) \wedge (ordonnançable = vrai)$ **faire**
13: **si** $(t \geq r_i^1)$ **alors**
14: Calculer $c_i(t)$
15: Calculer $d_i(t)$
16: **si** $((c_i(t) > d_i(t)) \vee ((t > r_i^1) \wedge (c_i(r^{l-}(t)) > 0) \wedge (\phi(r^{l-}(t)) \neq \tau_i) \wedge ((t - r_i^1) mod T_i = 0))$ **alors**
17: $ordonnançable \leftarrow faux$
18: **fin si**
19: **fin si**
20: $i \leftarrow i + 1$
21: **fin tant que**
22: $suivant \leftarrow$ minimum$(suivant, r^{l+}(t))$
23: $G^l = G^l \cup \{f\} \cup H$
24: **fin si**
25: **fin pour**
26: $t \leftarrow suivant$
27: **fin tant que**

Γ_n est ordonnançable alors une table d'ordonnancement est produite pour chaque processeur p_l.

La complexité au pire cas de cette analyse d'ordonnançabilité est de $O(n.m.l)$, n est le nombre de tâches, m le nombre de processeurs et l est la longueur de l'intervalle d'étude I_n.

6.6 Heuristique d'ordonnancement multiprocesseur de tâches dépendantes

L'heuristique d'ordonnancement multiprocesseur qu'on propose prend en entrée un ensemble de tâches Γ_n et un ensemble d'architecture Π_m. Elle effectue l'allocation en utilisant l'heuristique d'allocation présentée par l'algorithme 7 puis analyse l'ordonnançabilité des tâches en utilisant l'algorithme 8. L'algorithme 9 donne les différentes étapes de cette heuristique.

Algorithme 9 Heuristique d'ordonnancement multiprocesseur

1: $ordonnançable \leftarrow vrai$
2: Allouer les tâches de Γ_n aux processeurs de Π_m en utilisant l'algorithme 7
3: **si** il existe une solution d'allocation S **alors**
4: Faire l'analyse d'ordonnançabilité de cette solution d'allocation S en utilisant l'algorithme 8
5: **si** cette solution S est ordonnançable selon l'algorithme 8 **alors**
6: L'ensemble Γ_n est ordonnançable et on a en sortie les tables d'ordonnancement de chaque processeur
7: **sinon**
8: L'ensemble Γ_n n'est pas ordonnançable selon notre heuristique d'ordonnancement
9: **fin si**
10: **sinon**
11: L'ensemble Γ_n n'est pas ordonnançable sur Π_m selon notre heuristique d'ordonnancement
12: **fin si**

La complexité au pire cas de cette heuristique d'allocation et d'ordonnancement est de $O(n.m + n.m.l) = O((l+1).n.m) = O(l.n.m)$, l est la longueur de l'intervalle d'étude. Cette complexité est la somme des complexités de l'heuristique d'allocation et de l'analyse d'ordonnançabilité.

6.7 Application

On considère l'ensemble Γ_8 de tâches dépendantes représenté par le graphe de dépendances de tâches G_8 donné par la figure 6.6. Soit Π_2 l'ensemble des processeurs sur lequel on veut ordonnancer les tâches dans G_8. Π_2 est représenté par le graphe de processeurs P_2 dans la figure 6.7.

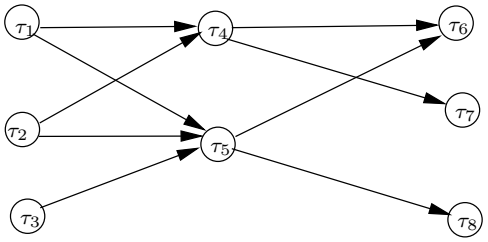

FIGURE 6.6 – Graphe G_8 de dépendances de données

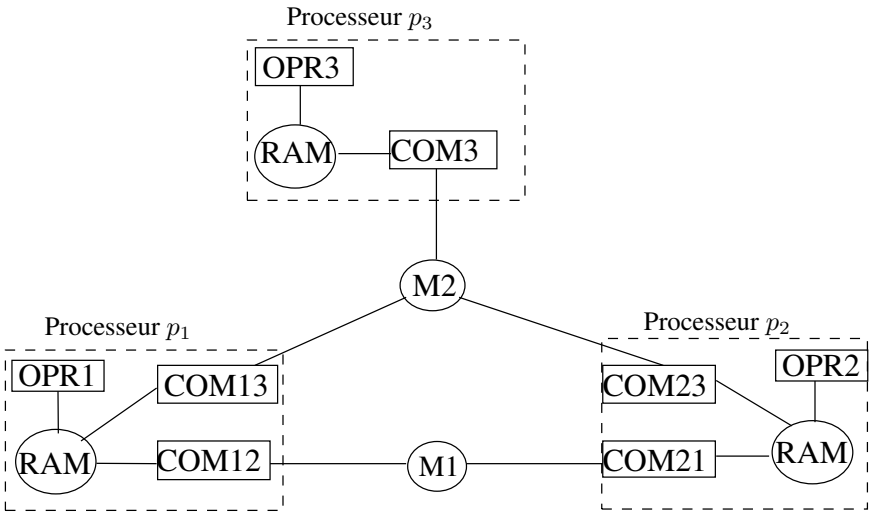

FIGURE 6.7 – Graphe d'architecture de P_2

Les tâches dans G_8 sont définies par : $\tau_1 = (1, 1, 6, 6)$, $\tau_2 = (1, 2, 12, 12)$, $\tau_3 = (0, 2, 24, 24)$, $\tau_4 = (8, 4, 12, 12)$, $\tau_5 = (2, 1, 6, 6)$, $\tau_6 = (7, 2, 24, 24)$, $\tau_7 = (4, 2, 12, 12)$ et $\tau_8 = (5, 3, 24, 24)$. On utilise le choix de priorités basé sur RM, c'est à dire que, la tâche la plus prioritaire est celle qui a la plus petite période. On suppose que le coût d'une préemption est donnée par $\alpha = 1$ unité de

temps. On suppose aussi que le temps de transfert d'une donnée, du processeur p_1 vers p_2 et réciproquement est fixe, $com_{12} = com_{21} = 1$ unité de temps.

L'application de notre heuristique d'ordonnancement multiprocesseur donnée par l'algorithme 9 implique d'utiliser l'heuristique d'allocation donnée par l'algorithme 7 pour faire d'abord l'allocation des tâches aux processeurs. En suivant les étapes de cette allocation nous obtenons :

1: Initialisation de l'ensemble des tâches candidates W à l'ensemble des tâches de G_8 qui n'ont pas de prédécesseurs. Nous obtenons $W = \{\tau_1, \tau_2, \tau_3\}$, les tâches de W sont rangées dans l'ordre décroissant de leurs priorités.

2: τ_1 est sélectionnée et allouée au processeur p_1. τ_1 pourrait aussi bien être allouée à p_2 car les processeurs sont identiques.

$$\begin{cases} p_1 \leftarrow \{\tau_1\} \\ p_2 \leftarrow \{\emptyset\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_2, \tau_4, \tau_3\}$

3: τ_2 est sélectionnée. Sur p_1 et p_2 nous avons $r_2^{(1,1)} = r_2^1$ et $D_2^1 = D_2$ car $pred(\tau_2) = \{\emptyset\}$. $C_2 = 2 \leq D_2^1 = 12$ donc on peut allouer τ_2 à p_1 ou à p_2. Le choix de l'allocation est déterminé par la marge d'exécution de τ_2 calculée sur chaque processeur. Sur p_1, nous avons $M_2^1 = (r_2^1 + D_2) - (r_2^{(1,1)} + (C_1 + C_2)) = (1 + 12) - (1 + (1 + 2)) = 13 - 4 = 9$. Et sur p_2 nous avons $M_2^2 = (r_2^1 + D_2) - (r_2^{(1,1)} + C_2) = (1+12) - (1+2) = 13 - 3 = 10$. Donc la marge d'exécution de τ_2 est plus grande sur p_2 que sur p_1 d'où τ_2 est allouée à p_2.

$$\begin{cases} p_1 \leftarrow \{\tau_1\} \\ p_2 \leftarrow \{\tau_2\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_4, \tau_3\}$.

4: τ_4 est sélectionnée. Sur p_1 nous avons $r_4^{(1,1)} = max(r_4^1, max((r_1^{(1,1)} + (k_{14} - 1) * T_1 + C_1), (r_2^{(1,2)} + (k_{24} - 1) * T_2 + C_2 + com_{24}))) = max(8, max((1 + (1 * 6) + 1), (1 + 0 + 2 + 1)) = 8$ et $D_4^1 = D_4 - (r_4^{(1,1)} - r_4^1) = 12 - (8 - 8) = 12$. $C_4 \leq D_4^1$ on peut alors calculer M_4^1. Ce qui donne $M_4^1 = (r_4^1 + D_4) - (r_4^{(1,1)} + C_4) = (8 + 12) - (8 + 4) = 20 - 12 = 8$. Sur p_2 nous avons $r_4^{(1,2)} = max(r_4^1, max((r_1^{(1,1)} + (k_{14} - 1) * T_1 + C_1 + com_{14}), (r_2^{(1,2)} + (k_{24} - 1) * T_2 + C_2))) = max(10, max((1 + (1 * 6) + 1 + 1), (1 + 0 + 2)) = 9$ et $D_4^2 = D_4 - (r_4^{(1,2)} - r_4^1) = 12 - (9 - 8) = 12 - 1 = 11$. $C_4 \leq D_4^2$ on peut alors calculer M_4^2. Ce qui donne $M_4^2 = (r_4^1 + D_4) - (r_4^{(1,2)} + C_4) = 20 - (9 + 4) = 20 - 13 = 7$. Donc la marge d'exécution de τ_4 est plus grande sur p_1 que sur p_2 d'où τ_4 est allouée au processeur p_1.

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4\} \\ p_2 \leftarrow \{\tau_2\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_3, \tau_6\}$.

5: τ_3 est sélectionnée. Sur p_1 et p_2 nous avons $r_3^{(1,1)} = r_3^{(1,2)} = r_3^1$ et $D_3^1 = D_3^2 = D_3$ car $pred(\tau_3) = \{\emptyset\}$. $C_3 = 2 \leq D_3^* = 24$ donc on peut allouer τ_3 à p_1 ou à p_2. Le choix de l'allocation est déterminé par la marge d'exécution de τ_2 sur chaque processeur. Sur p_1, nous avons $M_3^1 = (r_3^1 + D_3) - (r_3^{(1,1)} + (C_1 + C_4 + C_3)) = (0 + 24) - (0 + (1 + 4 + 2)) = 24 - 7 = 17$. Et sur p_2 nous avons $M_3^2 = (r_3^1 + D_3) - (r_3^{(1,2)} + (C_2 + C_3)) = (0+24) - (0+(2+2)) = 24-4 = 20$. Donc la marge d'exécution de τ_3 est plus grande sur p_2 que sur p_1 d'où τ_3 est allouée à p_2

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4\} \\ p_2 \leftarrow \{\tau_2, \tau_3\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_5, \tau_6\}$.

6: τ_5 est sélectionnée. Sur p_1 nous avons $r_5^{(1,1)} = max(r_5^1, max((r_1^{(1,1)} + (k_{15} - 1) * T_1 + C_1), (r_2^{(1,2)} + (k_{25} - 1) * T_2 + C_2 + com_{25}), (r_3^{(1,2)} + (k_{35} - 1) * T_3 + C_3 + com_{35})) = max(2, max((1+0+1), (1+0+2+1), (0+0+2+1))) = 4$ et $D_5^1 = (D_5 - (r_5^{(1,1)} - r_5^1)) = 6 - (4 - 1) = 3$. $C_5 \leq D_5^1$, on détermine la marge d'exécution de τ_5 sur p_1 : $M_5^1 = (r_5^1 + D_5) - (r_5^{(1,1)} + (C_5)) = (1+6) - (4+1) = 2$. Sur p_2 nous avons $r_5^{(1,2)} = max(r_5^1, max((r_1^{(1,1)} + (k_{15} - 1) * T_1 + C_1 + com_{15}), (r_2^{(1,2)} + (k_{25} - 1) * T_2 + C_2), (r_3^{(1,2)} + (k_{35} - 1) * T_3 + C_3)) = max(1, (1+0+1+1), (1+0+2), (0+0+2)) = 3$ et $D_5^2 = (D_5 - (r_5^{(1,1)} - r_5^1)) = 6 - (3 - 1) = 4$. $C_5 \leq D_5^2$, on détermine la marge d'exécution de τ_5 sur p_2 : $M_5^1 = (r_5^1 + D_5) - (r_5^{(1,2)} + (C_5)) = (1 + 6) - (3 + 1) = 3$. Donc la marge d'exécution de τ_5 est plus grande sur p_2 que sur p_1 d'où τ_5 est allouée à p_2.

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4\} \\ p_2 \leftarrow \{\tau_5, \tau_2, \tau_3\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_7, \tau_6, \tau_8\}$.

7: τ_7 est sélectionnée. Sur p_1 nous avons $r_7^{(1,1)} = max(r_7^1, (r_4^{(1,1)} + (k_{47} - 1) * T_4 + C_4)) = max(4, max((8 + 0 + 4)) = 12$ et $D_7^1 = (D_7 - (r_7^{(1,1)} - r_7^1)) = 12 - (12 - 4) = 12 - 8 = 4$. $C_7 \leq D_7^1$, on détermine la marge d'exécution de τ_5 sur p_1 : $M_7^1 = (r_7^1 + D_7) - (r_7^{(1,1)} + (C_7)) = (4+12) - (12+2) = 16 - 14 = 2$. Sur p_2 nous avons $r_7^{(1,2)} = max(r_7^1, (r_4^{(1,1)} + (k_{47} - 1) * T_4 + C_4 + com_{47})) = max(4, (8+0+4+1)) = 13$ et $D_7^2 = (D_7 - (r_7^{(1,1)} - r_7^1)) = 12 - (13 - 4) = 3$. $C_7 \leq D_7^2$, on détermine la marge d'exécution de τ_7 sur p_2 : $M_7^1 = (r_7^1 + D_7) -$

$(r_7^{(1,2)}+(C_7)) = (4+12)-(13+2) = 16-15 = 1$. Donc la marge d'exécution de τ_7 est plus grande sur p_1 que sur p_2 d'où τ_7 est allouée à p_1.

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4, \tau_7\} \\ p_2 \leftarrow \{\tau_5, \tau_2, \tau_3\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_6, \tau_8\}$.

8: τ_6 est sélectionnée. Sur p_1 nous avons $r_6^{(1,1)} = max(r_6^1, max((r_4^{(1,1)} + (k_{46} - 1) * T_4 + C_4), (r_5^{(1,2)} + (k_{56} - 1) * T_5 + C_5 + com_{56}))) = max(7, max((8 + (12 * 1) + 4), (3 + (1 * 6) + 1 + 1))) = 24$ et $D_6^1 = D_6 - (r_6^{(1,1)} - r_6^1) = 24 - (24 - 7) = 24 - 17 = 7$. $C_6 \leq D_6^1$ on peut alors calculer M_6^1. Ce qui donne $M_6^1 = (r_6^1 + D_6) - (r_6^{(1,1)} + C_6) = (7+24) - (24+2) = 31 - 26 = 5$. Sur p_2 nous avons $r_6^{(1,2)} = max(r_6^1, max((r_4^{(1,1)} + (k_{46} - 1) * T_4 + C_4 + com_{46}), (r_5^{(1,2)} + (k_{56} - 1) * T_5 + C_5))) = max(7, max((8 + (12 * 1) + 4 + 1), (3 + (1 * 6) + 1 + 1))) = 25$ et $D_6^2 = D_6 - (r_6^{(1,2)} - r_6^1) = 24 - (25 - 7) = 24 - 18 = 6$. $C_6 \leq D_6^2$ on peut alors calculer M_6^2. Ce qui donne $M_6^2 = (r_6^1 + D_6) - (r_6^{(1,2)} + C_6) = (7 + 24) - (25 + 2) = 31 - 27 = 4$. Donc la marge d'exécution de τ_6 est plus grande sur p_1 que sur p_2 d'où τ_6 est allouée au processeur p_1.

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4, \tau_7, \tau_6\} \\ p_2 \leftarrow \{\tau_5, \tau_2, \tau_3\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\tau_8\}$.

9: τ_8 est sélectionnée. Sur p_1 nous avons $r_8^{(1,1)} = max(r_8^1, (r_5^{(1,2)} + (k_{58} - 1) * T_5 + C_5 + com_{58})) = max(5, (3 + (3 * 6) + 2 + 1)) = 24$ et $D_8^1 = D_8 - (r_8^{(1,1)} - r_8^1) = 24 - (24 - 5) = 24 - 19 = 5$. $C_8 \leq D_8^1$ on peut alors calculer M_8^1. Ce qui donne $M_8^1 = (r_8^1 + D_8) - (r_8^{(1,1)} + C_8) = (5+24) - (24+3) = 29 - 27 = 2$. Sur p_2 nous avons $r_8^{(1,2)} = max(r_8^1, (r_5^{(1,1)} + (k_{58} - 1) * T_5 + C_5) = max(5, (3 + (3 * 6) + 2)) = 23$ et $D_8^2 = D_8 - (r_8^{(1,2)} - r_8^1) = 24 - (23 - 5) = 24 - 18 = 6$. $C_8 \leq D_8^2$ on peut alors calculer M_8^2. Ce qui donne $M_8^2 = (r_8^1 + D_8) - (r_8^{(1,2)} + C_8) = (5 + 24) - (23 + 3) = 29 - 26 = 3$. Donc la marge d'exécution de τ_8 est plus grande sur p_2 que sur p_1 d'où τ_8 est allouée au processeur p_2.

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4, \tau_7, \tau_6\} \\ p_2 \leftarrow \{\tau_5, \tau_2, \tau_3, \tau_8\} \end{cases}$$

Après mise à jour de W nous avons $W = \{\emptyset\}$.

10: $W = \{\emptyset\}$, donc toutes les tâches sont allouées et le résultat de l'allocation est :

$$\begin{cases} p_1 \leftarrow \{\tau_1, \tau_4, \tau_7, \tau_6\} \\ p_2 \leftarrow \{\tau_5, \tau_2, \tau_3, \tau_8\} \end{cases}$$

L'allocation des tâches de Γ_8 aux processeurs de Π_2 a réussi.

Après cette étape d'allocation, nous utilisons l'algorithme 8 pour ordonnancer les tâches. L'intervalle d'étude de l'ordonnancement de Γ_8 est donné par $I_8 = [r_{min}+r_{max}+2*H_8]$ avec $r_{min} = 0, r_{max} = 8, H_8 = ppcm(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8) = 24$. Donc nous avons $I_8 = [0, 8 + 2 * 24] = 56$. Selon l'algorithme 8, le système de tâches Γ_8 est ordonnançable et les tables d'ordonnancement respectives des tâches allouées à p_1 et à p_2 sont données respectivement par les tableaux 6.1 et 6.2. La table d'ordonnancement du processeur de communication $pcom12$ est donnée par le tableau 6.3. Dans cette table d'ordonnancement de $pcom12$, par exemple, $com(p_1, \tau_1, p_2, \tau_5)$ est le transfert d'une donnée produite par τ_1 sur le processeur p_1 et reçue par τ_5 sur le processeur p_2. com représente la durée qui détermine l'intervalle de temps pour réaliser une communication.

En utilisant ces tables d'ordonnancement, on trace le diagramme de Gantt d'ordonnancement des tâches donné par la figure 6.8.

Dans cette figure on observe qu'aux dates $t = 13, 25, 37$ et 49, la tâche τ_7 est préemptée par τ_1. À chacune de ces préemptions on prend en compte le coût de la préemption en ajoutant une unité de temps supplémentaire, correspondant au coût d'une préemption ($\alpha = 1$), sur la durée restant à exécuter par τ_7. On observe aussi que chaque tâche reçoit toutes les données de ses prédécesseurs avant de commencer à s'exécuter. Par exemple la tâche τ_6 attend ses données, produites par les tâches τ_4 et τ_5, de la date 7 à la date 28. Il y a aussi des blocages, mais ces temps de blocage sont de durées faibles. Par exemple sur p_2, la tâche τ_2 est bloquée pendant 2 unités de temps par la tâche τ_8 à partir de la date $t = 49$ jusqu'à la date $t = 51$. Ce temps de blocage, très faible, est le plus grand des temps de blocage parmi ceux qui se produisent pendant l'intervalle d'étude.

6.8 Étude de performance

Dans cette section on étudie les performances de l'heuristique d'ordonnancement multiprocesseur que nous avons proposée pour les tâches dépendantes et présentée dans l'algorithme 9. Pour cela nous avons comparé notre heuristique avec l'algorithme exact Branch and Bound ($B\&B$) donné par l'algorithme 10 puisqu'on ne dispose pas se "benchmark" sur la prise en compte du coût exact de la préemption. Cet algorithme exact énumère toutes les solutions d'allocation possibles puis parcourt toutes ces solutions afin de déterminer la solution ordonnançable selon l'algorithme 8 et qui minimise le makespan des tâches. En ce qui concerne les conditions techniques de test, l'ordinateur utilisé est un DELL muni d'un processeur $IntelCore^{TM}$ 2 Duo (2,66 GHz, mémoire RAM 4Go). Avant

t	$\phi^1(t) = \tau_i$	$c_i(t)$
1	τ_1	1
2	$idle$	2
4	$idle$	1
5	$idle$	1
6	$idle$	1
7	τ_1	1
8	τ_4	4
11	τ_4	1
12	τ_7	2
13	τ_1	1
14	τ_7	2
16	$idle$	1
17	$idle$	2
19	τ_1	1
20	τ_4	4
23	τ_4	1
24	τ_7	2
25	τ_1	1
26	τ_7	2
28	τ_6	2
30	$idle$	1
31	τ_1	1
32	τ_4	4
35	τ_4	1
36	τ_7	2
37	τ_1	1
38	τ_7	2
40	$idle$	1
41	$idle$	2
43	τ_1	1
44	τ_4	4
47	τ_4	1
48	τ_7	2
49	τ_1	1
50	τ_7	2
52	τ_6	2
54	$idle$	1
55	τ_1	1

TABLE 6.1 – Table d'ordonnancement du processeur p_1

t	$\phi^2(t) = \tau_i$	$c_i(t)$
0	τ_3	2
1	τ_3	1
2	τ_2	2
3	τ_2	1
4	τ_5	1
5	$idle$	3
8	$idle$	1
9	τ_5	1
10	$idle$	3
13	τ_2	2
14	τ_2	1
15	τ_5	1
16	$idle$	4
20	$idle$	1
21	τ_5	1
22	τ_8	3
24	τ_8	1
25	τ_2	2
26	τ_2	1
27	τ_3	2
29	$idle$	1
30	τ_5	1
31	$idle$	1
32	$idle$	1
33	$idle$	1
34	$idle$	3
37	τ_2	2
38	τ_2	1
39	τ_5	1
40	$idle$	4
44	$idle$	1
45	τ_5	1
46	τ_8	3
49	τ_2	2
51	τ_3	2
53	τ_5	1
54	$idle$	2

TABLE 6.2 – Table d'ordonnancement du processeur p_2

t	$communications$	com
2	$com(p_1, \tau_1, p_2, \tau_5)$	1
4	$com(p_2, \tau_2, p_1, \tau_4)$	1
5	$com(p_2, \tau_5, p_1, \tau_6)$	1
8	$com(p_1, \tau_1, p_2, \tau_5)$	1
10	$com(p_2, \tau_5, p_1, \tau_6)$	1
14	$com(p_1, \tau_1, p_2, \tau_5)$	1
15	$com(p_2, \tau_2, p_1, \tau_4)$	1
20	$com(p_1, \tau_1, p_2, \tau_5)$	1
22	$com(p_2, \tau_5, p_1, \tau_6)$	1
26	$com(p_1, \tau_1, p_2, \tau_5)$	1
27	$com(p_2, \tau_2, p_1, \tau_4)$	1
32	$com(p_1, \tau_1, p_2, \tau_5)$	1
34	$com(p_2, \tau_5, p_1, \tau_6)$	1
38	$com(p_1, \tau_1, p_2, \tau_5)$	1
39	$com(p_2, \tau_2, p_1, \tau_4)$	1
40	$com(p_2, \tau_5, p_1, \tau_6)$	1
44	$com(p_1, \tau_1, p_2, \tau_5)$	1
46	$com(p_2, \tau_5, p_1, \tau_6)$	1
51	$com(p_2, \tau_2, p_1, \tau_4)$	1
54	$com(p_2, \tau_5, p_1, \tau_6)$	1

TABLE 6.3 – Table d'ordonnancement du processeur de communication $pcom12$

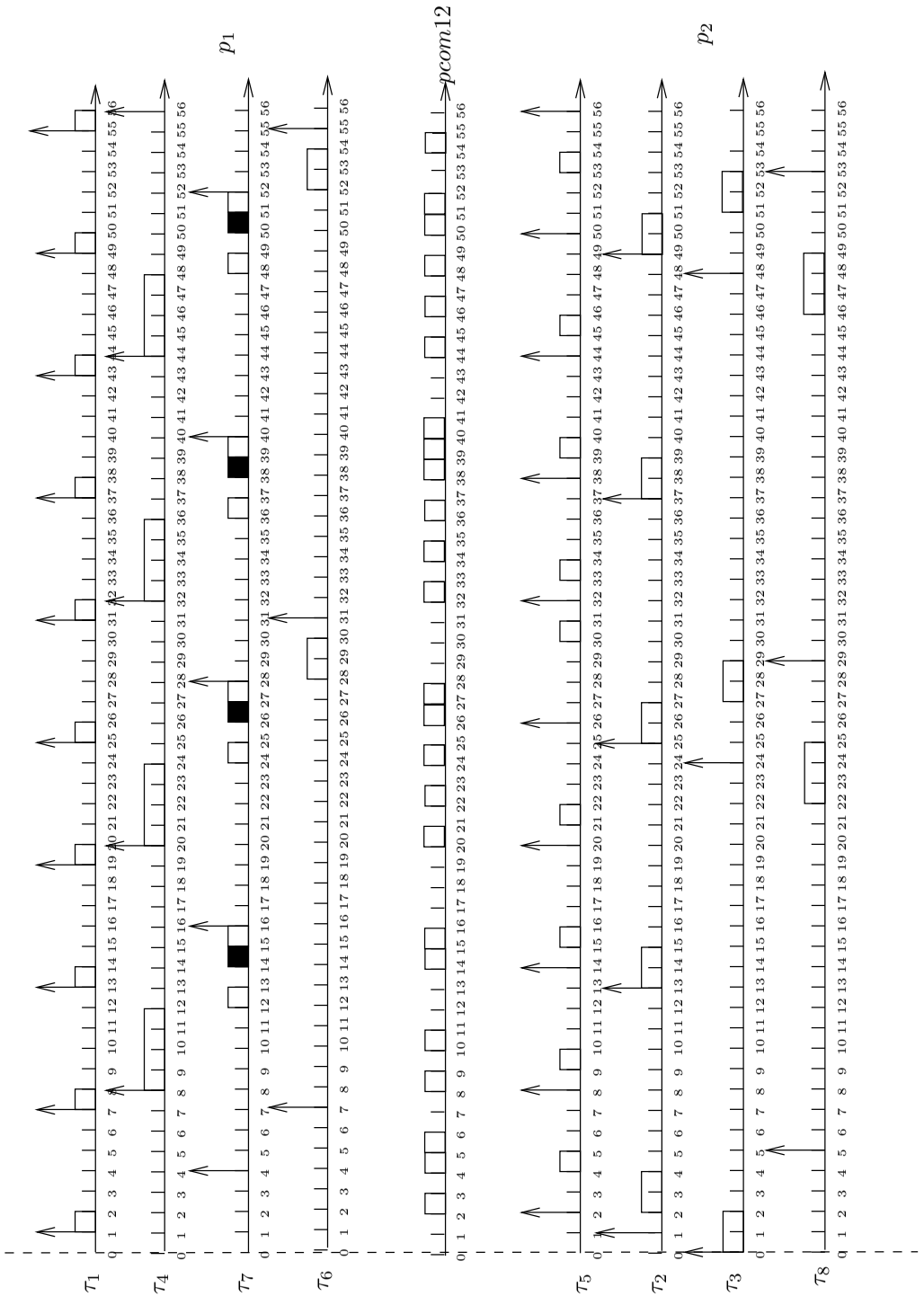

FIGURE 6.8 – Résultat de l'ordonnancement des tâches sur p_1 et sur p_2

de présenter les résultats de cette comparaison nous présentons le générateur de graphes de tâches que nous avons implémenté pour générer des tâches dépendantes.

6.8.1 Générateur de graphes de tâches dépendantes

Le générateur de tâches que nous avons implémenté est basé sur le principe des travaux présentés dans [Kal04, HGRC96]. On génère des graphes de tâches constitués de plusieurs niveaux ou rangs (au minimum 2 niveaux). Le niveau d'une tâche est la longueur du plus long chemin qui a pour extrémité cette tâche en partant d'une tâche sans prédécesseur. Chaque niveau contient une ou plusieurs tâches et une tâche d'un niveau $i > 1$ a au moins un prédécesseur au niveau $i - 1$. Par rapport au générateur présenté dans [Kal04] notre générateur de tâches dépendantes permet de spécifier le nombre de tâches d'entrées qui représentent les capteurs et le nombre de tâches de sorties qui représentent les actionneurs.

Les paramètres du générateur de graphe de tâches sont :

- la taille du graphe notée n qui correspond au nombre total de tâches dans le graphe,
- la longueur du graphe notée l qui correspond au nombre de niveaux dans le graphe,
- la hauteur du graphe notée h qui correspond au nombre maximum de tâches dans un niveau,
- le nombre de tâches d'entrées ou capteurs noté n_{in},
- le nombre de tâches de sorties ou actionneurs noté n_{out}.

Les valeurs de ces paramètres doivent être cohérentes pour qu'on puisse générer un graphe de tâches correspondant au type de graphe spécifié par un utilisateur. Pour cela il y a un certain nombre de conditions que ces paramètres doivent vérifier. Ce sont :

- $n_{in} \leq h$,
- $n_{out} \leq h$,
- $(l + h - 1) \leq n \leq (l * h - (2 * h - n_{in} - n_{out}))$.

Les deux premières inéquation assurent que le nombre maximum de tâches dans un niveau ne dépasse pas h. L'inéquation $(l + h - 1) \leq n$ assure que la taille du graphe soit supérieure à $(l + h - 1)$ car en créant un graphe de longueur l et de hauteur h il y a au minimum $(l + h - 1)$ tâches dans ce graphe. Et enfin l'inéquation $n \leq (l * h - (2 * h - n_{in} - n_{out}))$ permet d'assurer d'avoir exactement n_{in} tâches d'entrées et n_{out} tâches de sortie. Si l'utilisateur n'indique que la taille du graphe, les paramètres l, h, n_{in} et n_{out} sont générés aléatoirement.

Algorithme 10 Algorithme exact de $B\&B$ de tâches dépendantes

1: Initialiser l'ensemble W à l'ensemble des tâches sans prédécesseurs dans G_n
2: Initialiser la variable $liste\_solutions$ à l'ensemble vide
3: Initialiser la variable $allocation\_réussie$ à $vrai$
4: **tant que** ($W \neq \emptyset$) et ($allocation\_réussie = vrai$) **faire**
5: Sélectionner la tâche τ_i la plus prioritaire dans W
6: **si** $liste\_solutions = \{\emptyset\}$ **alors**
7: Créer la solution S_1 composé d'un seul processeur avec τ_i allouée à ce processeur
8: Ajouter S_1 à la liste des solutions $liste\_solutions$
9: **sinon**
10: $l\_sol \leftarrow \{\emptyset\}$
11: **pour** $k = 1$ à $nb\_solutions$ (nombre de solution de $liste\_solutions$) **faire**
12: $S_k \leftarrow$ solution k de $liste\_solutions$
13: **pour** $j = 1$ à $nb\_proc\_s_k$ ($nb\_proc\_s_k$ est le nombre de processeurs dans S_k) **faire**
14: **si** $C_i \leq D_i^k$ **alors**
15: Créer la solution $S_{(k,j)}$ qui est la solution extraite de S_k en allouant la tâche τ_i au processeur p_j
16: Ajouter $S_{(k,j)}$ à la liste des solutions l_sol
17: **fin si**
18: **fin pour**
19: **si** $nb\_proc\_s_k < m$ **alors**
20: Créer la solution $S_{(k,j+1)}$ qui est la solution extraite de S_k en ajoutant un $j+1$ ème processeur et en allouant la tâche τ_i au processeur p_{j+1}
21: Ajouter $S_{(k,j+1)}$ à la liste des solutions l_sol
22: **fin si**
23: **fin pour**
24: **si** $l\_sol = \emptyset$ **alors**
25: $allocation\_réussie = faux$
26: **sinon**
27: $liste\_solutions \leftarrow l\_sol$
28: **fin si**
29: **fin si**
30: Supprimer τ_i de l'ensemble W
31: Ajouter dans W toutes les successeurs de τ_i qui n'ont pas de prédécesseurs dans W
32: **fin tant que**
33: **si** $allocation\_réussie = vrai$ **alors**
34: Déterminer dans la liste des solutions $liste\_solutions$ et en utilisant l'algorithme 11, la meilleure solution S_k qui est ordonnançable
35: **sinon**
36: L'ensemble Γ_n n'est pas ordonnançable
37: **fin si**

Algorithme 11 Meilleure solution de la liste des solutions $liste\_solutions$

$ordonnançable \leftarrow faux$
si $liste\_solutions \neq \{\emptyset\}$ **alors**
  $meilleure\_solution \leftarrow$ la solution S_1 de $liste\_solutions$
  $meilleure\_makespane \leftarrow$ le makespan des tâches avec la solution S_1
  pour $k = 2$ à $nb\_solutions$ **faire**
    Faire l'analyse d'ordonnançabilité de la solution S_k en utilisant l'algorithme 9
    Calculer L_k, le makespan des tâches avec la solution S_k
    si (S_k est ordonnançable) et ($L_k < meilleure\_makespan$) **alors**
      $meilleure\_solution \leftarrow S_k$
      $meilleure\_makespan \leftarrow L_k$
      $ordonnançable \leftarrow vrai$
    fin si
  fin pour
  si $ordonnançable = vrai$ **alors**
    L'ensemble Γ_n est ordonnançable et $meilleure\_solution$ est la meilleure solution qui minimise le makespan des tâches
  sinon
    L'ensemble Γ_n n'est pas ordonnançable
  fin si
sinon
  L'ensemble Γ_n n'est pas ordonnançable
fin si

La génération des tâches se fait en deux étapes. Dans un premier temps, dans une matrice de taille $l * h$ on génère aléatoirement les n sommets du graphes. Pour cela on génère d'abord au premier niveau n_{in} sommets et n_{out} sommets pour le dernier niveau l du graphe. Ensuite pour chaque niveau $i/1 \leq i \leq l$ on génère k sommets ($1 \leq k \leq h$) en fonction du nombre de sommets du graphe s qu'il reste à générer. Dans un second temps en commençant au premier niveau du graphe, pour chaque sommet on génère les caractéristiques temporelles (r^1,C,T et D) pour ce sommet qui devient une tâche. Après cela on détermine aléatoirement pour cette tâche un ensemble de prédécesseurs parmi les tâches du niveau inférieur. Pour générer les paramètres des tâches on applique l'algorithme uunifast de Bini [BB05] qui détermine d'abord un facteur d'utilisation u_i pour chaque tâche τ_i de telle sorte que la somme des facteurs d'utilisation de toutes les tâches soit égale au facteur d'utilisation qu'on donne en entrée. Ensuite en fixant une période T_i pour cette tâche τ_i, on détermine la durée d'exécution C_i donnée par $C_i = u_i * T_i$. La période d'une tâche est générée de telle sorte qu'elle soit multiple des périodes de ses tâches prédécesseurs.

6.8.2 Graphe de processeurs

Pour simplifier les tests, on suppose que le graphe de processeurs est complètement connecté et que chaque processeur est relié à un autre processeur par une une liaison $point$-à-$point$. Les processeurs sont identiques et communiquent par passage de messages. On suppose aussi que le coût de communication entre deux processeur est fixe et égal à une unité de temps.

6.8.3 Résultats

Nous avons comparé notre heuristique multiprocesseur et l'algorithme exact $B\&B$ par rapport à leurs temps d'exécution en faisant varier le nombre de tâches puis le nombre de processeurs. Puis nous les avons comparé par rapport à leurs taux de succès et par rapport au makespan des tâches.

6.8.3.1 Temps d'exécutions des algorithmes

Pour comparer notre heuristique et l'algorithme exact $B\&B$ par rapport à leurs temps d'exécution nous avons dans un premier temps fixé le nombre de processeurs en considérant un seul graphe de processeurs composé de 3 processeurs et on a fait varier le nombre de tâches. Pour cela nous avons généré avec notre générateur de graphes de tâches 10 graphes de tâches. Le nombre de tâches dans les graphes de tâches est compris entre 6 et 100 tâches. On exécute notre heuristique et l'algorithme exact $B\&B$ sur chaque couple (un graphe de tâches et le graphe

de 3 processeurs considéré) puis on calcule le temps d'exécution des algorithmes pour tracer la courbes des temps d'exécution donnée par la figure 6.9 avec en abscisse le nombre de tâches dans chaque graphe de tâches.

Dans un second temps nous avons considéré un seul graphe de tâches composé de 10 tâches et deux graphes de processeurs composés respectivement de 2 et de 3 processeurs chacun. On a exécuté notre heuristique et l'algorithme exact $B\&B$ sur chaque couple (le graphe de 10 tâches, un graphe de processeurs) puis on a calculé le temps d'exécution de l'heuristique et de l'algorithme exact. Nous avons considéré ce petit nombre de tâches et de processeurs à cause de la complexité de l'algorithme exact $B\&B$. L'objectif étant juste de donner une évolution du temps d'exécution de notre heuristique et de l'algorithme exact lorsque le nombre de processeur croît. Ensuite en considérant uniquement notre heuristique nous avons considéré un graphe de tâches composé de 100 tâches et 13 graphes de processeurs différents avec un nombre de processeurs variant entre 4 et 15 processeurs. On a exécuté notre heuristique sur chaque couple (le graphe de 100 tâches, un graphe de processeurs). Nous avons tracer la courbe des temps d'exécution donnée par la figure 6.10 avec en abscisse le nombre de processeurs et en ordonnée le temps d'exécution des algorithmes.

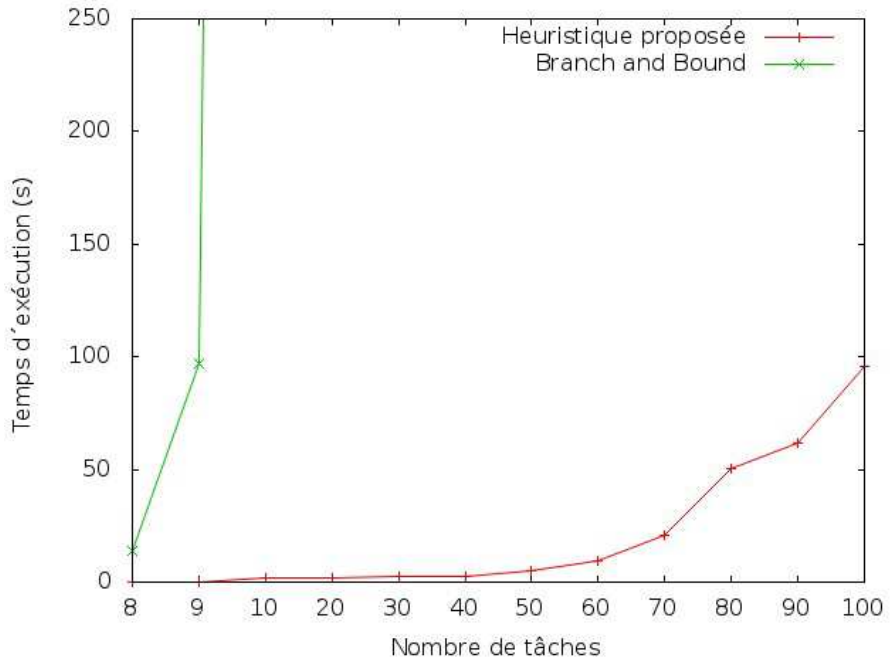

FIGURE 6.9 – Comparaison des durées d'exécution des algorithmes

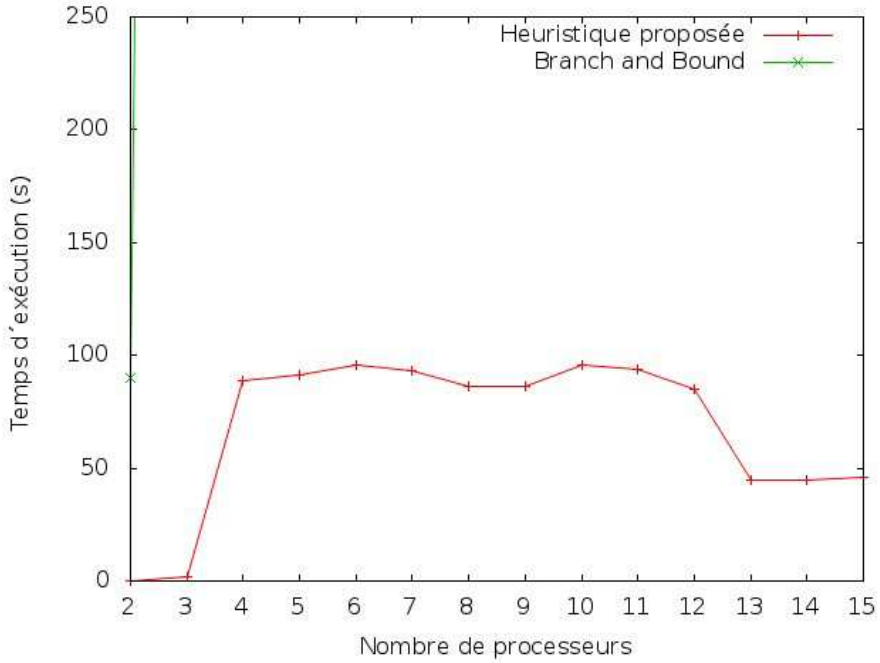

FIGURE 6.10 – Comparaison des durées d'exécution des algorithmes

Dans les figues 6.9 et 6.10 on observe qu'à partir de 10 tâches et 3 processeurs le temps d'exécution de l'algorithme exact $B\&B$ commence à exploser par contre notre heuristique a des temps d'exécution raisonnables. Dans la figure 6.10 on observe aussi que le temps d'exécution de notre heuristique ne croît pas de façon monotone en fonction du nombre de processeurs. Ce temps d'exécution a tendance à décroître si on augmente le nombre de processeurs. Ceci peut s'expliquer par le fait que plus le nombre de processeurs augmente plus les tâches sont réparties sur tous les processeurs ce qui diminue sur un processeur l'ensemble des dates d'appel de son ordonnanceur qui est fonction, entre autre, des dates d'activation des tâches sur ce processeurs. Ceci diminue la durée de l'analyse d'ordonnançabilité multiprocesseur et donc améliore la durée d'exécution de notre heuristique.

6.8.3.2 Taux de succès

Pour comparer notre heuristique et l'algorithme exact $B\&B$ par rapport à leurs taux de succès nous avons généré un seul graphe de processeurs composé de $m = 3$. Ensuite nous avons généré 10 ensembles dont chaque ensemble est composé de 10 graphes de tâches. À cause de la complexité de l'algorithme $B\&B$ au delà de 10 tâches, nous avons limité le nombre de tâches de chaque graphe

de tâches à 10 tâches. Pour chaque ensemble de graphes de tâches on exécute notre heuristique et l'algorithme exact $B\&B$ puis on a calculé le taux de succès de chaque algorithmes. Avec ce résultat on trace la courbe des taux de succès en mettant en abscisse la moyenne des facteurs d'utilisation des ensembles de graphes de tâches et en ordonnée les taux de succès. La figure 6.11 donne le résultat obtenu.

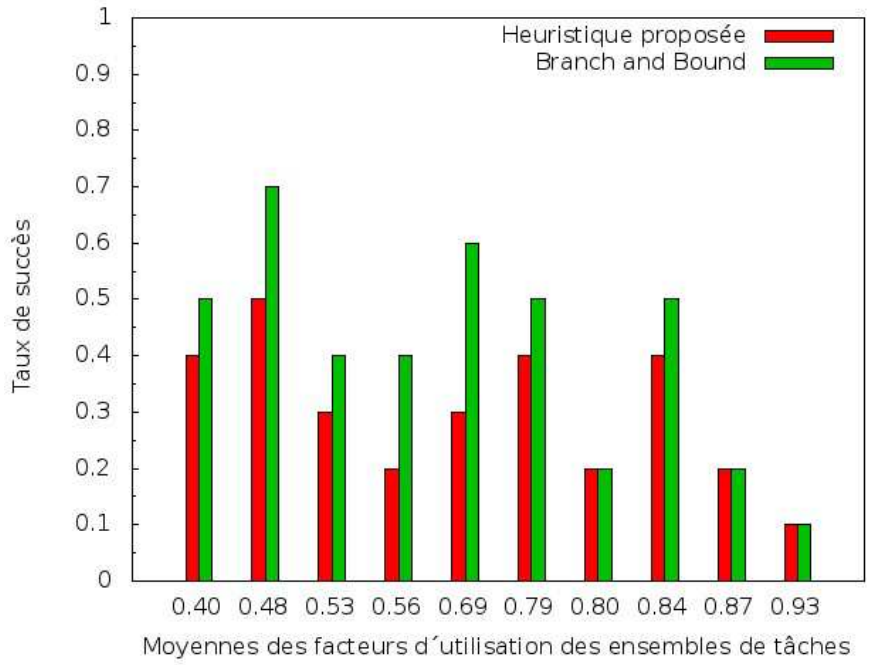

FIGURE 6.11 – Comparaison des taux de succès des algorithmes

Dans la figure 6.11, même si l'algorithme exact $B\&B$ donne des taux de succès plus importants que notre algorithme, ce qui était prévisible puisque l'algorithme $B\&B$ énumère toutes les solutions possibles, on remarque que notre algorithme n'est pas très éloigné, en terme de taux de succès.

6.8.3.3 Makespan

Pour comparer notre heuristique et l'algorithme exact $B\&B$ par rapport au makespan des tâches, nous avons considéré 8 graphes de tâches dont chaque graphe de tâches est composé de 10 tâches. On exécute notre heuristique et l'algorithme exact $B\&B$ sur chaque graphe de tâches et un graphe de processeurs composé de 3 processeurs. Nous avons considéré que des graphes de tâches ordonnançables sur ce graphe de processeurs puis on calcule le pire makespan des tâches avec

l'ordonnancement produit par chaque algorithme pour la courbe des makespan donnée à la figure 6.12.

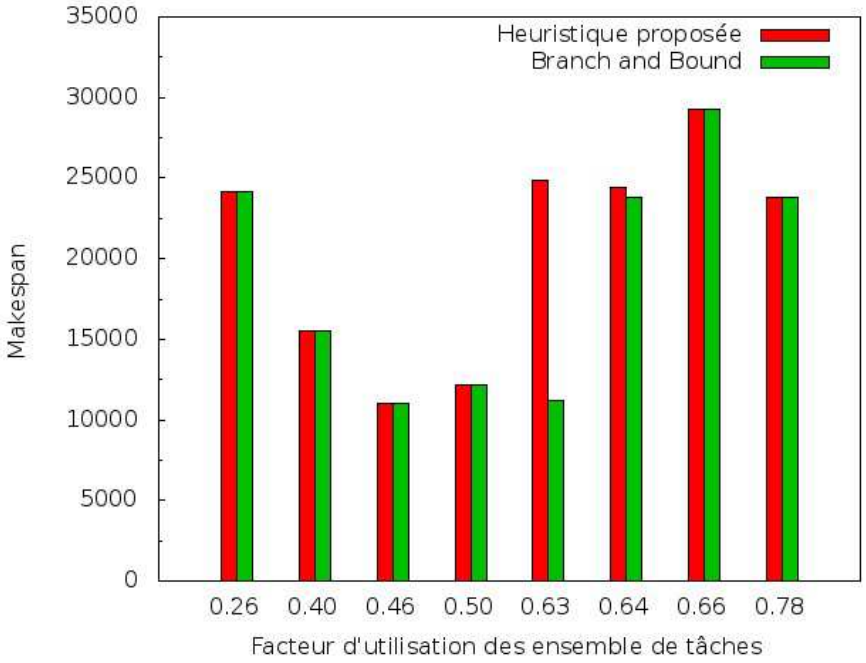

FIGURE 6.12 – Comparaison des makespan

Dans cette figure, on observe que notre heuristique même si elle ne parcourt pas toutes les solution afin de produire un ordonnancement qui donne le plus petit makespan, a des makespans relativement égal que ceux de l'algorithme exact de $B\&B$. Ceci à cause de la maximisation de la marge d'exécution qu'on effectue lors de l'allocation des tâches.

6.9 Conclusion

Dans ce chapitre nous avons proposé une heuristique d'ordonnancement multiprocesseur de tâches dépendantes avec la prise en compte du coût exact de la préemption. Pour allouer les tâches aux processeurs, on a utilisé une heuristique d'allocation de type liste qui maximise la marge d'exécution des tâches ce qui lui permet de minimiser le maksepan des tâches lors de l'exécution. Ensuite après l'étape d'allocation des tâches aux processeurs, on a appliqué sur chaque processeurs une extension de l'analyse d'ordonnançabilité hors ligne présentée au

chapitre 5 afin d'ordonnancer les tâches allouées à ce processeur. Après avoir ordonnancé les tâches sur les différents processeurs, notre heuristique produit une table d'ordonnancement pour chaque processeur que l'on utilisera pour implanter l'ordonnanceur (chapitre suivant). Nous avons aussi étudié les performances de notre heuristique en la comparant à un algorithme exact $B\&B$. On peut noter que si l'écart, en faveur de l'algorithme $B\&B$, entre les taux de succès et les makespans des deux algorithmes est faible, alors l'écart, en faveur de notre heuristique, entre les temps d'exécution est important.

Chapitre 7

Ordonnanceur en ligne avec prise en compte du coût exact de la préemption

7.1 Introduction

Dans ce chapitre nous présentons pour un processeur p_j, l'ordonnanceur en ligne utilisant la table d'ordonnancement produite après l'analyse d'ordonnançabilité hors ligne. Lors de cette analyse (voir chapitre 6) qui fait appel à un ordonnanceur hors ligne, nous avons pris en compte le coût exact de la préemption et effectué la synchronisation de l'accès aux mémoires de données, tout ceci en garantissant la viabilité de cette analyse d'ordonnançabilité. Dans la suite du chapitre, pour simplifier, on appellera cet ordonnanceur en ligne, ordonnanceur. Comme nous avons choisi l'approche d'ordonnancement multiprocesseur partitionné, cet ordonnanceur en ligne sera utilisé sur chaque processeur. Dans ce chapitre on présente aussi la gestion des communications inter-processeurs.

7.2 Principe de l'ordonnanceur

Nous cherchons à implanter un ordonnanceur en ligne en utilisant la table d'ordonnancement produite après l'analyse d'ordonnançabilité hors ligne, pour cela nous avons choisi un ordonnanceur dirigé par le temps (Time-Trigger).

Dans la classe des ordonnanceurs dirigés par le temps, les plus connus sont ceux qui sont invoqués périodiquement [Kop98, Pon01]. Une horloge périodique est utilisée pour invoquer l'ordonnanceur. L'inconvénient de cette invocation périodique, est que pour déterminer les activations des tâches, la période d'invocation de l'ordonnanceur doit être bien choisie de façon à ce que les instants pério-

diques de l'horloge correspondent à des dates d'activations des tâches. Dans le cas contraire, on risque de ne pas prendre en compte certaines activations de tâches. Par exemple si on a trois tâches à premières dates d'activations synchrones, de périodes 10, 14 et 30 unités de temps, il faut que la période de l'horloge soit le PGCD (Plus Grand Commun Diviseur) des trois périodes, égal à 2 unités de temps. De plus cette approche peut conduire à invoquer l'ordonnanceur plus que nécessaire. Dans l'exemple précédent si on considère que les tâches ont la même durée d'exécution égale à 1 unité de temps, alors l'exécution des trois premières instances de ces tâches se termine au bout de trois unités de temps. Cependant l'ordonnanceur continue à être invoqué deux fois de plus pour rien, jusqu'à la deuxième activation de la tâche de période 10 qui s'exécute à cette date.

À coté de ce type d'ordonnanceur dirigé par le temps invoqué périodiquement, il existe un autre type d'ordonnanceur dirigé par le temps qui permet d'invoquer l'ordonnanceur uniquement à des dates choisies [Kod07, Pon01]. Pour cela, on utilise une horloge, non nécessairement périodique, dont les instants sont déterminés par un "timer". À chaque instant de l'horloge on calcule l'instant suivant en armant le "timer" avec une certaine durée. Celui-ci se décrémente jusqu'à son expiration. On modifie alors à nouveau sa durée, pour déterminer la prochaine date à laquelle on va invoquer l'ordonnanceur. Ceci a l'avantage d'invoquer l'ordonnanceur uniquement aux dates contenues dans la table d'ordonnancement. Son principal inconvénient est que la modification du "timer" de l'ordonnanceur a un coût. Mais ce coût est constant et peut être ajouté dans le coût de l'ordonnanceur qu'on a supposé inclus dans le WCET des tâches.

Nous avons donc choisi le type d'ordonnanceur dirigé par le temps qui permet d'invoquer l'ordonnanceur uniquement à des dates choisies. Afin d'invoquer l'ordonnanceur aux dates données dans notre table d'ordonnancement, on détermine les durées d'expiration du "timer" de l'ordonnanceur (durée pendant laquelle le "timer" est décrémenté jusqu'à arriver à expiration) en faisant des différences de dates. Par exemple dans la table d'ordonnancement, supposons que l'on est à la date t_0 qui est l'origine des dates et que l'ordonnanceur doit être appelé successivement aux dates t_1, t_2 et t_3. Pour pouvoir invoquer notre ordonnanceur exactement aux dates t_1, t_2 et t_3, à la date t_0 on déclenche le "timer" de l'ordonnanceur avec la durée $t_1 - t_0$. Ceci pour avoir une invocation à la date t_1. À la prochaine expiration du "timer", correspondant à la date t_1, on modifie la durée du "timer" qui passe à $t_2 - t_1$. Ceci pour avoir une invocation de l'ordonnanceur à la date t_2. À la prochaine expiration du "timer", correspondant à la date t_2, on modifie sa durée qui passe à $t_3 - t_2$. Ceci pour avoir une invocation de l'ordonnanceur à la date t_2. Ce calcul des durées d'expiration du "timer" de l'ordonnanceur est effectué avant l'appel de l'ordonnanceur en ligne. Ainsi à chacune des expirations du "timer", correspondant à une invocation de l'ordonnanceur, l'ordonnanceur n'a rien d'autre à faire que de lire la tâche qui a été sélectionnée lors de l'analyse d'or-

donnançabilité et de la faire exécuter par le processeur. Ceci a l'avantage de rendre le temps de sélection constant (temps de lecture dans une table pour le processeur considéré) qui dans les "ordonnanceurs classiques" en ligne consiste à mettre à jour la liste des tâches qui sont dans l'état prêt d'exécution (voir section 1.7.2) et de sélectionner, selon un certain algorithme par exemple RM, DM, EDF, etc., celle qui doit s'exécuter. Dans ce cas la sélection de la tâche, effectuée en ligne, dépend du nombre de tâches prêtes à s'exécuter qui varie d'une invocation de l'ordonnanceur à une autre. Un autre avantage d'utiliser notre ordonnanceur fondé sur une table d'ordonnancement, est qu'on n'a pas besoin de synchroniser l'accès aux mémoires de données partagées par plusieurs tâches, car cette synchronisation a été déjà effectuée lors de l'analyse d'ordonnançabilité hors ligne 5.4.

7.3 Implantation de l'ordonnanceur

Dans cette section on présente les principes d'implantation de notre ordonnanceur en ligne.

Notre ordonnanceur en ligne prend en entrée la table d'ordonnancement, obtenue après l'analyse d'ordonnançabilité hors ligne, représentée par le tableau T_{p_j} dont la taille est notée $SIZE\_T$. On utilise un "timer" pour déterminer les invocations de l'ordonnanceur. Ce "timer" est armé avec une valeur qui est une durée, lue dans le tableau T_{p_j}. Cette valeur est décrémentée jusqu'à expiration du "timer" qui conduit à invoquer l'ordonnanceur. On parcourt le tableau T_{p_j} à l'aide d'un indice noté i qui suit les dates d'invocation de l'ordonnanceur et telle que la différence entre deux dates correspond à la durée d'expiration du "timer" contenue dans le tableau. Puisque l'exécution des tâches se répète indéfiniment à partir de la date $r_{max} + H_n$, date de début de la phase permanente, lorsque l'indice i est égal à $SIZE\_T - 1$ i est initialisé à i_perm, avec i_perm indice du tableau correspondant à la date début de la phase permanente. Chaque élément i du tableau T_{p_j}, noté $T_{p_j}[i]$, contient les informations nécessaires pour exécuter la tâche qui a été sélectionnée lors de l'analyse d'ordonnançabilité hors ligne. Un élément du tableau contient les champs suivants :

- id : identifiant de la tâche sélectionnée,
- $durée$: durée d'expiration du "timer", qui permet d'armer le "timer" et de déterminer si $i = 0$ la première invocation de l'ordonnanceur sinon la prochaine invocation l'ordonnanceur,
- $code$: adresse du code de la tâche à exécuter à la date correspondant à l'indice i du tableau,
- $statut$: permet de déterminer si le code $T_{p_j}[i].code$ reprend son exécution suite à une préemption lors de sa dernière exécution, dans ce cas $T_{p_j}[i].statut = r$ sinon si le code de la tâche démarre une nouvelle exécution $T_{p_j}[i].statut =$

d sinon $T_{p_j}[i].statut = idle$. Ce dernier cas correspond à l'exécution du code $idle$ de la tâche $idle$ qui est une boucle infinie.

L'ordonnanceur en ligne est donné à l'algorithme 12. Dans cet algorithme on utilise un unique "timer" matériel qu'on arme initialement avec la durée $(r_{min} - 0)$ (on suppose que 0 est l'origine des dates). Cette durée se décrémente et provoque une interruption quand elle passe à 0. Cette interruption entraîne un sauvegarde de contexte du processeur. À chaque interruption provoquée par l'expiration du "timer", qu'on associe avec un élément i du tableau T_{p_j}, l'ordonnanceur est invoqué par déclenchement du programme d'interruption $program\_interrupt$. Ce programme d'interruption $program\_interrupt$ arme à nouveau le "timer" $timer$ avec la durée $T_{p_j}[i].durée$ et met à jour les indices i, i_prec et i_succ. Il sélectionne ensuite la tâche $T_{p_j}[i].id$ dont il charge le code $T_{p_j}[i].code$ pour exécution. Le programme d'interruption retourne avec le pointeur d'instruction à la première instruction du code $T_{p_j}[i].code$. Si la tâche sélectionnée était préemptée ($T_{p_j}[i].statut = r$), alors le contexte du processeur au moment de sa préemption est restauré, afin de pouvoir poursuivre son exécution.

Il faut noter qu'une préemption d'une tâche qu'on avait prévue lors de l'analyse d'ordonnançabilité hors ligne, peut ne pas avoir lieu lors de l'exécution en ligne, car cette tâche peut finir son exécution plus tôt. Dans ce cas, à la date de reprise d'exécution de cette tâche, on exécute la tâche $idle$ jusqu'à la fin prévue. Ce mécanisme permet de garantir, en ligne, la viabilité de l'analyse d'ordonnançabilité garantie hors ligne.

On suppose que la durée d'exécution du programme d'interruption est inférieure à la plus petite durée d'expiration du "timer". Ceci permet de prendre en compte toutes les interruptions, donc toutes les invocations de l'ordonnanceur.

7.4 Application

Soient deux tâches $\tau_1 = (1, 1, 4, 4)$ et $\tau_2 = (0, 2, 8, 8)$. D'après l'analyse d'ordonnançabilité hors ligne ces deux tâches sont ordonnançables. La table d'ordonnancement de ces deux tâches est donnée à la table 7.1.

À la figure 7.1 on présente le fonctionnement de l'ordonnanceur en ligne pour τ_1 et τ_2. Dans cette figure, on a une étape d'initialisation qui consiste à armer pour la première fois le "timer" avec la durée 0, à autoriser les interruptions venant de ce "timer" et à initialiser les indices i, i_prec et i_succ ($i = 0$, $i\_prec = 0$ et $i\_succ = 0$). À $t = 0$, on a une première interruption du "timer" qui entraîne une sauvegarde de contexte. Ensuite l'ordonnanceur est invoqué par déclenchement du programme d'interruption $program\_interrupt$. Celui-ci commence par armer à nouveau le "timer" avec la durée $T[i].durée$ et met à jour les indices i,

Algorithme 12 Ordonnanceur en ligne de p_j

```
1: % Entrée : Table d'ordonnancement T_{p_j} du processeur p_j
2: i ← 0
3: i_prec ← 0
4: i_succ ←
5: % Armer le "timer" timer la première fois
6: timer ← armer_timer(r_min)
7: % En utilisant la fonction autoriser_interruption, autoriser les
8: % interruptions venant du périphérique timer et à chaque
9: % occurrence de ces interruptions exécuter le programme d'interruption
   program_interrupt
10: autoriser_interruption(timer, program_interrupt))
11: % Boucle qui s'exécute s'il n y a pas d'interruption ou lorsqu'il y a
12: % un retour suite à une interruption
13: tant que 1 faire
14:     % Instruction qui ne fait rien
15:     nop
16: fin tant que
```

i_prec et i_succ ($i = 0$, $i\_prec = 0$ et $i\_succ = 1$). Il sélectionne ensuite la tâche $T[i].id = \tau_2$ dont il charge le code $T[i].code$ pour exécution. Le programme d'interruption retourne avec le pointeur d'instruction à la première instruction du code $T[i].code$. À $t = 1$, on a une deuxième interruption qui entraîne une sauvegarde de contexte. Ensuite l'ordonnanceur est invoqué par déclenchement du programme d'interruption $program\_interrupt$. Celui-ci arme à nouveau le "timer" avec la durée $T[i].durée$ puis met à jour les indices i, i_prec et i_succ ($i = 1$, $i\_prec = 0$ et $i\_succ = 2$). Il sélectionne ensuite la tâche $T[i].id = \tau_1$ dont il charge le code $T[i\_nex].code = \tau_1()$ pour d'exécution. La sélection de la tâche τ_1 entraîne la préemption de $\tau_2 a$. Le programme d'interruption retourne avec le pointeur d'instruction à la première instruction du code $\tau_1()$. À date $t = 2$, on a une interruption du "timer", qui entraîne une sauvegarde de contexte. Ensuite l'ordonnanceur est invoqué par déclenchement du programme d'interruption $program\_interrupt$. Celui-ci commence par armer à nouveau le "timer" avec la durée $T[i].durée$ et met à jour les indices i, i_prec et i_succ ($i = 2$, $i\_prec = 1$ et $i\_succ = 3$). Il sélectionne ensuite la tâche $T[i].id = \tau_2$ qui reprend son exécution. Le contexte d'exécution de τ_2 est restauré. Le programme d'interruption retourne avec le pointeur d'instruction à l'instruction qui était en cours lors de la préemption de τ_2. Ceci se répète indéfiniment.

Algorithme 13 Programme d'interruption $program\_interrupt$

% On arme le "timer" $timer$ avec la durée
% $T_{p_j}[i].durée$ avec la fonction $armer\_timer$
$armer\_timer(timer, T_{p_j}[i].durée)$
% On met à jour les indices i, i_prec, et i_succ pour qu'ils correspondent
% respectivement à l'indice de élément du tableau en cours, à l'élément
% précédent et le prochain élément du tableau
$i\_prec \leftarrow i$
$i \leftarrow i\_succ$
si $i = SIZE\_T - 1$ **alors**
    $i\_succ \leftarrow i\_perm$
sinon
    $i\_succ \leftarrow i + 1$
fin si
% Si l'exécution de $T_{p_j}[i].code$ a été préemptée ce qui correspond à
% $Tp_j[i].statut = r$ alors restaurer son contexte d'exécution
si $Tp_j[i].statut = r$ **alors**
    Restaurer le contexte de $T_{p_j}[i].code$
    Retourner à l'instruction de $T_{p_j}[i].code$ qui était en cours au moment de la préemption de $T_{p_j}[i].code$
sinon
    % Sinon, l'exécution de $T_{p_j}[i].code$ n'était pas préemptée
    % donc on exécute directement le code $T_{p_j}[i].code$, on charge son code dans la pile d'exécution avec la fonction $charger$
    $charger(T_{p_j}[i].code)$
    Retourner à la première instruction du code $T_{p_j}[i].code$
fin si

id	$durée$	$code$	$statut$
τ_2	1	$\tau_2()$	d
τ_1	1	$\tau_1()$	d
τ_2	2	$\tau_2()$	r
$idle$	1	$idle()$	$idle$
τ_1	1	$\tau_1()$	d
$idle$	2	$idle()$	$idle$
τ_2	1	$\tau_2()$	d
τ_1	1	$\tau_1()$	d
τ_2	2	$\tau_2()$	r
$idle$	1	$idle()$	$idle$
τ_1	1	$\tau_1()$	d
$idle$	2	$idle()$	$idle$
τ_2	1	$\tau_2()$	d

TABLE 7.1 – Table d'ordonnancement T de τ_1 et τ_2

7.5 Gestion des communications inter-processeurs

Soit la communication $com(p_l, \tau_i, p_k, \tau_j)$ correspondant au transfert d'une donnée produite par τ_i sur le processeur p_l et reçue par τ_j sur le processeur p_k. Soit $send(p_k, \tau_i)$ la fonction qui permet d'envoyer la donnée produite par τ_i au processeur p_k et $receive(p_l, \tau_i)$ la fonction qui permet de recevoir la donnée de τ_i provenant de p_l. La communication $com(p_l, \tau_i, p_k, \tau_j)$ s'exécute sur le processeur de communication $pcom_{lk} = (COM_l, M, COM_k)$. Elle est réalisée en exécutant de manière non préemptive la fonction $send(p_k, \tau_i)$ sur le communicateur COM_l puis en exécutant de manière non préemptive la fonction $receive(p_l, \tau_i)$ sur le communicateur COM_k. En supposant qu'on connaît la durée d'exécution d'un $send(p_k, \tau_i)$, donnée par C_s, nous pouvons produire une table d'ordonnancement pour chaque communicateur. Pour cela pour chaque communication $com(p_l, \tau_i, p_k, \tau_j)$ qui doit être réalisée dans l'intervalle de temps $[t_0, t_1]$, on exécute la fonction $send(p_k, \tau_i)$ à la date t_0 sur COM_l et on exécute la fonction $receive(p_l, \tau_i)$ à la date $t_0 + C_s$ sur le communicateur COM_k. On rappelle que cela est possible car les communicateurs ont la même base de temps. On remplace les $send$ et $receive$ utilisés dans le cas d'un médium par passage de messages par des $write$ et $read$ dans le cas d'un médium par mémoire partagée.

Considérons le communicateur COM_l. On utilise un "timer" matériel afin d'invoquer son ordonnanceur aux dates définies dans la table d'ordonnancement de COM_l. Les durées d'expiration sont déterminées par différence de dates selon le même principe présentée à la section 7.2. Le "timer" est armé initialement avec

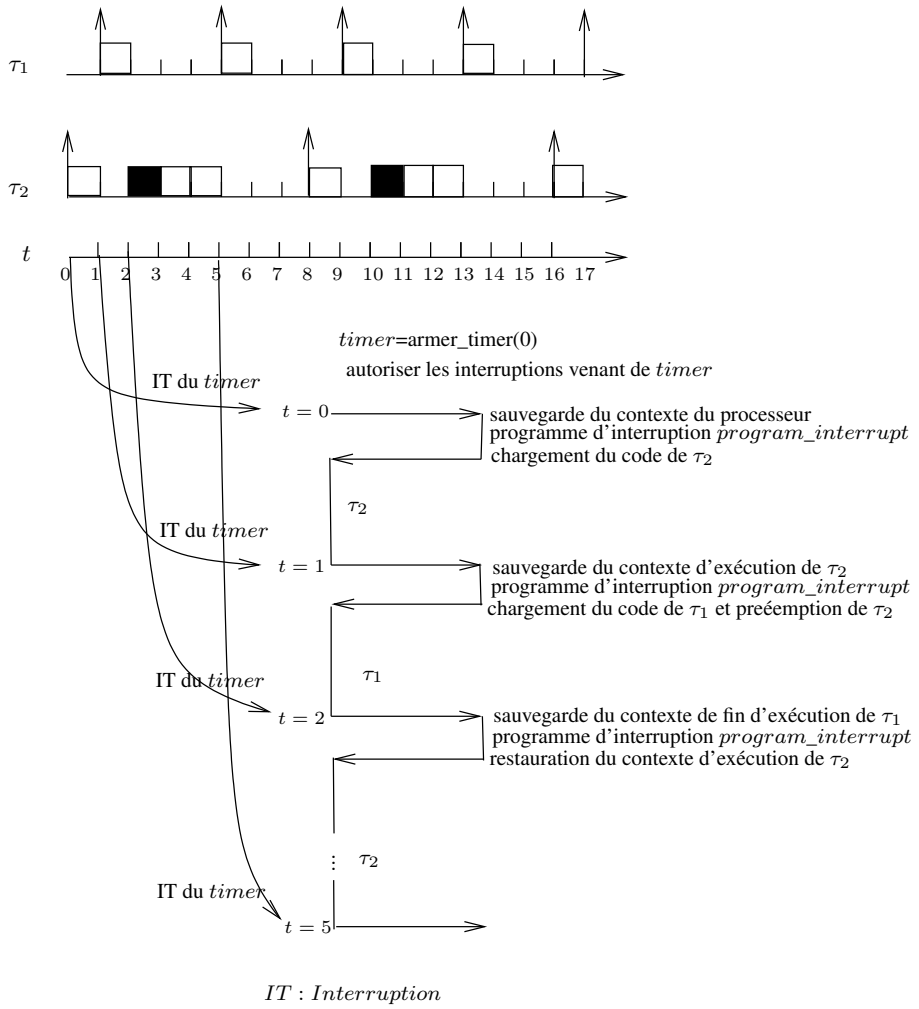

FIGURE 7.1 – Exemple de déroulement de l'algorithme 12

$(t_o - 0)$, t_0 est la plus petite date dans la table d'ordonnancement de COM_l. Ensuite à chaque expiration du "timer", on arme à nouveau le "timer" avec une durée lue dans la table, puis on exécute la tâche correspondante dans la table.

La différence principale entre un communicateur et un processeur de calcul est que dans un communicateur les tâches ($send$, $receive$, $write$, $read$) s'exécutent de manière non préemptive. Les durées d'expiration du "timer" sont déterminées de telles sorte qu'il n'y ait pas de préemptions. Par contre dans un processeur de calcul, les tâches peuvent être préemptées.

7.6 Conclusion

Dans ce chapitre nous avons présenté le principe d'ordonnanceur en ligne et son implantation sur un processeur. Cet ordonnanceur en ligne est dirigé par le temps. Les dates d'invocation de l'ordonnanceur sont déterminées en utilisant la table d'ordonnancement produite lors de l'analyse d'ordonnançabilité hors ligne. Dans cette analyse d'ordonnançabilité, nous avons pris en compte le coût exact de la préemption et les coûts de communication inter-processeur. Cet ordonnanceur en ligne a l'avantage de rendre constant le coût de sélection de la tâche à exécuter qui revient à une lecture dans la table d'ordonnancement produite hors ligne. Il évite aussi de synchroniser, lors de l'exécution des tâches, l'accès aux mémoires de données partagées, car cette synchronisation a été faite lors de l'analyse d'ordonnançabilité hors ligne. Dans le cas multiprocesseur à chaque processeur utilise des communicateurs pour effectuer les transferts de données. Nous avons présenté le principe d'un l'ordonnanceur en ligne pour chaque communicateur.

Conclusion générale et perspectives

Conclusion générale

Dans cette thèse nous avons étudié le problème d'ordonnancement temps réel multiprocesseur préemptif avec prise en compte du coût exact du système d'exploitation. Pour cela nous avons dans un premier temps étudier l'ordonnancement multiprocesseur hors ligne de **tâches indépendantes** avec prise en compte du coût exact de la préemption en proposant une heuristique d'ordonnancement multiprocesseur. Cette heuristique est de type gloutonne et utilise la stratégie par partitionnement de tâches. Pour prendre en compte le coût exact de la préemption sur chaque processeur nous avons utilisé la condition d'ordonnançabilité proposée par Meumeu et Sorel [Yom09] qui ne s'applique qu'à des tâches à priorités fixes. L'heuristique permet de maximiser le facteur d'utilisation restant sur les processeurs et réduit le temps de réponse des tâches.

Ensuite, nous avons étudié l'ordonnancement multiprocesseur hors ligne de **tâches dépendantes**. Puisque la condition d'ordonnançabilité utilisée pour ordonnancer les tâches indépendantes ne s'applique qu'à des tâches à priorités fixes, elle ne permet pas de gérer les inversions de priorités que peuvent entraîner les tâches dépendantes. Nous avons donc proposé une nouvelle condition d'ordonnançabilité pour des tâches à priorités dynamiques. Elle prend en compte le coût exact de la préemption et de l'ordonnanceur, et les dépendances sans aucune perte de données. Ensuite en utilisant toujours la stratégie d'ordonnancement par partitionnement, nous avons proposé pour des tâches dépendantes une heuristique d'ordonnancement multiprocesseur qui réutilise cette nouvelle condition d'ordonnançabilité au niveau de chaque processeur. Cette heuristique d'ordonnancement prend en compte les coûts de communication inter-processeurs. Elle permet aussi de minimiser sur chaque processeur le makespan (temps total d'exécution) des tâches. Cette heuristique produit pour chaque processeur une table d'ordonnancement hors ligne contenant les dates de début et de fin de chaque tâches et de chaque communication inter-processeur.

En supposant que nous avons une architecture multiprocesseur de type dirigée par le temps (Time Trigger TT) pour laquelle tous les processeurs ont une

référence de temps unique, nous avons proposé pour chacun des processeurs un ordonnanceur en ligne qui utilise la table d'ordonnancement produite lors de l'ordonnancement hors ligne. Cet ordonnanceur en ligne a l'avantage d'avoir un coût constant et facile à déterminer de manière exacte. En effet il correspond uniquement au temps de lecture dans la table d'ordonnancement pour obtenir la tâche sélectionnée lors de l'analyse d'ordonnançabilité hors ligne, alors que dans les "ordonnanceurs classiques" en ligne ce coût correspond à mettre à jour la liste des tâches qui sont dans l'état prêt à l'exécution puis à sélectionner une tâche selon un algorithme, par exemple RM, DM, EDF, etc. Il varie donc avec le nombre de tâches prêtes à s'exécuter qui change d'une invocation à l'autre de l'ordonnanceur. C'est ce coût qui est utilisé dans les analyses d'ordonnançabilités évoquées ci-dessus. Un autre avantage est qu'il n'est pas nécessaire de synchroniser l'accès aux mémoires de données partagées par plusieurs tâches, car cette synchronisation a été déjà effectuée lors de l'analyse d'ordonnançabilité hors ligne.

Perspectives

Afin d'augmenter le taux de succès dans l'analyse d'ordonnançabilité, il serait intéressant d'autoriser la migration de certaines tâches tout en prenant en compte les coûts de migration dans la condition d'ordonnançabilité.

Nous avons considéré que les dépendances de données étaient sans pertes de données, ceci entraîne des restriction au niveau des tâches puisque les tâches dépendantes doivent avoir des période multiples. Il serait intéressant d'assouplir ce mode de dépendances données de telle sorte qu'il puisse y avoir des pertes de données sans nuire au bon fonctionnement de l'application considérée.

Dans l'architecture multiprocesseur que nous avons considérée, nous avons des communicateurs qui permettent de paralléliser les calculs et les communication. Ce type d'architecture donne de meilleures performance mais paralléliser n'est pas toujours possible. Il serait intéressant d'étudier des types d'architectures qui ne disposent pas de communicateurs et dont les processeurs ont des caches, pipeline, etc.

Poursuivre le travail d'implantation effectué sur l'OS Linux de l'ordonnanceur en ligne que nous avons proposé, en réalisant une implantation sur un processeur sans OS. Cela demandera aussi de mesurer le coût d'une préemption sur ce processeur.

Bibliographie

[AaR95] J. M. Adan, M. F. Magalh aes, and K. Ramamritham.
Meeting hard real-time constraints using a client-server model of interaction.
In *Proceedings of the 7th Euromicro Workshop on Real-Time systems*, pages 286–293, 1995.

[AB08] B. Andersssons and K. Bletsas.
Sporadic multiprocessor scheduling with few preemptions.
In *Proceedings of the 22th Euromicro Conference on Real-Time Systems ECRTS'08*, 2008.

[ABJ01] B. Anderson, S. Baruah, and J. Jonson.
Static-priority scheduling on multiprocessors.
In *Proceedings of 21th IEEE Real-Time Systems Symposium RTSS'01*, 2001.

[ACD74] L.T. Adams, K. M. Chandy, and J. R. Dickson.
A comparison of list schedules for parallel processing systems.
Commun. ACM, 17:685–690, December 1974.

[AH98] P. Altenbernd and H. Hansson.
The slack method: A new method for static allocation of hard real-time tasks.
Real-Time Systems, 15:103–130, 1998.

[AJ] B. Andersson and J. Jonsson.
Fixed-priority premtive multiprocessor scheduling: To partition or not partition.
In *Proceedings of the 7th International Conference on Real-Time Systems and Applications RTCSA'00*.

[AJ03] B. Andersson and J. Jonsson.
The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%.
In *Proceedings of the 15th Euromicro Conference on Real-Time Systems, ECRT'03*, 2003.

[AT06] B. Andersson and E. Tovar.

Multiprocessor scheduling with few preemptions.
In *Proceeding of the 12th IEEE Internationnal Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'07, IEEE Computer Society, Washington, DC, USA, 20006*, pages 322–334, 2006.

[Bak91] Theodore P. Baker.
Stack-based scheduling of realtime processes.
Real-Time Systems, 3(1):67–99, 1991.

[Bak03] T. P. Baker.
Multiprocessor edf and deadline monotonic schedulability analysis.
In *Proceedings of 24th IEEE International Real-Time Systems Symposium RTSS'03*, 2003.

[Bar04] S. K. Baruah.
Optimal utilization bounds for the fixed-priority scheduling of the periodic task system on identical multiprocessors.
IEEE Transactions On Computers, vol. 53(6), June 2004.

[Bar07] Sanjoy Baruah.
Techniques for multiprocessor global schedulability analysis.
In *Proceedings of 28th IEEE Real-Time Systems Symposium RTSS'07*, 2007.

[BB05] E. Bini and G. C. Buttazzo.
Measuring the performance of schedulability tests.
Real-Time Syst., 30(1-2):129–154, May 2005.

[BB08] S. Baruah and T.P Baker.
Schedulability analysis of global edf.
Real Time Systems, vol. 38:223–235, December 2008.

[Ben06] F. Benhamou, editor.
volume 4204 of *Lecture Notes in Computer Science*. Springer, 2006.

[BF97] J. Blazewicz and G. Finke.
Minimizing mean weigthed execution time loss on identical and uniform processors.
Information processing letters, 24:259–263, 1997.

[BG03] S. K. Baruah and J. Goossens.
Rate-monotonic scheduling on uniform multiprocessors.
IEEE Transactions On Computers, vol. 52(7), July 2003.

[BG08] S. K. Baruah and J. Goossens.
The edf scheduling of sporadic task system on multiprocessors.
In *Proceedings of 24th IEEE International Real-Time Systems Symposium RTSS'08*, 2008.

[Bim07] F. Bimbard.
Dimensionnement temporel de systèmes embarqués: application à OSEK.
PhD thesis, CEDRIC Laboratory, Paris, France, 2007.

[BLOS95] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son.
New strategies for assigning real-time tasks to multiprocessor systems.
IEEE Transaction on computers, 44(12):1429–1442, April 1995.

[BTW95] A. Burns, K. Tindell, and A. Wellings.
Effective analysis for engineering real-time fixed priority schedulers.
IEEE Trans. Softw. Eng., 21:475–480, May 1995.

[Bur94] Alan Burns.
Preemptive priority-based scheduling: An appropriate engineering approach.
In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[CA95] S. Cheng and A. K. Agrawala.
Allocation and scheduling of real-time periodic tasks with relative timing constraints.
In *Second International Workshop on Real-Time Computing Systems and Applications (RTCSA), IEEE*, pages 25–27, 1995.

[CCH+99] P. Calégari, G Coray, A. Hertz, D. Kobler, and P. Kuonen.
A taxonomy of evolutionary algorithms in combinatorial optimization.
Journal of Heuristics, 5:145–158, July 1999.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri.
Ordonnancement temps réel - Cours et exercices corrigés.
Hermès, janvier 2000.

[CG11] P. Courbin and L. Georges.
Fortas: Framework for real-time analysis and simulation.
In *Proceedings of the 2th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, Porto, Portugal, July 2011.

[CGG04] A. Choquet-Geniet and E. Grolleau.
Minimal schedulability interval for real-time systems of periodic tasks with offsets.
Theor. Comput. Sci., 310(1-3):117–134, 2004.

[CGMV98] E.G. Coffman, G. Galambos, S. Martello, and D. Vigo.
Bin packing approximation algorithms: Combinatorial analysis.

Handbook of combinatorial optimization, 1998.

[CHMTE80] W. W. Chu, L. J. Holloway, L. Min-Tsung, and K. Efe.
Task allocation in distributed data processing.
Computer, 13:57–69, November 1980.

[CL87] W. W. Chu and L. M-T. Lan.
Task allocation and precedence relation for distributed real-time systems.
IEEE Transactions on Computers, vol.C-36(18), June 1987.

[CMTN08] J. Carlson, J. Mäki-Turja, and M. Nolin.
Event-pattern triggered real-time tasks.
In *Proceedings of the 16th International Conference on Real-Time and Network Systems, RTNS'08*, Rennes, France, 2008.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf.
Dynamic scheduling of real-time tasks under precedence constraints.
Real-Time System., 2(3):181–194, sepember 1990.

[Cuc04] L. Cucu.
Ordonnancement non préemptif et condition d'ordonnançabilité pour systèmes embarqués à contraintes temps réel.
PhD thesis, Université Paris Sud, 2004.

[CY90] G-H. Chen and J-S. Yur.
A branch-and-bound-with-underestimates algorithm for the task assignment problem with precedence constraint.
In *Procedings of ICDCS, , 20006*, pages 494–501, 1990.

[DB09] R. I. Davis and A. Burns.
A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems.
Technical report, University of York, Department of Computer Science, 2009.

[DD86] S. Davari and S. K. Dhall.
An on line algorithm for real-time allocation.
In *Proceedings of IEEE Real-Time Systems Symposium*, pages 194–200, 1986.

[DL78] S.K. Dhall and C.L. Liu.
On a real-time scheduling problem.
Operation Research, vol. 26(1), 1978.

[DT03] G. B. Dantzig and M. N. Thapa.
Linear programming 2: Theory and.
Springer Series in Operations Research, 2003.

[EJ00] C. Ekelin and J. Jonsson.
Solving embedded system scheduling problems using constraint programming.
Technical report, Dept. of Computer Engineering, Chalmers University of Technology, 2000.

[EL97] E.Aarts and J. K. Lenstra.
Local search in combinatorial optimization.
Wiley, 1997.

[ERC95] J. Echague, I. Ripoll, and A. Crespo.
Hard real-time preemptively scheduling with high context switch cost.
In *Proceedings of 7th Euromicro workshop on Real-Time Systems*, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[ESHA90] R. Hong E. S. Hou and N. Ansari.
Efficient multiprocessor scheduling based on genetic algorithms.
Technical Report CS-TR-3402, 1990.

[ESHA94] R. Hong E. S. Hou and N. Ansari.
A genetic algorithm for multiprocessor scheduling.
Transaction on parallel and distributed systems, 5(2), 1994.

[FBG+10a] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti.
Scheduling dependent periodic tasks without synchronization mechanisms.
In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 301–310, Washington, DC, USA, 2010.

[FBG+10b] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti.
Scheduling dependent periodic tasks without synchronization mechanisms.
In *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, Stockolm, Sweden, April 12-15th 2010.

[FGLM96] P. M. Franca, M. Gendreau, G. Laporte, and F. M. Muller.
A tabu search heuristic for the multiprocessor scheduling problem with sequence dependent setup times.
International Journal of Production Economics, 43(2-3):79–89, June 1996.

[FGPR11] J. Forget, E. Grolleau, C. Pagetti, and P. Richard.
Dynamic Priority Scheduling of Periodic Tasks with Extended Precedences.

In *IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA)*, Toulouse, France, September 2011.

[FJF62] L.R. Ford, Jr., and D.R. Fulkerson.
Flows network.
Princeton Univ. Press, 1962.

[GFB03] J. Goossens, S. Funk, and S. Baruah.
Priority-driven scheduling of periodic task systems on multiprocessors.
Real Time Systems, vol. 25:187–205, 2003.

[GJ79] Garey and Johnson.
Computers and intractability: a guide to the theory of NP-completeness.
W.H. Freeman and Company, New York, NY, USA, 1979.

[GL93] F. Glover and M. Laguna.
Tabu search, pages 70–150.
John Wiley & Sons, Inc., New York, NY, USA, 1993.

[GLN01] P. Gai, G. Lipari, and M. D. Natale.
Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip.
In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, pages 73–83, Washington, DC, USA, December 2001. IEEE Computer Society.

[Goo99] J. Goossens.
Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints.
PhD thesis, Université Libre de Bruxelles, 1999.

[Goo07] J. Goossens.
Introduction à l'ordonnancement temps réel multiprocesseur.
Ecole d'étè temps réel, 2007.

[Gra00] T. Grandpierre.
Modèlisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.
PhD thesis, Université Paris-Sud, 30/11/2000.

[GVY90] A. Gerasoulis, S. Venugopal, and T. Yang.
Clustering task graphs for message passing architectures.
In *Proceedings of the 4th international conference on Supercomputing*, pages 447–456, New York, NY, USA, 1990. ACM.

[Hav68] J. W. Havende.
Avoiding deadlock in multitasking systems.
IBM Syst. J., 7(2):74–84, June 1968.

[HCAL89] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee.
Scheduling precedence graphs in systems with interprocessor communication times.
SIAM J. Comput., 18:244–257, April 1989.

[HGRC96] M. Hutton, J. P. Grossman, J. Rose, and D. Corneil.
Characterization and parameterized random generation of digital circuits.
In *Proceedings of the 33rd annual Design Automation Conference*, pages 94–99, New York, NY, USA, 1996.

[Hla04] P. E. Hladik.
Ordonnançabilité et placement des systèmes temps réel ditribuès préemtifs et à priorités fixes.
PhD thesis, Université de Nantes, Spécialité Automatique et informatique Appliquée, 17/12/2004.

[JP86] M. Joseph and P. Pandya.
Finding response times in a real-time system.
Comput. J., 29(5):390–395, 1986.

[Kai81] C. Kaiser.
De l'utilisation de la priorité en presence d'exclusion mutuelle.
Technical Report RR-0084, INRIA, Jul 1981.

[Kal04] H. Kalla.
Génération automatique de distributions/ordonnancements temps réel fiables et tolérant les fautes.
PhD thesis, Institut National Polytechnique de Grenoble, Spécialité Systèmes et Logiciel, 17/12/2004.

[KAS93] D. I. Katcher, H. Arakawa, and J. K. Strosnider.
Engineering and analysis of fixed priority schedulers.
IEEE Trans. Softw. Eng., 19(9):920–934, September 1993.

[KBKP04] F. Kiraç, U. Bilge, M. Kurtulan, and P. Pekgün.
A tabu search algorithm for parallel machine total tardiness problem.
2004.

[Ker09] Omar Kermia.
Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence, de périodicité stricte et de latence.
PhD thesis, Université Paris XI, UFR scientifique d'Orsay, 2009.

[Kod07] A. H. Kodancha.
Time Management in Partitioned Systems.

PhD thesis, Indian Institute of Science, Bangalore, Department of Computer Science and Automation, 2007.

[Kop98] H. Kopetz.
The time-triggered model of computation.
In *Proceedings of the IEEE Real Time Systems Symposium, RTSS'98*, pages 168–177. IEEE Computer Society, 1998.

[KS07] O. Kermia and Y. Sorel.
A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor.
In *Proceedings of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07*, Las Vegas, Nevada, USA, sep 2007.

[KT00] D. Kadamuddi and J. J. P. Tsai.
Clustering algorithm for parallelizing software systems in multiprocessors environment.
IEEE Trans. Softw. Eng., 26:340–361, April 2000.

[KY07] S. Kato and N. Yammasaki.
Real-time scheduling with task splitting on multiprocessors.
In *Proceeding of the 13th IEEE Internationnal Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'07, IEEE Computer Society, Washington, DC, USA*, 2007.

[KY08] S. Katoa and N. Yammasaki.
Semi-partitioning technique for multiprocessor real-time scheduling.
In *Proceedings of WIP Session of the 29th Real-Time Systems Symposium (RTSS), IEEE Computer Society*, 2008.

[LCGG10] I. Lupu, P. Courbin, L. George, and J. Goossens.
Multi-criteria evaluation of partitioning schemes for real-time systems.
In *The 15th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA'10*, Bilbao, Spain, September 2010.

[LDG03] J. M. Lopez, J. L. Diaz, and M. Garcia.
Utilization bounds for multiprocessor rate-monotonic scheduling.
Real Time Systems, vol 24(number 2):5–28, 2003.

[LDG04] J. M. Lopez, J. L. Diaz, and M. Garcia.
Utilization bounds for edf multiprocessor rate-monotonic scheduling.
Real Time Systems, vol 28(number 1), 2004.

[LGD04] J. M. Lopez, M. Garcia, and D. F. Diaz.
Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling.
IEEE Transactions on Parallel and Distributed Systems, vol. 15(7), October 2004.

[LGDG00] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia.
Worst-case utilization bound for edf scheduling on real-time multiprocessor system.
In *Proceedings of 19th Euromicro Conference on Real-Time Systems, ECRTS'00*, Stockholm, Sweden, june 2000.

[LL73] C. L. Liu and J W. Layland.
Scheduling algorithms for multiprogramming in a hard-real-time environnment.
Journal of the ACM, vol. 20(1), January 1973.

[LM80] J. Leung and M. Merrill.
A note on preemptive scheduling of periodic, real-time tasks.
Information Processing Letters, 11(3), November 1980.

[LSD89] J. P. Lechoczky, L. Sha, and Y. Ding.
The rate-monotonic scheduling algorithm:exact characterization an average behavior.
In *IEEE Real-Time Systems Symposium RTSS'89*, pages 166–171, 1989.

[LW82] J. Y-T. Leung and J. Whitehead.
On the complexity of fixed-priority scheduling of periodic real-time tasks.
Performance Evaluation, vol. 2:237–250, 1982.

[Mit02] J. E. Mitchell.
Branch-and-cut algorithms for combinatorial optimization problems.
In *Handbook of Applied Optimization*, pages 65–67, Oxford University Press, 2002.

[MKTM93] C. L. McCreary, A. A. Khan, J.J. Thompson, and M. E. McArdle.
A comparison of heuristics for scheduling dags on multiprocessors.
Technical report, Auburn, AL, USA, 1993.

[Mok83] A. K. Mok.
Fundamental design problems of distributed systems for the hard-real-time environment.
Technical report, Cambridge, MA, USA, 1983.

[OB98] D. I. Oh and T. P. Baker.

Utilization bounds for n-processor rate monotone scheduling with stable processor assignment.
Real Time Systems, vol. 15(2):183–193, September 1998.

[OS93] Y. Oh and S.H. Son.
Tight performance bounds of heuristics for a real-time scheduling problem.
Technical Report CS-93-24, Univ. of Virginia. Dep. of Computer Science, Charlottesville, VA 22903, May 1993.

[OS95] Y. Oh and S.H. Son.
Fixed-prioriy scheduling of periodic task on multiprocessor systems.
Technical Report CS-95-16, Univ. of Virginia. Dep. of Computer Science, Charlottesville, VA 22903, 1995.

[PBAF10] D. Potop-Butucaru, A. Azim, and S. Fischmeister.
Semantics-preserving implementation of synchronous specifications over dynamic tdma distributed architectures.
In *Proceedings of the eighth ACM international conference on Embedded software, EMSOFT'10*, Scottsdale, AZ, USA, October 2010.

[PEHDN08] H. Pierre-Emmanuel, C. Hadrien, A.-M. Deplanche, and J. Narendra.
Solving a real-time allocation problem with constraint programming.
J. Syst. Softw., 81:132–149, January 2008.

[PFB+11] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens.
Multi-task implementation of multi-periodic synchronous programs.
Discrete Event Dynamic Systems, 21(3), sep 2011.

[Pon01] J. M. Pont.
Patterns for Time-triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers.
ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2001.

[PSA97] D-T Peng, K. G. Shin, and T. F. Abdelzaher.
Assignment and scheduling communicating periodic tasks in distributed real-time systems.
IEEE Trans. Softw. Eng., 23:745–758, December 1997.

[Raj90] R. Rajkumar.
Real-time synchronization protocols for shared memory multiprocessor.

In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116 –123, 1990.

[Ram90] K. Ramamritham.
Allocation and scheduling of complex periodics tasks.
In *Proceedings of the 10th International Conference on distributed Computing Systems (ICDCS)*, 1990.

[RCR01] P. Richard, Cottet, and M. Richard.
On-line scheduling of real-time distributed computers with complex communication constraints.
In *ICECCS*, pages 26–34, 2001.

[RM00] S. Ramamurth and M. Moir.
Static-priority periodic scheduling on multiprocessors.
In *Proceedings of 21th IEEE Real-Time Systems Symposium RTSS'00*, 2000.

[Sar84] Michel Sarkarovitch.
Optimisation Combinatoire: Programmation discrète.
Harmann, 1984.

[SB02] A. Srinivasnan and S. Baruah.
Deadline-based scheduling of periodic task systems on multprocessors.
Information Processing letters, vol. 84:93–98, January 2002.

[She11] Ahmad Al Sheikh.
Resource allocation in hard real-time avionic systems. Scheduling and routing problems.
PhD thesis, Institut National des Sciences Appliquées de Toulouse, Département Systèmes Informatiques et Systèmes Embarqués, 2011.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky.
Priority inheritance protocols: An approach to real-time synchronization.
IEEE Transactions on Computers, 39:1175–1185, 1990.

[SS97] O. Sinnen and L. Sousa.
List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures.
Real-Time Systems, 30:81–101, 1997.

[Sto77] H. S. Stone.
Multiprocessor scheduling with the aid of network flow algorithms.
IEEE Trans. Softw. Eng., 3:85–93, January 1977.

[Sub05] K. Subramani.

Periodic linear programming with applications to real-time scheduling.
Mathematical. Structures in Comp. Sci., 15:383–406, April 2005.

[SW00] M. Saksena and Y. Wang.
Scalable real-time system design using preemption thresholds.
In *Proceedings of 1th IEEE Real-Time Systems symposium*, November 2000.

[Tal09] E. G Talbi.
Metaheuristics: From design to implementation.
Wiley, 2009.

[TBW92] K. Tindell, A. Burns, and A Wellings.
Allocating hard real time task.
Real time systems, 4:145–165, 1992.

[Van96] R. J. Vanderbei.
Linear programming: Foundations and extensions, 1996.

[VM05] P. V.Hentenryck and L. Michel.
Constraint-based Local search.
The MIT Press, 2005.

[WS99] Y. Wang and M. Saksena.
Scheduling fixed-priority tasks with preemption threshold.
In *Proceedings of the 6 International Conference on Real-Time Computing Systems and Applications*, RTCSA'99, Washington, DC, USA, 1999.

[XTL10] G. Liao X. Tang, K. Li and R. Li.
List scheduling with duplication for heterogeneous computing systems.
J. Parallel Distrib. Comput., 70:323–329, April 2010.

[Xu93] J. Xu.
Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations.
IEEE Trans. Softw. Eng., 19:139–154, February 1993.

[Yom09] P. Meumeu Yomsi.
Prise en compte du coût exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples.
PhD thesis, Université paris Sud, Spécialité PHYSIQUE, 02/04/2009.

[ZA05] O.U.P. Zapata and P.M. Alvarez.
Edf and rm multiprocessor scheduling algorithms: Survey and performance evaluation.

Technical report, Report No. CINVESTAV-CS-RTG-02. CINVESTAV-IPN, Sección de Computación, Zacatenco, Mexico, Oct 2005.