Université Paris XI
UFR scientifique d'Orsay

École doctorale STITS

Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence, de périodicité stricte et de latence

Thèse présentée

pour obtenir le grade de

Docteur en science

par

Omar KERMIA

soutenue le 19/03/2009 devant le jury composé de

Laurent George	Rapporteurs
Denis Trystram	
Arnaud De La Fortelle	Examinateurs
Alain Mérigot	
Yves Sorel	Directeur de thèse

Année 2009
Numéro d'Ordre :

École Doctorale STITS

Table des matières

Table des figures

Remerciements

Je remercie en premier lieu Yves Sorel, mon directeur de thèse, pour la confiance qu'il m'a témoigné. Il a su apporter un regard critique sur mes travaux et se montrer disponible pour m'écouter et me conseiller. Encadrant idéal, il a su guider mes travaux tout en me laissant une grande autonomie. J'ai notamment acquis auprès de lui la rigueur scientifique qui faisait défaut dans mes présentations ainsi que mes rédactions.

Je tiens à remercier également Denis Trystram et Laurent George qui ont eu la gentillesse de bien vouloir être les rapporteurs de cette thèse. De même, j'adresse un grand merci à Alain Mérigot et Arnaud De La Fortelle pour avoir accepté de faire partie de mon jury de thèse.

Je remercie également tous mes collègues de l'équipe projet AOSTE et des autres équipes projet, les anciens comme les nouveaux, pour les très bon moments que j'ai passés pendant mon séjour à l'INRIA-Rocquencourt.

Un grand merci à tous mes amis qui m'ont soutenu et encouragé.

Résumé de la thèse

La réalisation de systèmes temps réel embarqués complexes que l'on trouve dans les domaines de l'avionique, de l'automobile, de la robotique, etc. conduisent à résoudre des problèmes d'ordonnancement temps réel non préemptif pour des architectures multiprocesseurs en respectant des contraintes multiples de précédence, de périodicité stricte et de latence.

Dans la littérature les problèmes de ce type sont résolus avec des méthodes approchées (heuristiques) donnant des résultats dans un temps raisonnable comparées à des méthodes exactes. Par ailleurs le problème tel que nous le posons a été peu étudié. Ce dernier étant complexe nous avons choisi d'étudier séparément la périodicité d'une part et la latence d'autre part, avec aussi dans les deux cas des contraintes de précédence. L'ensemble des résultats obtenus est utilisé pour traiter l'ordonnancement avec les trois contraintes.

Afin de résoudre le problème d'ordonnancement avec précédence et périodicité stricte nous avons proposé une heuristique composée de trois étapes. La première étape appelée "assignation" est la plus importante car elle permet de décider si un système est ordonnançable ou pas sans être obligé d'attendre l'exécution des deux autres étapes de l'heuristique. Comme nous avons choisi d'utiliser la méthode du partitionnement - partitionner le problème multiprocesseur en plusieurs problèmes monoprocesseur - plutôt que la méthode globale pour faire l'ordonnancement multiprocesseur, nous avons pu donner une condition pour qu'une tâche, éventuellement plusieurs, soient ordonnançables sur un processeur auquel d'autres tâches ont déjà été assignées. Nous avons proposé deux versions d'algorithme d'assignation, une version gloutonne très rapide et une version "recherche locale" fondée sur le retour arrière (backtracking) qui revient à tester localement plusieurs assignations pour trouver celle qui satisfait les contraintes de périodicité stricte. Nous avons montré que la version "recherche locale", bien que moins rapide que la version gloutonne, donne des résultats très proches de ceux d'un algorithme exact de type "Branch & Cut". La seconde étape appelée "déroulement" consiste simplement à répéter chaque tâche et les arcs de précédence qui la concernent suivant le rapport entre l'hyper-période (PPCM des périodes de toutes les tâches) et sa période. La troisième étape consiste à ordonnancer les tâches sur les processeurs auxquels elles ont été assignées tout en minimisant le temps d'exécution de toutes les tâches (makespan), en prenant en compte le coût des communications interprocesseurs dues au fait que deux tâches liées par une précédence ont été assignées à deux processeurs différents. Par ailleurs comme nous considérons des systèmes embarqués pour lesquels les ressources sont limitées nous avons ajouté une quatrième étape, spécifique à l'embarqué, qui effectue de manière gloutonne de la répartition de charge et de mémoire. L'heuristique d'ordonnancement avec précédence et périodicité stricte a été programmée en OCAML dans le logiciel SynDEx diffusé par l'équipe projet AOSTE. Pour

tester ces résultats théoriques ainsi que leur implantation dans le logiciel SynDEx on a effectué une expérimentation sur une application de suivi en train virtuel de CyCabs (véhicule électrique automatique conçu par l'équipe projet IMARA) avec contraintes de précédence et de périodicité.

Afin de résoudre le problème d'ordonnancement multiprocesseur avec précédence et latence nous avons effectué une étude d'ordonnançabilité qui a montré que sa résolution est très liée aux chemins de tâches reliant la paire de tâches sur laquelle la contrainte de latence est imposée. Nous avons proposé une heuristique dans le cas d'une seule latence se composant d'une première étape appelée "clusterisation" et une deuxième étape appelée "union". La clusterisation consiste à regrouper les tâches faisant partie du même chemin dans le graphe et l'union cherche à adapter le nombre de ces clusters au nombre de processeurs en procédant à des unions entre clusters. Le cas de plusieurs latences demande de prendre en compte les différentes possibilités de chemins entre plusieurs paires de tâches soumises à différentes latences. Pour le cas le plus complexe correspondant à des chemins, entre paires de tâches soumises à différentes latences, croisés on a proposé une heuristique qui minimise la durée de l'ordonnancement entre chacune de ces paires de tâches.

Les résultats obtenus précédemment ont été utilisés pour proposer une heuristique d'ordonnancement avec contraintes de précédence, de périodicité et de latence.

Introduction générale

Contexte

Aujourd'hui, les systèmes informatiques temps réel embarqués sont présents dans de nombreux secteurs d'activités : transport terrestre (automobile, ferroviaire, etc.) et aérien, spatial, militaire, robotique, télécommunication, contrôle des systèmes automatisés de production, etc.

On qualifie de temps réel tout système informatique dont le fonctionnement est conditionné par par des contraintes temporelles. Afin de simplifier, à partir de maintenant on utilisera le terme système pour système informatique. La maîtrise des aspects temporels dans un système temps réel est un problème mal connu qui constitue un enjeu important en terme de recherche. En effet dans ce contexte la validité du système ne dépend pas seulement de la justesse des calculs, mais aussi de l'instant de production des résultats. Pour un système temps réel un résultat juste, mais ne respectant pas ses contraintes temporelles, est un résultat inutilisable qui correspond à une faute temporelle. La présence de contraintes temporelles dans les systèmes temps réel est l'aspect fondamental qui les distingue des systèmes classiques. De plus pour certains de ces systèmes temps réel le non respect des contraintes peut avoir des conséquences catastrophiques en termes de pertes humaines, d'écologie, etc. On qualifie de durs, stricts ou critiques de tels systèmes temps réel auxquels nous nous intéresserons plus particulièrement.

Les systèmes temps réel embarqués ont comme particularité d'être intégrés dans un équipement généralement mobile, comme une automobile, un avion, un téléphone, etc., ce qui les distingue des systèmes informatiques classiques. De tels systèmes ont en général des ressources limitées qu'il convient d'exploiter au mieux. De manière générale les systèmes temps réel embarqués conduisent à minimiser des ressources : puissance de calcul, mémoire, consommation électrique, etc.

Enfin ces systèmes temps réel embarqués sont de plus en plus souvent réalisés avec des architectures multiprocesseur distribuées, c'est-à-dire comportant plusieurs processeurs. Ceci est dû d'une part au besoin de puissance de calcul qu'un seul processeur (monoprocesseur) ne peut fournir et d'autre part à un besoin de modularité et de flexibilité, à la fois lors de la conception système pour réutiliser et faire évoluer le nombre de processeurs utilisés et à la fois pour rapprocher les capteurs et les actionneurs des processeurs qui les traitent. Cela afin de minimiser le câblage qui est un élément très coûteux dans une architecture multiprocesseur, et aussi de minimiser les perturbations électromagnétiques.

Au cours des trente dernières années, l'informatique temps réel s'est progressivement établie comme une discipline à part entière qui rassemble une forte communauté issue à la fois du monde académique et de l'industrie. Elle conduit principalement à traiter des problèmes d'ordonnance-

ment temps réel. En effet chacun des processeurs de l'architecture étant séquentiel il s'agit de déterminer dans quel ordre et à quels instants on va devoir exécuter les fonctionnalités du système temps réel, concrétisées par des programmes séquentiels que l'on appelle des tâches. Cette dénomination est utilisée dès que ces programmes sont caractérisés temporellement (période, échéance, etc.). Le choix de cet ordre peut se faire avant l'exécution du système (hors ligne) ou bien pendant l'exécution du système (en ligne).

Les travaux de recherche dans le domaine de l'informatique temps réel ont conduit à proposer aux utilisateurs des outils logiciel leur permettant d'une part de spécifier ces systèmes (spécification fonctionnelle, spécification non fonctionnelle : contraintes temporelles, contraintes matérielles, contraintes de sûreté de fonctionnement, etc.), de faire des vérifications sous la forme de tests ou de preuves formelles (model-checking, preuve de théorème, etc.), et pour finir dans certains cas produire automatiquement du code exécutable conforme aux spécifications.

Objectifs

À cause des exigences de plus en plus fortes des contraintes temps réel et d'embarquabilité, les problèmes d'ordonnancement temps réel deviennent de plus en plus complexes. Afin d'évaluer cette complexité des études de complexité sont nécessaires. Le choix des modèles utilisés est d'autant plus important que cette complexité en dépend. Nous avons choisi dans cette thèse un modèle qui prend en compte des contraintes temps réel ainsi que des contraintes d'embarquabilité. Les contraintes temps réel prises en compte sont considérées comme strictes (dures), c'est-à-dire que le non-respect d'une de ces contraintes peut avoir des conséquences catastrophiques pour le système. Pour compliquer les choses elles sont multiples : précédences, périodicités strictes, latences.

Une tâche, qui constitue la base du modèle, peut être reliée à d'autres tâches par des précédences soit parce qu'elles s'échangent des données soit parce que cela est imposé par le concepteur. Ce qui signifie que le début d'exécution d'une tâche n'est réalisable qu'après que toutes ses tâches prédécesseurs soient exécutées. Chaque tâche possède une période stricte indiquant que l'exécution de celle-ci se fait de manière répétitive et que la durée de temps qui sépare deux dates de début d'exécution successives est toujours la même, et est égale à sa période. De plus un délai peut être imposé entre le début de l'exécution d'une tâche et la fin de l'exécution d'une autre tâches, ces deux tâches pouvant ou non être dépendantes. Nous appelons ce délai, souvent qualifié de "bout-en-bout", une contrainte de latence. Dans un système temps réel complexe il peut y avoir plusieurs latences, éventuellement reliées entre elles ce qui complique le problème. Par ailleurs, pour des raisons d'embarquabilité, chaque tâche est caractérisée par la quantité mémoire qu'elle nécessite pour s'exécuter. En effet, la mémoire, tout comme la consommation, constituent une des principales préoccupations des concepteurs de systèmes temps réel embarqués.

Entre l'approche en-ligne et l'approche hors-ligne, nous nous intéressons à l'approche hors-ligne qui consiste à trouver un ordonnancement valide avant l'exécution du système. La principale spécificité de cette approche est qu'une fois l'ordonnancement calculé, l'exécution du système a un comportement déterministe qui se répète tout le long de la vie du système. Cette propriété prend toute son importance dès qu'il est question de systèmes temps réel stricts complexes où tout doit être maîtrisé. Les tâches prises en compte dans notre modèle sont non-préemptives, c'est-à-dire que

les tâches s'exécutent sans interruption jusqu'à leur terminaison. Nous avons choisi cette approche afin d'éviter de devoir faire le compromis classique, gaspillage de ressource versus respect des contraintes temporelles, dû à l'approximation du coût de la préemption de Liu et Layland.

Par ailleurs, en raison de la nature critique des systèmes temps réel auxquels nous nous intéressons nous avons recherché des conditions d'ordonnançabilité nous permettant de dire si un système est ordonnançable avant d'exécuter l'algorithme d'ordonnancement.

Parce que nous visons le prototypage rapide de systèmes temps réel embarqués avec le logiciel SynDEx les algorithmes d'ordonnancement temps réel multiprocesseurs proposés dans cette thèse produisent, bien sûr, des ordonnancements respectant les contraintes, mais aussi s'exécutent de manière la plus rapide afin de répondre à des problèmes industriels réalistes.

Enfin nous avons testé les résultats théoriques implantés dans le logiciel SynDEx sur une applications réaliste.

Plan général de la thèse

Le manuscrit se compose de deux parties : une première partie théorique et une deuxième partie expérimentale. Les plans détaillés des deux parties sont donnés dans leurs introductions respectives.

La première partie décrit les travaux théoriques réalisés. À partir d'un état de l'art nous proposons des conditions d'ordonnançabilité pour savoir si un système est ordonnaçable ou non, c'est-à-dire s'il respecte les contraintes de précédence, de périodicité stricte et de latence. Nous avons proposé des heuristiques pour résoudre le problème d'ordonnancement multiprocesseur avec ce type de contraintes en se basant sur ces conditions. Afin d'évaluer ces heuristiques nous les avons comparées à un algorithme exact. De plus comme nous visons des systèmes temps réel embarqués dont les ressources sont limitées, nous avons étudié le problème d'équilibrage de charge et de mémoire et avons proposé un algorithme en conséquence qui s'ajoute aux heuristiques précédentes.

La deuxième partie décrit les développements logiciels qui sont une concrétisation des travaux théoriques réalisés dans la première partie. Nous y exposons les nouvelles fonctionnalités offertes par la nouvelle version du logiciel SynDEx dans lesquelles elles ont été intégrées. Ensuite nous détaillons l'application de suivi pour train virtuel de CyCabs (véhicule électrique automatique conçu par le projet IMARA à l'INRIA) qui a servi de test pour valider les nouvelles fonctionnalités du logiciel SynDEx.

Première partie

Étude théorique

Introduction

Cette première partie du manuscrit est consacrée aux résultats théoriques obtenus en analyse d'ordonnancement temps réel multiprocesseur avec contraintes de précédence, de périodicité et de latence. Ces trois contraintes contribuent à accroître la complexité du problème d'ordonnancement. En parcourant les travaux réalisés dans le domaine du temps réel multiprocesseur on s'est rendu compte que ces trois contraintes sont rarement réunies ensemble. Par conséquent, afin de résoudre ce problème nous avons choisi d'étudier séparément la périodicité d'une part et la latence d'autre part, avec aussi dans les deux cas des contraintes de précédence. L'ensemble des résultats obtenus est utilisé pour traiter l'ordonnancement avec les trois contraintes.

Le premier chapitre, consacré à l'état de l'art, présente une vue globale sur les systèmes temps réel embarqué et l'ordonnancement temps réel. Les différentes approches et les résultats existants nous ont permis de choisir le type d'approches le plus adapté à notre démarche.

Ensuite l'enchaînement des chapitres dans le restant de cette partie reflète la stratégie adoptée. Tout d'abord on s'intéresse, dans le deuxième chapitre, au problème d'ordonnancement avec contraintes de précédence et de périodicité. Ensuite, dans le troisième chapitre, on se tourne vers le problème d'ordonnancement avec contraintes de précédence et de latence. Finalement le quatrième chapitre propose des solutions à l'ordonnancement avec contraintes de précédence, de périodicité et de latence.

Dans chaque chapitre on s'est appliqué à mettre en évidence les informations relatives au modèle utilisé, la justification des approches utilisées, la complexités des algorithmes proposés ainsi que les études dont ils résultent.

Chapitre 1

État de l'art

1.1 Systèmes temps réel

1.1.1 Définition

Les systèmes temps réel sont des systèmes numériques qui permettent l'implantation d'applications où le respect des contraintes temporelles est la principale contrainte à satisfaire. On rencontre des applications temps réel dans de nombreux domaines : aéronautique, militaire, télécommunication, robotique, etc. Ces systèmes temps réel sont tout d'abord réactifs, parce qu'ils réagissent continûment au stimuli venant de leur environnement considéré comme externe au système. Un tel système pour qu'il fonctionne correctement doit obligatoirement réagir à chacun des stimuli qu'il reçoit. La réponse aux stimuli d'entrée ne dépend pas seulement des stimuli mais aussi de l'état du système quand les stimuli arrivent. L'interface entre un système temps réel et son environnement est constituée par un ensemble de périphériques d'entrée appelés capteurs servant à la collecte des signaux émis par l'environnement et de sortie appelés actionneurs fournissant à l'environnement les commandes du système de contrôle (figure 1.1). Ensuite la validité d'un système temps réel dépend non seulement des résultats logiques du traitement effectué mais aussi de l'aspect temporel de production de ces résultats. Ainsi un système temps réel doit satisfaire deux contraintes importantes :

- exactitude logique (exactitude des traitements) : calculer les bonnes sorties du système en fonction de ses entrées,
- exactitude temporelle : les résultats de calcul sont présentés au bon moment (un calcul juste mais hors délai est un calcul non utilisable). En d'autres termes, un retard sur la production d'un résultat est considéré comme une erreur qui peut entraîner de graves conséquences. Un système temps réel doit respecter des contraintes temps réel.

L'utilisation de l'informatique apparaît de plus en plus fréquemment dans les domaines où l'interaction avec l'environnement constitue une raison d'être essentielle du système. L'intérêt de recourir à l'informatique dans ce genre de système est dû au fait que les fonctionnalités offertes par celle-ci soient peu coûteuses, en termes de temps de développement, de temps et de coûts de fabrication, d'encombrement ou de poids, par rapport aux solutions purement électroniques ou

FIG. 1.1 – *Définition d'un système réactif temps réel*

mécaniques. De plus elle offre une grande flexibilité due à la programmation, et l'ajout de nouvelles fonctionnalités, la constitution de mises à jour, ou la personnalisation du produit se révèlent souvent être des tâches moins contraignantes.

On trouve aujourd'hui des systèmes informatiques temps réel dans les domaines de l'aéronautique, de l'aérospatiale, des transports ferroviaires, de l'automobile, de la robotique, des télécommunications, de l'électronique grand public, du contrôle de procédés industriels, de supervision de centrales nucléaires et même de gestion de salles de marchés ou de télé-médecine. Pour plusieurs de ces domaines, une des caractéristiques importantes est que le système informatique se voit confier une grande responsabilité en termes de vies humaines, de conséquences sur l'environnement et de conséquences économiques. On parle alors de *systèmes critiques*, qui sont alors soumis à des contraintes de fiabilité.

1.1.2 Classification des systèmes temps réel

En fonction de la criticité des contraintes temporelles, on distingue essentiellement deux types de systèmes temps réel :

Temps réel strict ou dur

La majorité des systèmes temps réel critiques est exclusivement constituée de traitements qui ont des contraintes temporelles strictes. C'est-à-dire que la condition indispensable de fonctionnement du système est que tous les traitements du système doivent impérativement respecter toutes leurs contraintes temporelles. On parle alors de traitements temps réel strict ou dur (hard en anglais). Ceci suppose deux choses : i) qu'on soit capable de définir les conditions de fonctionnement nominales en termes d'hypothèses sur l'environnement avec lequel le système interagit; ii) qu'on soit capable de garantir la fiabilité du système avant son exécution, c'est à dire que tous les scénarios d'exécution possibles dans ces conditions respecteront leurs contraintes temporelles. Ceci suppose à son tour qu'on puisse extraire ou disposer de suffisamment d'informations sur le système pour déterminer tous les scénarios possibles.

Temps réel souple

Une autre classe de systèmes est moins exigeante quant au respect des contraintes temporelles. Les systèmes de cette classe, dits temps réel souple (soft en anglais), peuvent souffrir un taux "acceptable" de fautes temporelles de la part d'une partie des traitements (eux-mêmes dits "temps réel souple"), et sans que cela ait des conséquences catastrophiques. Cette classe comprend, entre autres, les systèmes où la qualité est appréciée par les sens de l'être humain sous la forme d'un service: c'est le cas de systèmes et d'applications multimédia (téléphonie, vidéo, rendu visuel interactif par exemple). La mesure du respect des contraintes temporelles prend la forme d'une donnée probabiliste : la qualité de service relative à un service particulier (nombre d'images ou

nombre d'échantillons sonores rendus par secondes, par exemple), ou relative au comportement du système dans son ensemble (nombre de traitements qui ont pu être rendus dans les temps, tous services confondus, par exemple), ou les deux combinés. Une problématique de cette classe de systèmes est d'évaluer la qualité de service [1], avant ou pendant le fonctionnement, que le système offre ou va pouvoir offrir en cours de fonctionnement, en fonction des caractéristiques de l'environnement et du système.

Dans cette thèse on ne s'intéresse qu'aux systèmes temps réel strict.

1.2 Généralités sur l'ordonnancement temps réel embarqué

La problématique de l'ordonnancement temps réel revient à définir dans quel ordre exécuter des tâches sur chaque processeur d'une architecture donnée. Dans un système temps réel les tâches sont soumises à des contraintes temporelles et éventuellement à d'autres contraintes. Le but de l'ordonnancement temps réel est donc de prévoir avec le plus d'exactitude possible le comportement temporel d'un système. Il existe dans la littérature différents termes pour designer l'action d'associer un ensemble de tâches à un ensemble de processeurs. Dans cette thèse on utilise les termes "assignation" et "distribution" qui expriment l'idée d'associer une tâche à un processeur mais, à la différence avec le mot "ordonnancement", ne lui attribuent pas un ordre précis par rapport aux autres tâches.

1.2.1 Caractéristiques d'une tâche temps réel

Une tâche est un ensemble d'instructions destinées à être exécutées sur un processeur. La caractérisation d'une tâche peut varier d'un modèle d'ordonnancement à l'autre et suivant la nature de celle-ci. Parmi les paramètres les plus utilisés on peut citer (tous ces paramètres sont illustrés sur la figure 1.2) :

- $R(t_i)$ (Ready time ou Release time) : c'est la date à laquelle la tâche t_i peut commencer son exécution, elle est appelée aussi date de demande d'activation,
- $S(t_i)$ (Start time), $E(t_i)$ (End time) : sont respectivement la date à laquelle la tâche t_i est exécutée sur le processeur appelée aussi la date de début d'exécution et la date à laquelle la tâche t_i finit son exécution appelée aussi la date de fin d'exécution,
- $RT(t_i)$ (Response time) : ceci représente $E(t_i) - S(t_i)$,
- $C(t_i)$ (Computing time) : c'est la durée d'exécution d'une tâche t_i. Ce paramètre est considéré dans la majorité des travaux sur l'ordonnancement temps réel comme le pire cas des temps d'exécution (WCET pour Worst Case Execution Time) sur le processeur où elle va être exécutée. Le WCET représente une borne supérieure du temps d'exécution c'est à dire que la tâche peut se terminer plus tôt. Pour être valable, la valeur de ce paramètre ne doit pas être trop surestimée et doit être sûre (jamais dépassée) ;
- $D(t_i)$ (Deadline) : c'est l'échéance ou la date au plus tard. Elle représente l'instant auquel l'exécution d'une tâche doit être terminée et dont le dépassement provoque une transgression

de la contrainte temporelle. Deux types de deadlines existent :

- relative $D(t_i)$: l'échéance est relative au release time de la tâche,
- absolue $D(t_i) + R(t_i)$: l'échéance est définie avant l'exécution,

– $l(t_i)$ (Laxity) : c'est la laxité d'une tâche t_i, qui représente le temps restant avant l'occurrence de sa date de début d'exécution ou de reprise au plus tard. Si ce paramètre vaut zéro à un instant donné, la tâche correspondante doit être impérativement exécutée à cet instant et sans interruption sinon. son échéance sera inévitablement dépassée.

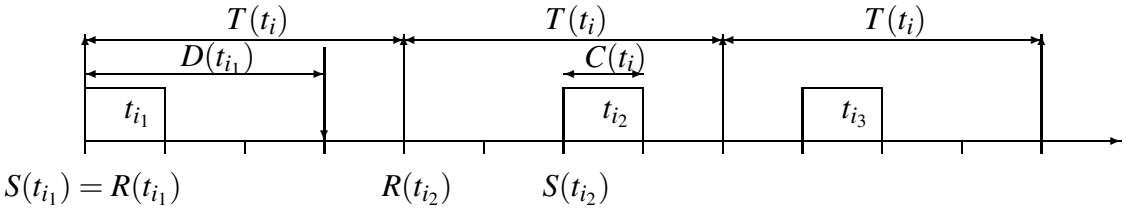

FIG. 1.2 – *Représentation graphique du modèle temps réel*

1.2.2 Nature des tâches

Une tâche peut s'exécuter à des intervalles réguliers (tâches périodiques) ou de manière aléatoire (tâches non périodiques).

1.2.2.1 Tâches périodiques

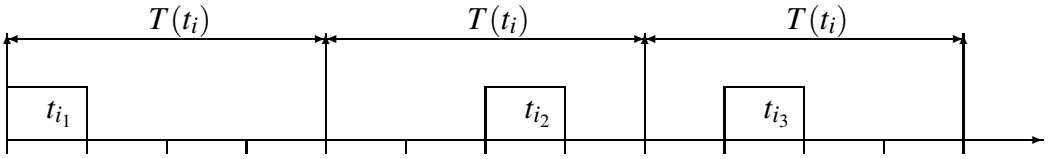

FIG. 1.3 – *Périodicité classqiue*

On dit que la tâche t_i est périodique de période $T(t_i)$ si l'événement qui conditionne l'activation de cette tâche se produit à des intervalles de temps réguliers $T(t_i)$. Donc le service fourni par cette tâche périodique est rendu indéfiniment. Pour cette raison, chacune de ces exécutions est appelée instance. Les figures 1.2 et 1.4 représentent deux types de tâches périodiques : i) le premier type

est ce qu'on peut appeler la période classique puisque c'est le plus utilisé dans les applications temps réel. Chaque instance de la tâche t_i doit s'exécuter entièrement à l'intérieur de l'intervalle de longueur t_i qui démarre à la date d'activation de cette instance (voir la figure 1.3); ii) le deuxième type est appelé période stricte. À la différence de la période classique, les instances d'une tâche périodique stricte t_i doivent démarrer exactement au début de chaque intervalle de longueur $T(t_i)$ (voir la figure 1.4). On peut remarquer que la période stricte est inclue dans la période classique et que dans ce cas $R(t_i) = S(t_i)$. Nous y reviendrons avec plus de détails dans le chapitre 2.1.4.

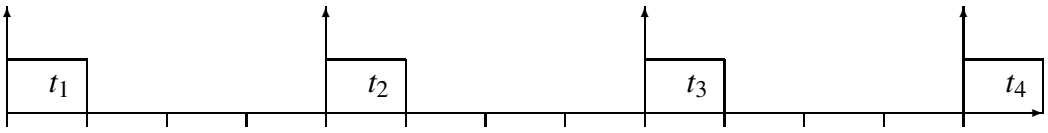

FIG. 1.4 – *Périodicité stricte*

1.2.2.2 Tâches non périodiques

Il existe deux types de tâches non périodiques :
apériodique : les dates d'activations sont aléatoires et ne peuvent être anticipées. Son exécution est le produit d'événements internes ou externes qui peuvent se déclencher à tout instant. Dans [2] différents procédés, pour ordonnancer l'exécution de tâches apériodiques d'un système temps réel, sont exposés.
sporadique : c'est un cas particulier des tâches apériodiques où une durée de temps minimale sépare deux activations successives. Pour les prendre en compte, ces tâches sont souvent considérées comme tâches périodiques [3] pour appliquer les résultats existant des tâches périodiques.

Ces deux types de tâches n'est pas étudiée dans cette thèse.

1.2.2.3 Tâches concrètes/non-concrètes

Si un scénario d'activation particulier est imposé aux tâches on dit qu'elles sont concrètes. Par ailleurs les tâches sont non-concrètes si les dates de première activation ne sont pas connues à priori.

1.2.2.4 Approches synchrones/asynchrones

Dans le cas synchrone un scénario d'activation est imposé et reproduit à l'identique pendant toute la vie du système, tandis que dans le cas asynchrone une hypothèse sur les activations des tâches est prise en compte.

1.2.2.5 Dépendance et précédence

En général les automaticiens et traiteurs de signal spécifient fonctionnellement les systèmes temps réel embarqué avec des blocs diagrammes décrivant des fonctions qui sont soit indépendantes, soit dépendantes dans le sens où une fonction produit des données pour une autre fonction qui les consomme. Une dépendance impose une précédence entre la fonction qui produit et celle qui consomme. Les fonctions deviennent des tâches dès qu'on les a caractérisées temporellement. Une contrainte de précédence entre deux tâches impose un ordre entre ces tâches [4]. Il faut noter que cette contrainte de précédence est la plupart du temps (donc pas toujours) due au fait que les deux fonctions sont dépendantes. Le terme "précédence" désigne un ordre entre deux tâches sans qu'il y ait transfert de données alors que "indépendance " signifie qu'un ordre n'est pas imposé entre deux tâches. Par ailleurs deux tâches sont dites "dépendantes" si l'une a besoin du résultat d'exécution de l'autre pour pouvoir être exécutée à son tour, ce qui impose, bien évidemment, une précédence entre ces deux tâches. On appelle la tâche qui produit les données "la tâche productrice" (ou "prédécesseur") et la tâche qui consomme les données "la tâche consommatrice" (ou "successeur"). La dépendance entre les tâches peut être modélisée par un graphe orienté où les noeuds représentent les tâches et les arcs les relations de dépendance entre les tâches. On distingue deux types de dépendances :

- sans perte de données : les données produites par l'exécution des instances de la tâche productrice sont toutes consommées par les instances de la tâche consommatrice,
- avec perte 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.

Dans cette thèse on suppose que les dépendances entre les tâches sont sans perte de données, c'est-à-dire qu'une tâche ne peut entamer son exécution que lorsqu'elle dispose de toutes les données produites par ses prédécesseurs. On étudiera le cas qu'on voit souvent dans la réalité et selon lequel les tâches dépendantes sont soit à la même période ou à des périodes multiples. Supposons que t_a et t_b sont deux tâches dépendantes alors soit : i) $T(t_a) = T(t_b)$ qui signifie que chaque instance de t_a produit une donnée pour chaque instance de t_b; ii) $T(t_b) = qT(t_a)$ qui signifie que q instances de t_a produisent les q données pour une instance de t_b. Dès lors, les tailles des mémoires allouées pour chaque tâche et qui prennent en compte ces transferts de données sont connues d'avance. Ces restrictions sur les périodes des tâches dépendantes permettent de satisfaire la condition de dépendance sans perte de donnée.

1.2.2.6 Latence

Imposer une contrainte de latence entre les deux tâches t_a et t_b équivalente à $L(t_a,t_b)$ signifie imposer que la durée de temps qui s'écoule entre la date de début d'exécution de la tâche t_a et la date de fin d'exécution de la tâche t_a ne doit pas dépasser $L(t_a,t_b)$. L'intérêt de borner le temps de calcul entre ces deux tâches est de garantir que le temps de réponse - le temps total que met une tâche pour s'exécuter entièrement - de la deuxième tâche ne dépassera jamais une certaine valeur critique à partir de laquelle les performances du système ne seraient plus satisfaisantes ou à partir de laquelle le système serait instable [5].

1.2.2.7 Makespan

Dans les systèmes temps réels chaque sortie peut être fonction de plusieurs entrées. Le temps total d'exécution, que l'on appellera dans la suite "makespan" car c'est le terme utilisé dans la littérature [6], reflète le temps qui s'écoule entre la date de début d'exécution de la première tâche exécutée et la date de fin de la dernière tâche tâche exécutée. L'objectif est de développer des algorithmes qui, en plus du respect des autres contraintes temporelles, minimisent le makespan. Prenons l'exemple d'une application de chronomètre électronique, l'environnement est constitué par des boutons de mise en marche, d'arrêt, de remise à zéro et d'un afficheur. Le système informatique qui contrôle cet environnement est un système temps réel. En effet, l'action "arrêter le comptage du chronomètre" déclenchée par l'événement "stop" (pression sur un bouton) doit être accomplie en un temps inférieur à la précision recherchée sur la mesure sous peine d'aboutir à un résultat erroné. Ici un temps de réaction trop long conduit à un mauvais fonctionnement de l'application. Un autre exemple, le cas du véhicule détecteur d'obstacles. La détection tardive d'un obstacle peut conduire à une collision susceptible d'engendrer des dégâts.

1.2.3 Classes des problèmes d'ordonnancement temps réel

- Monoprocesseur/multiprocesseur : si l'architecture ne dispose que d'un seul processeur on dit que le problème d'ordonnancement est monoprocesseur. Si plusieurs processeurs sont disponibles alors le problème est multiprocesseur ;
- Préemptif/non-préemptif : si les tâches du systèmes peuvent être préemptées alors le problème est préemptif. La préemption se traduit par la suspension temporaire de l'exécution d'une tâche au profit d'une autre tâche plus prioritaire par exemple ;
- En-ligne/hors-ligne (off-line/on-line): on dit qu'un ordonnancement est hors-ligne si la séquence d'ordonnancement est établie avant le lancement de l'application pour être répétée à l'infini. À l'exécution, cette approche a l'avantage d'avoir un sur-coût minimal sachant que pour être efficace tous les paramètres des tâches doivent être connues. Par ailleurs, ceci la rend inefficace en cas de changement de l'environnement [7]. Dans les systèmes complexe l'ordonnancement hors-ligne est souvent le seul moyen de prédiction de la satisfaction ou pas des contraintes temporelles imposées. Des techniques performantes de résolution de ce types de problèmes (mais qui sont très coûteuse en temps), comme les algorithmes optimaux, peuvent être utilisés. On dit qu'un ordonnancement est en-ligne si il se fait au fur et à mesure que les tâches arrivent dans le système. À tout moment (quelque soit l'événement) l'algorithme d'ordonnancement est capable de traiter des tâches qui n'ont pas été activé auparavant en utilisant les paramètres des tâches propres à cet instant. Ce qui le rend flexible et capable de s'adapter aux changements de l'environnement. Cependant comme ce choix doit être fait le plus rapidement possible afin de d'influencer le moins possible l'ordonnancement, l'utilisation d'heuristiques sous-optimales est préférée à celle d'algorithmes optimaux ;
- oisif/non oisif : en général, un processeur exécute la tâche la plus prioritaire dès qu'elle est prête à être exécutée et qu'il ne peut pas la retarder s'il n'a rien d'autre à faire. On dit, dans ce cas, que l'ordonnancement est non oisif, c'est-à-dire qu'il fonctionne sans insertion de temps creux (non-idling ou work-conservative en anglais). Dans certains cas, un système

peut n'être ordonnançable qu'en n'exécutant aucune tâche pendant un certain temps. On dit, alors, que l'ordonnancement est non oisif, c'est à dire qu'il fonctionne sans insertion de temps creux (non-idling ou work-conservative en anglais) ;

- Statique/dynamique : un ordonnancement statique (prédictif) est basé uniquement sur les paramètres des tâches au départ (avant l'activation). À l'inverse, un ordonnancement dynamique (réactif) est basé sur les paramètres des tâches qui varient au cours de l'exécution ;
- Optimal/non optimal : un algorithme d'ordonnancement est dit optimal [8] pour un problème donné si il permet de trouver un ordonnancement qui respecte toutes les contraintes lorsqu'un tel ordonnancement existe. Si l'algorithme optimal ne trouve pas de solution alors aucun autre algorithme ne pourra le faire. Par ailleurs, un algorithme d'ordonnancement non-optimal vise à trouver des solutions approchées (en ignorant une partie des contraintes par exemple).

1.2.4 Non-préemptif vs préemptif

Cette section compare les systèmes de tâches non-préemptives avec les systèmes de tâches préemptives et expose nos motivations concernant le choix non-préemptif.

Même si la plupart des algorithmes d'ordonnancement sont préemptifs, il existe des situations où l'ordonnancement non-préemptif est préférable. Par exemple le surcoût engendré par la préemption n'est pas toujours négligeable par rapport aux durées d'exécutions des tâches et aux temps de communication entre processeurs. Le système peut comprendre des sections critiques où la préemption n'est pas permise. Cette situation est gérée, par exemple, par l'utilisation du protocole de plafonnement de priorité [9] alors qu'en non-préemptif tout cela est inutile. De plus, pour la majorité des systèmes embarqués, la préemption est trop chère en temps et espace à cause du coût lié au changement de contexte. Ce coût, il n'est soit pas pris en compte soit pas maîtrisé. Pour toutes ces raisons nous avons opté pour des ordonnancements non-préemptifs tant qu'on ne maîtrise pas totalement le coût de la préemption. Dans [10] Meumeu et Sorel s'intéressent d'une part à la formalisation du problème d'ordonnancement en définissant plusieurs représentations permettant de décrire tous les ordonnancements possibles en fonction de toutes les durées d'exécution possibles des tâches du système, ceci en prenant en compte le coût exact de la préemption, et d'autre part cherchent à obtenir de nouveaux résultats d'ordonnançabilité qui améliorent les résultats connus actuellement dans le cadre de l'ordonnancement temps réel classique.

1.2.5 Analyse d'ordonnançabilité

Pour avoir une certaine prédictibilité et pouvoir s'assurer du respect des contraintes temporelles, en particulier quand il s'agit d'un système temps réel strict, une analyse de l'ordonnancement doit être effectuée.

Ainsi l'étude de l'ordonnancement des systèmes temps réel se focalise principalement sur deux problèmes :

- la faisabilité : en considérant les tâches, leurs contraintes ainsi que les processeurs dont dispose l'architecture, le but est de dire s'il existe un ordonnancement qui satisfait toutes les

échéances;

– l'ordonnançabilité : en considérant les tâches, leurs contraintes, les processeurs dont dispose l'architecture ainsi qu'un algorithme d'ordonnancement, le but est de déterminer s'il existe un ordonnancement, qu'on obtient avec cet algorithme, respectant toutes les contraintes.

L'analyse d'ordonnançabilité peut être extrêmement difficile pour des systèmes complexes. C'est pourquoi, il se peut qu'un modèle de système complexe soit transformé en un autre modèle plus simple dont l'analyse est connue. Cependant cette pratique ne permet d'obtenir que des résultats partiels (par exemple une condition nécessaire et suffisante d'ordonnançabilité en monoprocesseur devient une condition nécessaire en multiprocesseur [11]).

Nous présentons dans la suite les principales approches pouvant être employées pour l'analyse d'ordonnançabilité.

1.2.5.1 Approche analytique

L'analyse par approche analytique consiste à : i) identifier le ou les pires cas d'exécution pour une tâche dans un système donné; ii) produire une expression analytique, une condition d'ordonnançabilité ou/et de faisabilité, permettant d'évaluer les caractéristiques du système. Ces conditions, généralement en temps polynomial ou pseudo-polynomial, peuvent être des conditions suffisantes ou des condition nécessaires et suffisantes dépendant des paramètres des tâches [12].

1.2.5.2 Simulation

La simulation est couramment utilisée pour l'analyse des performances, le dimentionnement ou la mise au point des systèmes temps réel et l'analyse d'ordonnancement. Elle consiste à modéliser le comportement du système à l'aide d'un formalisme adéquat puis à l'exécuter avec un outil de simulation sur des scénarios définis par l'utilisateur. Les informations résultantes peuvent être de simples traces de l'exécution du système ou des évaluations sur certains comportements du système. L'un des principaux inconvénients de la simulation est qu'elle peut être optimiste [13]. En effet, comme il est difficile de réaliser une étude exhaustive de tous les scénarios possibles, elle peut montrer qu'un système n'est pas ordonnançable (en montrant un scénario non ordonnançable) , mais elle ne peut prouver qu'un système est ordonnançable (même si tous les scénarios testés le sont).

Des outils de simulations existent et sont utilisés dans l'industrie, parmi les quels on peut citer Cheddar [14].

1.2.5.3 Model-checking

Le model-checking est une méthode permettant l'exploration exhaustive de l'espace d'états du système. Ces méthodes permettent de vérifier de nombreuses propriétés sur les systèmes autres que l'ordonnancement [15]. Le formalisme utilisé est souvent celui des systèmes de transitions étendus temporellement : automates, réseaux de Petri. Il existe des extensions spécifiques aux systèmes temps réel comme les réseaux de Petri étendus à l'ordonnancement [16]. L'inconvénient majeur,

comme on peut le prévoir, est que l'espace d'états que les méthodes de model-cheking doivent prendre en compte a une taille exponentielle, par conséquent, ceci limite la taille des modèles analysables.

1.3 Systèmes embarqués

Dans cette thèse les applications qui nous intéressent sont temps réel et aussi embarquées. Ceci nous contraint à prendre en considération les propriétés de ces systèmes dans les différents travaux que nous menons.

1.3.1 Définition

Un système embarqué est un système intégré dans un système plus large (d'où l'appellation "embedded systems" en anglais) avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières (contrôle, surveillance, communication). Beaucoup de systèmes embarqués sont temps réel, ils sont en interaction continue avec leur environnement (le monde physique), ils doivent prendre certaines décisions ou effectuer certains calculs en respectant une contrainte temporelle. Une faute temporelle peut entraîner une catastrophe dans le système.

1.3.2 Architectures pour systèmes embarqués

De nos jours, de plus en plus de systèmes embarqués nécessitent pour leur fonctionnement plusieurs processeurs qui peuvent être de types différents. Par exemple un téléphone mobile GSM contient au moins deux processeurs, un processeur de traitement de signal (DSP) pour le calcul et le codage/décodages des données radio et un processeur principal pour exécuter les différentes applications (IHM, gestion répertoire, jeux, etc.). Afin de pouvoir fonctionner de manière autonome, un système embarqué doit disposer de l'ensemble minimum des éléments physiques d'un système : processeur(s), mémoire(s), entrées/sorties, ainsi que d'un programme qui contrôle ces différents éléments. Il doit également disposer d'une source d'énergie : générateur ou batteries.

Hormis la classification monoprocesseur/multiprocesseur des architectures, il existe différents classements des architectures multiprocesseur. On peut les classer en :

- (homogène/hétérogène) par rapport à la nature des processeurs dont dispose l'architecture :
  - homogène : dans ce cas les processeurs sont identiques .i.e. ils sont interchangeables et ils ont la même capacité de calcul;
  - hétérogène : les processeurs sont soit indépendants .i.e. les processeurs ne sont pas destinés à exécuter les mêmes tâches, ou uniformes .i.e. les processeurs exécutent les mêmes tâches mais chaque processeur à sa propre capacité de calcul.
- ou (homogène/hétérogène) suivant la nature des communications entre processeurs :
  - homogène : si les coûts de communication entre chaque paire de processeurs de l'architecture sont toujours les mêmes;

- hétérogène : si les coût de communication entre processeurs varient d'une paire de processeurs à une autre.

– (parallèle/distribuée) selon le type de mémoire dont dispose l'architecture :

- parallèle : ce modèle d'architecture correspond à un ensemble de processeurs communicant par mémoire partagée;
- distribuée : il correspond à un ensemble de processeurs à mémoire distribuée communicant par messages.

Les architectures multiprocesseurs sont souvent représentées par un graphe où les sommets sont les processeurs. Si un arc relie deux sommets cela signifie que ces deux sommets peuvent communiquer directement par le biais du médium de communication (bus, mémoire, ...). Il est utile de noter que ce graphe peut avoir plusieurs topologie afin de satisfaire un but bien précis, parmi lesquelles on peut citer :

– la topologie en étoile : tous les processeurs communiquent entre eux à travers le même médium de communication 1.5. Cette configuration permet entre autre de réduire les câblages [5];

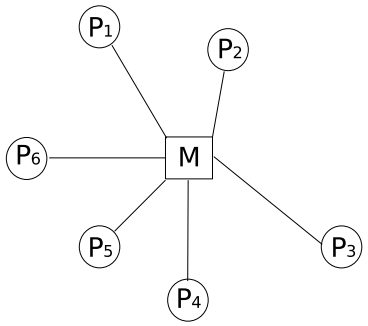

FIG. 1.5 – *Architecture en étoile*

– en graphe complet : chaque paire de processeurs possède son propre médium de communication 1.6. Cette configuration permet de minimiser les coûts de communication.

1.3.3 Contraintes d'embarquabilité

Ces contraintes sont la conséquence directe de la restriction de ressource dans les systèmes embarqués. Ces systèmes disposent d'une puissance de calcul ainsi que d'une mémoire, en générale, limités, que ce soit pour des raisons de poids, de volume, de consommation énergétique (véhicules autonomes), de durcissement aux radiations (nucléaire, spatial), ou de prix (applications grand public). Entre ces différentes contraintes, la gestion de la mémoire et la consommation énergétique sont les plus importantes et par conséquent les plus étudiées dans littérature. contraintes.

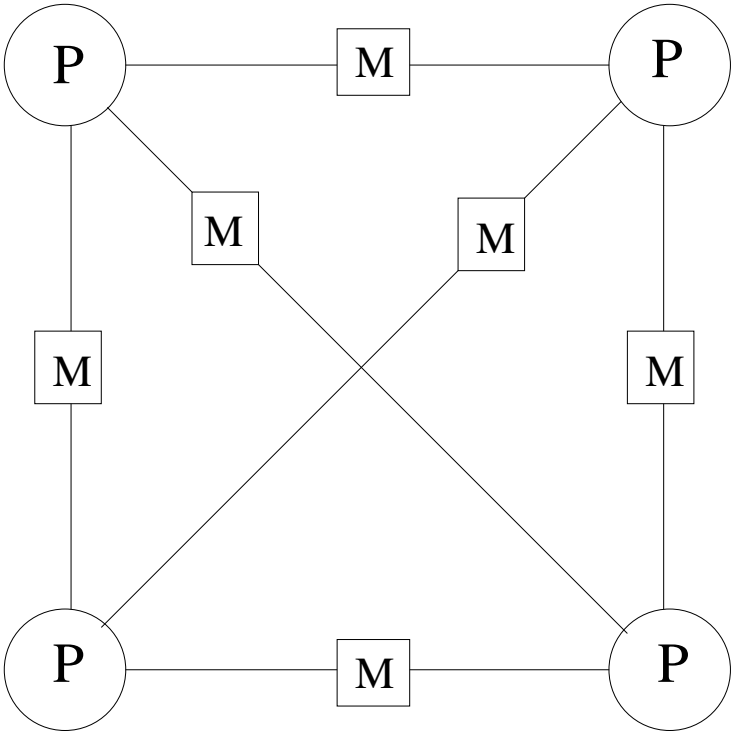

FIG. 1.6 – *Architecture en graphe complet*

1.3.3.1 Gestion de la mémoire

Les processeurs étant de plus en plus rapides, les mémoires leur semblent de plus en plus lentes, pour la simple raison que les performances de ces dernières progressent beaucoup plus lentement que les leurs. La mémoire disponible pour l'exécution des applications dans les systèmes embarqués a toujours été une ressource précieuse. En effet bien que la baisse des prix et l'augmentation des capacités mémoire des matériels soient devenue courants, il n'en demeure pas moins vrai que les besoins en mémoire de ces applications ne cessent d'augmenter [17] car ces applications sont de plus en plus complexes. On assiste donc à une course, probablement sans fin, entre augmentation des ressources et augmentation des besoins qui font que la gestion judicieuse de la mémoire reste un point délicat. Historiquement les premières techniques de gestion de mémoire ont été des techniques manuelles, qui consistaient à laisser l'utilisateur indiquer explicitement les allocations (par exemple, malloc en C) et surtout les restitutions (par exemple free en C). Les techniques automatiques, plus complexes à mettre en oeuvre, sont arrivées plus tard. L'utilisateur n'intervient que peu ou pas du tout dans cette fonction. Il existe de nombreux algorithmes d'allocation statique ou dynamique de mémoire [18]. Quelques études récentes visent à utiliser des techniques de gestion de mémoire élaborées, comme l'allocation et la libération automatique de la mémoire (garbage collector) ou encore l'espace d'adressage séparé, dans un contexte temps réel embarqué [19].

1.3.3.2 Consommation d'énergie

La maîtrise de la consommation d'énergie est un problème largement abordé dans le domaine des systèmes embarqués. Avec le temps elle est devenue un facteur essentiel pour leur conception vu leur complexité qui engendre une consommation énergétique de plus en plus élevée. La consommation d'énergie est un paramètre dont la prise en compte est essentielle pour le développement des application embarquées. Il existe différentes approches pour pallier ce problème mais qu'on ne cite pas pour la simple raison que ce n'est pas une contrainte qu'on s'est fixé de respecter au départ. Par ailleurs, si on parle de consommation c'est parce qu'il se trouve que le fait de minimiser le makespan d'un système multiprocesseur conduit à minimiser la consommation d'énergie. En effet, de fortes contraintes temps réel imposent l'utilisation de processeurs puissants basés sur des processeurs cadencés à des fréquences élevées pour aller le plus vite possible. Or en microélectronique, l'utilisation de fréquences élevées est synonyme de consommation électrique plus élevée ce qui va à l'encontre des contraintes de consommation d'énergie [5]. Le fait de minimiser le makespan permettrait d'utiliser de processeurs moins puissants et surtout moins gourmands pour des résultats équivalents [20, 21, 22, 5]. Par conséquent la répartition de la charge [23] constitue une bonne alternative au problème de la maîtrise de la consommation d'énergie puisqu'elle permet entre autre la minimisation du makespan (on reviendra dans la suite de l'état de l'art sur la notion de la répartition de la charge).

1.3.4 Autres contraintes

En plus des contraintes temporelles et d'embarquabilité il existe d'autres contraintes comme celles d'accès aux ressources ou de localité [24]. Si la ressource (processeur, mémoire, entrée/sortie, etc.) est partagée alors cela peut nécessiter d'ajouter des contraintes aux tâches pour éviter qu'une ressource critique ne soit utilisée simultanément par plusieurs tâches. Par ailleurs il est possible d'imposer une contrainte de localité signifiant qu'une tâche s'exécute exclusivement sur un processeur (par exemple pour bénéficier d'une ressource qui ne se trouve que sur ce processeur). Il existe aussi une contrainte d'anti-localité qui précisent si deux tâches doivent s'exécuter sur des processeurs différents (par exemple quand il s'agit d'introduire de la redondance spatiale dans les traitements afin que le système soit tolérant aux fautes physiques) [24].

1.4 Algorithmes d'ordonnancement et conditions d'ordonnançabilité

Dans cette section on fait l'état de l'art des algorithmes d'ordonnancement en précisant à chaque fois les contraintes qui sont prises en compte ainsi que les conditions d'ordonnançabilité si celles-ci existent.

Dans la littérature, les algorithmes d'ordonnancement se divisent selon les critères en plusieurs types. Par exemple suivant les priorités des tâches les algorithmes d'ordonnancement se divisent en trois types : i) les algorithmes a priorité fixe au niveau des tâches; ii) les algorithmes a priorité dynamique au niveau des tâches mais fixe au niveau des travaux (une tâche préemptive se divise

en plusieurs travaux); iii) les algorithmes a priorité dynamique au niveau des travaux. Les deux derniers algorithmes représentent le même algorithme si les tâches sont non-préemptives. On peut aussi distinguer deux types d'algorithmes d'ordonnancement suivant l'échéance des tâches : i) les algorithmes à échéance sur requête ($D = T$); ii) les algorithmes à échéance contrainte ($D < T$).

1.4.1 Monoprocesseur

Dès les premières recherches on constate que ce domaine a été largement étudié. Il existe de nombreux travaux solides depuis les premiers travaux décrits dans [25] qui ont fondé la théorie de l'ordonnancement. La littérature dans ce domaine est très riche et compte des algorithmes d'ordonnancement optimaux ainsi que des analyses d'ordonnançabilité très performantes (avec des complexités pseudo-polynomiales et même polynomiale). Dans la suite de la section on considère, tout d'abord, les algorithmes d'ordonnancement préemptif monoprocesseur de tâches périodiques sans contraintes de précédence ainsi que les conditions d'ordonnançabilité correspondantes, ensuite nous citons quelques résultats dans le cas non-préemptifs et sans contraintes de précédence.

• Priorité fixe

La propriété d'une tâche est la même pour toutes ses activations.

RM (Rate Monotonic)

Cet algorithme a été introduit par liu et layland en 1973 [25]. L'ordonnancement Rate monotonique est un ordonnancement préemptif à priorité statique qui s'applique à un ensemble de tâches périodiques indépendantes, et à échéance sur requête. Les priorités sont allouées aux différentes tâches de la façon suivante : plus la période de la tâche est courte, plus la priorité allouée sera importante. La contribution principale de cet algorithme est qu'il est optimal pour le modèle de liu et layland [25] (les tâches sont préemptives, périodiques à échéance sur requête et indépendantes) du point de vue de l'ordonnançabilité (si un ordonnancement qui satisfait les différentes contraintes existe il le trouve).

La condition suffisante d'ordonnançabilité avec l'algorithme RM pour un système de n tâches est [25] :

$$\sum_{i=1}^{i=n} \frac{C(t_i)}{T(t_i)} \leq n(2^{\frac{1}{n}} - 1)$$

Par la suite une condition nécessaire et suffisante a été introduite par [26]. Elle montre que si un ensemble de tâches périodiques triées par priorités décroissantes est ordonnancé par RM et dont le temps de réponse $RT(t_i)$ d'une tâche t_i est borné supérieurement par la solution de l'équation $RT(t_i)^{q+1} = (\sum_{j=1}^{i-1} \lceil \frac{RT(t_i)^q}{T(t_j)} \rceil . C(t_j)) + C(t_i)$ (cette équation peut être résolue récursivement où $\lceil x \rceil$ est la partie entière supérieure de x), alors l'ordonnancement de cet ensemble est faisable si et seulement si : $\forall i_{(1 \leq i \leq n)} RT(t_i) \leq T(t_i)$.

RM n'est plus optimal dans le contexte non-préemptif.

DM (Deadline Monotonic)

Cet algorithme a été introduit par Leung et Whiteheaden en 1982 [27] pour des tâches à échéances contrainte, c'est-à-dire qu'elles respectent la caractéristique suivante : $T(t_i) \geq D(t_i) \geq C(t_i)$. Plus $D(t_i)$ est petit plus la tâche t_i est prioritaire. DM est optimal pour le modèle de liu et layland [25] (les tâches sont préemptives, périodiques et indépendantes) du point de vue de l'ordonnançabilité.

La condition suffisante d'ordonnançabilité de n tâches périodiques (triées par priorités décroissantes) $\{t_1,...,t_i,...,t_n\}$ avec DM est :

$$\forall i : 1 \leq i \leq n : C(t_i) + \sum_{j=1}^{i-1} \lceil \frac{D(t_i)}{T(t_j)} \rceil . C(t_j) \leq D(t_i)$$

De la même manière que pour RM, il existe une condition nécessaire et suffisante pour l'algorithme DM. Si un ensemble de tâches périodiques triées par priorités décroissantes est ordonnancé par DM et dont le temps de réponse $RT(t_i)$ d'une tâche t_i est borné supérieurement par la solution de l'équation $RT(t_i)^{q+1} = (\sum_{j=1}^{i-1} \lceil \frac{RT(t_i)^q}{T(t_j)} \rceil . C(t_j)) + C(t_i)$, alors l'ordonnancement de cet ensemble est faisable si et seulement si : $\forall i_{(1 \leq i \leq n)} RT(t_i) \leq D(t_i)$.

DM n'est plus optimal en non-préemptif en général, sauf pour le cas suivant : $\forall (i,j) C(t_i) \leq C(t_j) D(t_i) \leq D(t_j)$.

• Priorité dynamique

La propriété d'une tâche est définie pour chaque activation de cette tâche.

EDF (Earliest Deadline First)

Cet algorithme a été introduit tout comme RM par Liu et Layland en 1973 [25]. C'est un ordonnancement qui peut être préemptif ou non-préemptif à priorité dynamique qui s'applique à des tâches périodiques indépendantes et à échéances sur requêtes. Le principe d'allocation de priorités dans cet algorithme est que la tâche dont l'échéance absolue arrive le plus tôt aura la priorité la plus élevée. Les priorités sont réévaluées, si nécessaire, au cours du temps. Cette réévaluation est effectuée lorsque, par exemple, une nouvelle tâche arrive et que son échéance est la plus proche comparée à celles des autres tâches prêtes à être exécutées. EDF est optimal pour résoudre le problème d'ordonnancement préemptif monoprocesseur à priorité dynamique des tâches indépendantes périodiques et à échéance sur requête. De plus, il a été démontré plus tard que cet algorithme reste optimal si les tâches ne sont pas périodiques.

La condition nécessaire et suffisante d'ordonnançabilité dans le cas préemptif si $\forall i : D(t_i) = T(t_i)$ (échéance sur requête) est :

$$\sum_{i=0}^{i=n} \frac{C(t_i)}{T(t_i)} \leq 1$$

Dans [35] George et al. ont prouvé dans le cas synchrone que pour n'importe quel ensemble de tâches non-préeptives $\{t_1,...,t_i,...,t_n\}$ tel que $\sum_{i=0}^{i=n} \frac{C(t_i)}{T(t_i)} \leq 1$ est ordonnançable en utilisant EDF si et

seulement si ρ est un instant donné dans le scénario synchrone de l'exécution tel que $\nexists i : D(t_i) > \rho$ alors :

$$\forall \rho \in \mathbb{S},\ \rho \geq h(\rho)$$

Avec :

- $\mathbb{S} = \bigcup_{i=1}^{n} \{kT(t_i) + D(t_i), k \in \mathbb{N}\} \cap [0, L[$,
- h la demande processeur qui représente la durée cumulée d'exécution des tâches,
- $\mathbb{L}$ la plus longue période d'activité du processeur obtenue dans le scénario synchrone (toutes les tâches démarrent en 0).

LLF (Least-Laxity First)

Cet algorithme se base sur la laxité. LLF a été introduit par Mok et Dertouzos [28, 29]. À chaque invocation, LLF élit la tâche dont la laxité est la plus faible. L'ouvrage [30] montre que les conditions d'ordonnançabilité pour l'algorithme LLF sont les mêmes que pour EDF. c'est-à-dire que la condition nécessaire et suffisante d'ordonnançabilité dans le cas préemptif si $\forall i : D(t_i) = T(t_i)$ (échéance sur requête) est :

$$\sum_{i=0}^{i=n} \frac{C(t_i)}{T(t_i)} \leq 1$$

L'algorithme LLF présente l'inconvénient, lorsque plusieurs tâches possèdent la même laxité, d'engendrer un grand nombre de changements de contexte ce qui explique qu'il soit aussi peu utilisé dans le cas monoprocesseur [31].

Autres travaux

- En présence de tâches non-concrètes [32] prouve que la résolution de n'importe quel problème d'ordonnancement de tâches périodiques est NP-difficile au sens fort (cette notion est définie dans la section 1.5) dans le contexte oisif.
- Jeffay et al [33] ont montré que déterminer la faisabilité d'un système de tâches périodiques non-préemptives sur un processeur est un problème NP-difficile au sens fort. Ils proposent, également, une condition nécessaire d'ordonnançabilité dans la cas de tâches périodiques non-concrètes sans précédence. On dit qu'une tâche est concrète si sa date de début d'exécution est connue à l'avance. Cette condition est la suivante :
  Soit $\{t_1,...,t_i,...,t_n\}$ un ensemble de tâches périodiques triées dans un ordre croissant de période (i.e., pour chaque paire de tâches (t_i,t_j), si $i > j$, alors $T(t_i) \geq T(t_j)$). Si les tâches de cet ensemble sont ordonnançables alors :

  1. $\sum_{i=1}^{n} \frac{C(t_i)}{T(t_i)} \leq 1$
  2. $\forall i, 1 < i \leq n; \forall Q,\ T(t_1) < Q < T(t_i)$:
     $Q \geq C(t_i) + \sum_{j=1}^{i-1} \left\lfloor \frac{Q-1}{T(t_j)} \right\rfloor C(t_j)$.

– Par contre si les tâches périodiques sont concrètes alors déterminer leurs ordonnançabilité sur un processeur est un problème NP-difficile (cette notion est définie dans la section 1.5) au sens fort. Cai et Kong [34] ont consolidé ce résultat en montrant que le problème demeure NP-difficile au sens fort même si les tâches ont les mêmes dates d'arrivées, et si les périodes sont harmoniques (c'est-à-dire que chaque tâche a une période qui divise toutes les périodes supérieures à celle-ci).
– Dans [35] George et al. ont montré que l'optimalité de l'algorithme d'ordonnancement de priorités commandées (Fixe Priority Driven) proposé par [36] dans le contexte non-préemptif est valable dasn le cas non-préemptif.

1.4.2 Multiprocesseur

Dans cette section on considère les systèmes constitués d'une architecture à plusieurs processeurs qui communiquent entre eux. Un état de l'art de ce problème est présenté comprenant les problèmes liés à l'ordonnancement dans ce contexte ainsi que les différentes approches pour les résoudre.

Le problème d'ordonnancement multiprocesseur a été formulé pour la première fois par Liu en 1969 [37]. Lorsqu'on aborde l'ordonnancement multiprocesseur, les premiers résultats qu'on peut obtenir sont : i) l'absence d'algorithmes d'ordonnancement optimaux ayant une complexité polynomiale. A part quelques cas particuliers, tous les problèmes d'ordonnancement sont de complexité NP-complet (plus de détails sont proposés dans 1.5); ii) les solutions aux problèmes d'ordonnancement multiprocesseur ne sont certainement pas de triviales extensions des solutions monoprocesseur [38].

Il existe en général deux types d'approches pour résoudre le problème d'ordonnancement multiprocesseur :

– par partitionnement : trouver un partitionnement de l'ensemble des n tâches en m sous-ensembles (m étant le nombre de processeurs), puis d'ordonnancer chaque sous-ensemble sur un processeur différent. Les tâches ne sont pas autorisées à migrer d'un processeur à un autre ;
– globale : il s'agit d'appliquer sur tous les processeurs une stratégie unique d'ordonnancement et de faire en sorte qu'à chaque instant, les m tâches les plus prioritaires soient attribuées aux m processeurs (m étant le nombre de processeurs). Dans ce cas on autorise la migration, en d'autres termes une tâche peut commencer son exécution sur un processeur, être préemptée, et la reprendre sur un autre processeur.

Dans la plupart des cas les approches par partitionnement et globales ne peuvent pas être opposées ou comparées. Dans [27] Leung et Whitehead ont montré que pour des cas d'ordonnancement multiprocesseur de tâches périodiques, un ordonnancement est trouvé à l'aide de l'approche par partitionnement alors qu'aucun algorithme du type approche globale n'a été capable de le réaliser. D'un autre côté, il y a des systèmes multiprocesseur de tâches périodiques ordonnançables avec l'approche globale mais pour lesquels aucun algorithme de partitionnement n'existe.

Entre les deux approches citées plus haut, celle qui est la plus appropriée pour l'ordonnancement non-préemptif est le partitionnement. Trouver un partitionnement est équivalent au problème

du Bin-packing, c'est-à-dire à déterminer la manière de diviser l'ensemble des tâches en un nombre de sous-ensembles égal au nombre des processeurs. Les tâches qui appartiennent au même sous-ensemble doivent être ordonnançables sur le même processeur. Une fois le partitionnement terminé il ne reste plus qu'à ordonnancer chaque sous-ensemble sur un processeur tout en respectant les précédences, les périodes des tâches et la minimisation du makespan.

Voici une synthèse de ce qu'on peut trouver dans la littérature en ce qui concèrne l'ordonnancement multiprocesseur avec contrainte de période.

- Il est important de noter que certains algorithmes d'ordonnancement monoprocesseur ont leurs variantes multiprocesseur. Prenons comme exemple l'algorithme EDF multiprocesseur qui est employé avec les deux approches d'ordonnancement multiprocesseur, en globale dans [39] et par partitionnement dans [40, 41]. On trouve aussi des versions améliorées de cet algorithme comme l'algorithme proposé dans [42] qui ordonnance toutes les tâches périodiques que EDF peut ordonnancer et ordonnance en plus les tâches périodiques qui manquent leurs échéances avec EDF.
- De nombreuses recherches ont été menées sur l'ordonnancement multiprocesseur selon l'approche globale. Ces recherches sont basées sur le concept de l'impartialité proportionnée (proportionate fairness). L'algorithme d'ordonnancement proportionate fair (Pfair) [43, 44] est le seul algorithme optimal (et de complexité polynomiale) d'ordonnancement multiprocesseur connu, cependant, excepté ces résultats théoriques cet algorithme n'est pas applicable dans la réalité. En effet ce modèle considère qu'à tout moment les tâches occupent une portion donnée et connue du processeur qui leur est réservée et ces tâches sont préemptées très souvent pour garantir l'optimalité, ce qui conduit à un sur coût prohibitif du système [45].
- Les travaux de Sun et Liu [46] ciblent les applications "job-shop" dans les systèmes temps réel multiprocesseur. Dans le cas du job-shop chaque tâche est une chaîne d'opérations, chacune d'entre elles nécessite l'utilisation d'exactement une machine. Sun et Liu fournissent des bornes aux temps de réponse des tâches. Audsley [36] et Audslay et al. [47] proposent une méthode d'analyse d'ordonnançabilité basée sur l'analyse du temps de réponse des tâches où les tâches sont autorisées à s'exécuter à des dates irrégulières. Tindell [48] et Tindell et Clark [49] appliquent la précédente analyse à des systèmes où les dépendances entre les tâches sont prises en compte.
- Dans, [50] propose une analyse d'ordonnançabilité avec EDF non préemptif et détermine une condition suffisante (plutôt qu'une condition nécessaire et suffisante vu la complexité du problème).
- Récemment, dans [51], Fisher et Baruah examinent les principaux résultats monoprocesseur préemptif pour des éventuelles extensions au cas multiprocesseur. Ils proposent des conditions d'ordonnançabilité pour l'algorithme EDF-global.

Si on met de coté la contrainte de périodicité des tâches, on s'oriente vers un problème d'ordonnancement multiprocesseur de tâches dépendantes avec comme objectif la minimisation du makespan. Ce problème est dit classique puisqu'une large documentation traitant cet ordonnance-

ment existe dans la littérature [52]. Il ne nécessite pas d'analyse d'ordonnancement car il existe toujours une solution. Ce problème d'ordonnancement, qui est NP-difficile [6] a été, en général, résolu suivant trois méthodes qui utilisent un graphe acyclique orienté (DAG) pour représenter les dépendances entre les tâches en associant des poids aux arcs (symbolisant les coûts de communication entre processeurs) qui relient les sommets (représentant les tâches à exécuter), ainsi que des poids aux sommets (symbolisant les durée d'exécution des tâches). Ces trois stratégies sont :

- la stratégie du chemin critique [53, 53, 54] : le chemin critique est le chemin le plus long en termes de poids dans un graphe. Ce chemin part d'un sommet racine ou source (un sommet qui n'a pas de prédécesseur) et finit par un sommet feuille (un sommet qui n'a pas de successeur). L'objectif de cette méthode est de minimiser le makespan en se basant sur la longueur du chemin critique. La longueur d'un chemin est réduite en ordonnançant au moins deux de ses tâches sur le même processeur ;
- la stratégie de l'ordonnancement de listes (list scheduling) [55] : les algorithmes procédant suivant cette méthode attribuent des priorités aux tâches pour former une liste ordonnée de tâches. Par exemple un niveau de priorité élevé est attribué à la tâche qui a la somme des poids de ses prédécesseurs la plus élevée ou à la tâches qui n'a pas de prédécesseur. Les objectifs dépendent du type de priorité attribué, un objectif peut être la minimisation du makespan ;
- la stratégie de décomposition de graphes [56] : basée sur la théorie du même nom [57], cette stratégie décompose un graphe en plusieurs sous-graphes en gardant les mêmes contraintes de dépendance entre les tâches. L'objectif est de minimiser les dépendances entre les sous-graphes .

1.5 Complexités

1.5.1 Calcul de la complexité d'un algorithme

Dans cette section, on ne s'intéresse qu'aux algorithmes dont la terminaison est garantie, et on appelle complexité d'un algorithme son temps de calcul (autrement dit le temps nécessaire à son exécution). On sait que si ce temps de calcul est exprimé, par exemple, en secondes, il va dépendre de la machine utilisée (vitesse du processeur, temps d'accès à la mémoire, etc.) et décroît rapidement avec les progrès de la technologie. Pour se rapprocher d'une notion indépendante de la technologie, la complexité d'un algorithme est exprimée par le nombre d'instructions élémentaires exécutées pendant son exécution. Les complexités d'algorithmes calculées dans cette thèse sont du type "complexité au pire" qui représentent la complexité maximale que l'algorithme peut avoir. La notation O (grand-O) est utilisée pour borner la complexité d'un algorithme. Par exemple l'algorithme de recherche d'un élément dans une liste de taille n a une complexité $O(n)$ ce qui signifie que pour trouver cet élément le nombres d'instructions élémentaires à exécuter ne dépassera jamais n instructions.

1.5.2 Théorie de la complexité

Le but de la théorie de la complexité est la classification des problèmes de décision suivant leur degré de difficulté de résolution. En algorithmique, un problème de décision est une question mathématiquement définie portant sur des paramètres donnés sous forme manipulable informatiquement, et demandant une réponse par oui ou non. Ainsi, savoir si, étant donné les distances entre les villes d'une carte et une distance d, il existe un chemin passant par toutes les villes et de longueur inférieure à d, est un problème de décision. Tout problème d'optimisation peut se ramener à un problème de décision.

Dans la littérature, il existe plusieurs classes de complexité pour les problèmes de décision (notés problèmes dans la suite), mais les plus connues sont les suivantes :

- La classe P : c'est la classe des problèmes pouvant être résolus en un temps polynomial. C'est-à-dire qu'il existe un algorithme de complexité $O(n^k)$ pour un certain k qui le résout (nous verrons dans la suite comment calculer la complexité d'un algorithme). On appelle cette classe celle des problèmes dits faciles ;
- La classe NP : c'est l'abréviation pour "non deterministic polynomial time". Cette classe renferme tous les problèmes pour lesquels on peut associer à chacun d'eux un ensemble de solutions potentielles (de cardinal au pire exponentiel) tel qu'on puisse vérifier en un temps polynomial si une solution potentielle satisfait la question posée.

On parle aussi dans ce manuscrit de problèmes pseudo-polynomiaux (les algorithmes qui les résolvent sont de complexité pseudo-polynomiale). Si x est une instance d'un problème A et $\max(x)$ la valeur du plus grand entier utilisé dans l'instance x alors on dit que l'algorithme, résolvant le problème A, est de complexité pseudo-polynomiale si son temps d'exécution est polynomialement borné inférieurement par $|x|$ et supérieurement par $\max(x)$.

La théorie de la complexité s'intéresse à la classification de problèmes et aux frontières existant entre ces différentes classes. Elle étudie aussi les limites du calcul permettant de résoudre un problème donné. La problématique centrale, la plus connue, dans cette théorie est la fameuse question : (P$\neq$NP?). En d'autres termes, s'il est toujours facile de vérifier qu'une solution d'un problème NP est correcte, est-il tout aussi facile de lui trouver une solution que d'en trouver pour un problème P ?

Les problèmes NP-complets appartiennent à la classe NP et sont NP-difficiles. On qualifie de "NP-difficile" les problèmes d'optimisation dont le problème de décision correspondant est NP-complet. Pour démonter qu'un problème est NP-difficile on utilise une technique de réduction. Les techniques de réduction permettent d'affirmer en temps polynomial qu'un problème est aussi difficile qu'un autre problème connu pour être NP-difficile.

Pour finir les problèmes NP-difficiles au sens fort sont ceux qui appartiennent à la classe NP et restent NP-difficiles même si on diminue la taille de leurs paramètres.

1.5.3 Classes de complexité de quelques problèmes d'ordonnancement multiprocesseur

Le tableau suivant donne les complexités de quelques problèmes d'ordonnancement non-préemptif hors-ligne pour multiprocesseur [23]. L'objectif commun de ces algorithmes est de satisfaire la contrainte de précédence, si elle existe, ainsi que de minimiser le makespan. n et m sont respectivement le nombre de tâches et le nombre de processeurs.

Nombre de processeurs	Durée d'exécution d'une tâche	Contraintes de précédence	Complexité
m	toutes égales	arbre	polynomial $O(n)$
2	toutes égales	quelconques	polynomial $O(n^2)$
2	sans	quelconques	NP-complet
2	deux valeurs possibles	quelconques	NP-complet
m	toutes égales	quelconques	NP-complet
m	quelconques	arbre	NP-complet

1.6 Ordonnancement multiprocesseur non-préemptif avec contraintes de précédence, de périodicité stricte, et de latence

Après avoir introduit la notion de temps réel, de systèmes embarqués et les différent problèmes d'ordonnancement issus de la combinaison de ces deux domaines ainsi que les solutions proposées, dans cette section, on va se focaliser sur un problème particulier qui est l'ordonnancement multiprocesseur non-préemptif avec contraintes, de précédence, de périodicité stricte et de latence qui constitue le sujet de cette thèse. Le but est de trouver un algorithme qui respectent les différentes contraintes et en plus, comme les applications visées sont du type embarqué, cet algorithme doit pouvoir minimiser le makespan.

1.6.1 Contexte de l'étude

Dans cette section on parcourt les approches et les méthodes de résolution ayant servi à proposer des algorithmes d'ordonnancement traitant des problèmes similaires au notre. Ceci va nous permettre par la suite de choisir une approche et de l'appliquer à notre problème qui est l'ordonnancement hors-ligne multiprocesseur non-préemptif avec contraintes de précédence, de périodicité stricte, et de latence.

La première observation suite à notre recherche bibliographique, est que le problème d'ordonnancement tel que nous le traitons est très peu abordé dans la littérature. Les raisons de ce manque sont principalement :

– la multitude de contraintes qui sont associées à un problème d'ordonnancement (sous la contrainte de précédence) qui est comme on l'a vu dans la section 1.5, déjà, très difficile. En effet, en rajoutant les contraintes de périodicité stricte et de latence le problème devient encore plus difficile (NP-difficile au sens fort [58]). On se doit de rappeler que ces contraintes

ont une importance capitale dans les applications traitées au sein de notre équipe. Prenons comme exemple les transports, où une défaillance peut avoir des conséquences catastrophiques, telles que la mise en danger de vies humaines, de l'environnement et des pertes financières importantes. Dès lors on se devait de les prendre en compte ;

- la périodicité stricte est un cas particulier de la contrainte de périodicité classique que l'on trouve dans la littérature. On rappelle que les périodes sont dites strictes si pour une tâche t_a de période $T(t_a), \forall q \in N,\ (S(t_{a_{q+1}}) - S(t_{aq})) = T(t_a)$, où les tâches t_{aq} et $t_{a_{q+1}}$ sont les q^{eme} et $(q+1)^{eme}$ répétitions (instances) de la tâche t. Il est vrai que la présence de cette contrainte dans un système restreint ses chances d'ordonnançabilité [59]. La période qui est prise en compte par les travaux qu'on cite dans cet état de l'art est la contrainte de période classique, et non la période stricte, (même si on ne fait pas la distinction entre ces deux contraintes à chaque citation) [60, 61, 58, 62] ;
- la notion de contrainte de latence imposée entre deux tâches dépendantes quelconques dans un système a été récemment introduite par Cucu et Sorel [63]. Hormis les travaux réalisés par Cucu et Sorel [64] on trouve peu de travaux proposant des solutions pour respecter cette contrainte en dépit de son intérêt dans les systèmes temps réel embarqués ;
- la minimisation du makespan pour satisfaire les contraintes d'embarquabilité que l'on a vu précédemment. L'ordonnancement qu'on cherche à produire doit présenter un makespan aussi réduit que possible.

À partir de là on peut dire que le problème traité, avant d'être un problème d'optimisation (la minimisation du makespan), est un problème de décision (existe-t-il un ordonnancement qui satisfait toutes les contraintes ?). Même en monoprocesseur ce problème reste un problème très compliqué à résoudre [65].

L'intérêt de cette partie de l'état de l'art est de parcourir toutes les approches qui existent pour résoudre des problèmes du même type.

Suivant les contraintes temps réel et d'embarquabilité auxquelles on est confronté nous allons tenter de déterminer, parmi les approches existantes, celle dont l'exécution est rapide et qui permet, en plus de faire de l'optimisation, de résoudre des problèmes de décision.

Comme pour les problèmes d'optimisation combinatoire les algorithmes proposés se divisent globalement en deux catégories : i) algorithmes exacts ou optimaux; ii) algorithmes sous-optimaux. Dans la suite on va parcourir en détails chaque catégorie.

1.6.2 Algorithmes exacts ou optimaux

Un algorithme exact est un algorithme qui cherche, en parcourant tous les ordonnancements possibles, une solution faisable (l'ordonnancement où toutes les contraintes sont respectées), si jamais celle-ci existe. Par ailleurs si cet algorithme ne trouve pas de solutions, on en déduit que le problème n'a pas de solutions. Si le problème d'ordonnancement inclut une optimisation (par exemple la minimisation du makespan), la solution trouvée par l'algorithme exact est la meilleure solution possible.

Le problème que l'on traite, comme la plupart des problèmes d'ordonnancement avec contraintes, est NP-difficle au sens fort. C'est pourquoi ces algorithmes sont sans cesse améliorés afin de ré-

soudre des problèmes plus larges que ceux résolus précédemment. Chaque algorithme se distingue par sa façon d'explorer l'espace des solutions, cet espace englobe les bonnes comme les mauvaises solutions ainsi que celles qui respectent les contraintes comme celles qui ne les respectent pas. Cependant différentes techniques permettent d'éviter les zones qui ne contiennent pas les solutions recherchées.

Il existe plusieurs méthodes qui exploitent les principes cités ci-dessus et peuvent être utilisées pour réaliser des ordonnancements qui respectent les contraintes de précédence, de latence et de périodicité stricte.

Branch and Bound

La méthode du branch and bound (B&B) [66] (procédure par évaluation et séparation progressive) consiste à construire un arbre d'exploration qui énumère toutes les solutions. Ensuite, en utilisant certaines propriétés du problème en question, il faut trouver une manière intelligente d'explorer cet arbre. Cette méthode utilise une fonction qui permet de fixer une ou plusieurs bornes (tout dépend du problème traité) afin d'évaluer certaines solutions pour, soit les exclure, soit les maintenir comme des solutions potentielles. Bien entendu la performance d'une méthode de B&B dépend, entre autres, de la qualité de cette fonction (de sa capacité d'exclure des solutions partielles le plus tôt possible).

Chen et Yur [67], ont utilisé le B&B pour minimiser le makespan en prenant en compte les coûts de communications entre les tâches non-préemptives. Le modèle d'architecture pris en compte est une architecture hétérogène où tous les processeurs peuvent communiquer les uns avec les autres à travers les média de communication. Cependant ces algorithmes ne garantissent pas le respect des contraintes temps-réel. La borne inférieure présentée dans cet article a permis de réduire considérablement la complexité de l'algorithme ce qui le rend potentiellement exploitable pour résoudre des applications réelles.

Peng, Shin et Abdelzaher [68] appliquent le principe du B&B au problème d'allocation de tâches périodiques communicantes. [69] décrit un algorithme optimal du type BB d'ordonnancement de tâches sous contraintes de précédence et d'anti-localité (cette notion est évoquée dans la section 1.3.4).

Programmation par contraintes

C'est une méthode en pleine expansion qui permet de modéliser et de résoudre efficacement de nombreux problèmes [70]. Elle allie en effet la simplicité d'écriture de programmes et la déclarativité de la programmation dans un langage de haut niveau, à la puissance et l'efficacité du calcul sur des domaines spécifiques à l'aide d'outils particuliers et optimisés (résolveurs de contraintes). De nombreuses applications peuvent s'exprimer simplement et efficacement dans le cadre de la programmation par contraintes : problèmes combinatoires, ordonnancement, simulation, diagnostic, analyse financière, aide à la décision, etc. Ces applications utilisent bien sûr différents domaines de calcul, contraintes, et algorithmes de résolution, mais elles peuvent toutes s'exprimer naturellement dans un cadre uniforme et être mises en oeuvre efficacement dans les principaux langages de

programmation par contraintes existant actuellement. Le langage le plus répandu de la programmation par contraintes est le langage Oz [71].

Schild et wurtz [72] ont démontré que la programmation par contraintes constitue une méthode prometteuse pour traiter l'ordonnancement temps réel. Ces travaux ont été relayés par Thane et Larsson [73] qui expliquent comment utiliser cette méthode pour résoudre des problèmes d'ordonnancement multiprocesseur non-préemptif hors-ligne. Ekelin et Jonsson [74] proposent un ordonnancement basé sur la programmation par contraintes multi-objectifs (optimisation multi-critères). Ils appliquent la qualité principale de cette méthode qui consiste à définir ce qu'est une solution acceptable sans être obligé de savoir comment on aboutit à cette solution. Tout récemment dans [75], il est démontré aussi que cette méthode est complète, c'est-à-dire que si un problème n'a pas de solution, l'algorithme est capable de le prouver, et qu'elle peut s'adapter à n'importe quel modèle temps réel.

Autres méthodes

Pour résoudre le problème qui nous intéresse d'autres méthodes exactes existent comme celle de la programmation mathématique [76]. Davidovic et al. [77] présentent une formulation sous la forme d'un modèle de Programmation Linéaire en Nombres Entiers (PLNE) [78, 79] pour le problème d'ordonnancement avec des tâches dépendantes dans un système multiprocesseur homogène en présence de coûts de communications. Pour diminuer le nombre de contraintes ils utilisent une reformulation du problème à l'aide d'une procédure de Réduction de Contraintes. Ils ont résolu plusieurs exemples de petite taille du modèle reformulé avec CPLEX 8.1 [80]. Les bornes supérieures ont été calculées par des techniques VNS (Variable Neighborhood Search) [81] appliquées directement à une formulation basée sur la théorie des graphes. Les bornes inférieures ont été obtenues en résolvant des relaxations linéaires de la formulation PLNE.

L'approche dans [82] explore l'espace des solutions d'ordonnancement des groupes (clusters) de tâches au lieu des tâches elles-mêmes pour réduire l'espace de recherche. En plus l'exploration est dirigée par un algorithme qui commence par les parties ou la probabilité de trouver des solutions est la plus élevée. En revanche le regroupement des tâches en clusters engendre la non satisfaction de contraintes de certaines tâches puisque ce regroupement ne prend pas en compte les dépendances entre les tâches du même groupe (cluster).

[83] propose un algorithme d'ordonnancement selon l'approche par partitionnement de tâches préemptives périodiques et dépendantes. L'algorithme proposé est une variante d'un autre algorithme, développé par ces même auteurs, destiné à résoudre le problème du minimum K-coupe où les tâches ainsi que les processeurs sont représentés par des graphes (le problème de la K-coupe consiste à déterminer un ensemble d'arcs d'un graphe de telle sorte que la suppression de ces arcs sépare le graphe en exactement k composantes connexes). En considérant k comme étant le nombre de processeurs utilisés pour ordonnancer les tâches, la solution du K-coupe permet de résoudre le problème d'ordonnancement. Les auteurs ont montré que dans le cas particulier du 2-coupe avec 10 à 25 sommets et 4 à 6 arcs de dépendances incidents par sommet, leur algorithme trouve en moyenne des solutions à moins de 3% des solutions optimales et trouve la solution optimale 74% du temps.

1.6.3 Algorithmes approchés ou sous-optimaux

Malgré les progrès réalisés par les méthodes exactes, les problèmes susceptibles d'être résolus par ces méthodes sont néanmoins très restreints [84]. En effet les méthodes exactes ne sont pas applicables à cause du temps de calcul car pour des instances de grande taille le temps nécessaire à la résolution peut être très important et l'optimisation doit être réalisée dans un temps raisonnable pour l'être humain. Effectivement comme presque tous les problèmes d'optimisation combinatoire liés aux graphes [85, 84], l'ordonnancement souffre de son impressionnant espace de recherche [86]. Prenons comme exemple l'ordonnancement non-préemptif d'un graphe de n tâches sur un seul processeur. Ce problème, pourtant simple, a un espace de recherche exponentiel (2^n).

L'apparition de nouvelles méthodes, dites approchées, permet d'apporter des solutions satisfaisantes à de nombreux problèmes d'optimisation. Ces solutions sont, dans la plupart des cas, des solutions sous-optimales mais leur avantage majeur est leur temps de calcul qui les rend exploitables pour les problèmes de grandes tailles, comparées aux méthodes précédentes. Il existe plusieurs classifications pour ces méthodes parmi lesquelles on va citer les deux classifications les plus répandues :

- la première classification identifie deux classes majeures :
  - les heuristiques : ce sont des méthodes dédiées à un problème où la recherche est guidée par des astuces dépendantes de ce problème. Parmi les heuristiques existantes on peut citer les heuristiques gloutonnes (ces méthodes seront détaillées dans la suite);
  - les métaheuristiques : ce sont des méthodes génériques dont les principes généraux sont inspirés d'autres domaines et sont applicables à différents problèmes d'optimisation combinatoire. Parmi ces métaheuristiques on peut citer (le Recuit Simulé : Mécanique Statistique, Algorithmes Génétiques, Colonie de fourmis : Bioinformatique, Recherche Tabou : Intelligence artificielle,...). D'après Widmer [87] les caractéristiques des métaheuristiques sont les suivantes :
    - les métaheuristiques sont des stratégies qui permettent de guider la recherche d'une solution optimale,
    - le but visé par les métaheuristiques est d'explorer l'espace de recherche efficacement afin de déterminer des solutions (presque) optimales,
    - les techniques qui constituent des algorithmes de type métaheuristique vont de la simple procédure de recherche locale (qui sera détaillée dans la suite) à des processus d'apprentissage complexes,
    - les métaheuristiques sont en général non-déterministes et ne donnent aucune garantie d'optimalité,
    - les métaheuristiques peuvent contenir des mécanismes qui permettent d'éviter d'être bloqué dans des régions de l'espace de recherche,
    - les concepts de base des métaheuristiques peuvent être décrits de manière abstraite, sans faire appel à un problème spécifique,
    - les métaheuristiques peuvent faire usage de l'expérience accumulée durant la recherche de l'optimum pour mieux guider la suite du processus de recherche ;

– la deuxième classification fait référence au nombre de solutions parcourues par la méthode [88, 89]. On distingue :
    - les méthodes utilisant une population de solutions. Ces méthodes sont basées sur la notion de population [90]. Elles manipulent un ensemble de solutions en parallèle lors de chaque itération. Les algorithmes évolutionnaires, les algorithmes génétiques ou encore les algorithmes de colonies de fourmis sont des exemples de méthodes à base de population ;
    - les méthodes basées sur l'évolution d'une seule solution. elles sont basées sur la recherche locale qui fait évoluer une unique solution. Elles sont aussi appelées méthodes de trajectoire. La notion de voisinage, qui est l'ensemble des solutions obtenues à partir d'une solution donnée en effectuant un petit nombre de transformations simples, est alors primordiale. Il existe un grand nombre de méthodes de recherche locale. Les plus connues sont probablement la Recherche Tabou et le Recuit Simulé.

Au fil du temps, on s'est rendu compte que la mise en commun des principes de plusieurs métaheuristiques voire de méthodes exactes pouvait mener à l'élaboration d'heuristiques encore plus performantes. Cependant la mise au point d'une méthode basée sur diverses métaheuristiques devient difficile, car il faut tout d'abord sélectionner un sous-ensemble de principes qui devront être appropriés au problème à résoudre. On peut donc dire que bien des heuristiques actuellement développées sont un assemblage, plus ou moins judicieux, d'un nombre relativement restreint de principes de base [91]. Talbi [92] a proposé une taxonomie de l'hybridation en présentant différents niveaux possibles de combinaison entre les méthodes. Globalement, les méthodes hybrides combinent méthodes exactes et méthodes approchées [93] ou des méthodes approchées entre elles : par exemple la recherche locale combinée avec une métaheuristique [94].

La recherche locale

Elle est basée sur l'évolution itérative d'une solution unique. Le passage d'une solution à une autre se fait grâce à la définition de la structure de voisinage qui est un élément très important dans la définition de ce type de méthode. La structure de voisinage diffère d'un problème à un autre et consiste à spécifier un voisinage pour chaque solution. Par la suite une solution est remplacée par une meilleure solution située dans son voisinage. Ensuite se pose la question de l'adaptation d'une méthode de recherche locale pour permettre à celle-ci de parcourir l'espace de recherche malgré la présence de minimums locaux. Le principal inconvénient des méthodes de recherche locale est leur difficulté à s'extraire des optimums locaux, les empêchant ainsi d'aboutir à un optimum global. La plupart des méthodes de recherche locale utilisent une seule structure de voisinage ce qui rend très difficile l'optimisation de problèmes disposant de nombreux minimums locaux. Récemment différents travaux ont été publiés sur des méthodes de recherche locale combinant différentes structures de voisinage ou faisant varier l'ensemble des solutions accessibles via un mécanisme d'extension et/ou de restriction du voisinage. Ces méthodes se sont montrées très prometteuses sur différents problèmes tel que la tournées de véhicules. La combinaison de différentes structures de voisinage ou de mécanismes d'extension et de restriction du voisinage, peut permettre à la méthode de s'extraire plus facilement d'un optimum local et ainsi de devenir performante. [95, 96] sont des

exemples d'algorithmes basés sur la recherche locale résolvant des problèmes d'ordonnancement.

1.6.3.1 Métaheuristiques

Ce sont des méthodes d'optimisation globale qui requièrent un nombre important d'évaluations pour visiter les plus d'optimums locaux possible afin de trouver l'optimum global. Dans la suite on cite quelques une des métaheuristiques les plus utilisées.

Le recuit simulé

Le problème de l'ordonnancement de tâches temps réel dur dans un environnement multiprocesseur peut être vu comme un problème d'optimisation globale. Des études en thermodynamique ont permis de modéliser le refroidissement d'un morceau de métal (simulated annealing) qui a été utilisé pour résoudre des problèmes d'optimisation dans de nombreux domaines (en traitement d'images notamment). En effet il permet de trouver un optimum global pour un système donné avec la propriété de ne pas rester sur un optimum local. Cette méthode a donc également été appliquée dans le cadre de l'ordonnancement temps réel sur multiprocesseur. L'ordonnancement initial des tâches est effectué aléatoirement, puis on calcule une fonction d'énergie qui sert à caractériser l'état du système. C'est cette fonction que l'on cherchera à minimiser progressivement, en agissant sur l'ordonnancement des différentes tâches. L'ajout à cette fonction d'un facteur d'agitation aléatoire permet à l'algorithme de passer par des phases où sa valeur d'énergie diminue puis augmente pour diminuer par la suite jusqu'à l'obtention de la valeur minimale (problème des minima locaux). Cette fonction d'énergie peut être une combinaison de nombreux paramètres comme les temps d'exécution, les besoins en ressources et en mémoire, les durée de communications,... Du point de vue des performances, les résultats obtenus [97] sont remarquables puisqu'ils correspondent au résultat optimal trouvé par un algorithme exact, mais il s'agit de valeurs expérimentales sur des cas relativement simples, et pas encore sur des cas plus complexes.

L'algorithme présenté dans [98] ordonnance un système de tâches périodiques sur une architecture multiprocesseur. La dépendance et les communications prises en compte par cet algorithme n'existent qu'entre les sous-tâches (alors que les tâches, elles, sont indépendantes). La satisfaction de ces contraintes a nécessité de procéder à quelques modifications du recuit simulé de base.

La recherche tabou

Appelée aussi recherche avec tabous, cette méthode, introduite par Fred Glover [99], a suscité l'intérêt dans plusieurs domaines. Que se soit en informatique et réseaux numériques d'information ou en industrie, logistique et transport, la recherche tabou a été largement utilisée et ses résultats ont été plutôt satisfaisants. Des présentations et des évaluations d'heuristiques de recherche tabou pour l'ordonnancement multiprocesseur sont proposés par Thesen dans [100].

La recherche tabou est fondée sur des idées simples et efficaces à la fois. Elle combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes lui permettant de surmonter l'inconvénient des minima locaux, tout en évitant le problèmes de cycles. En effet elle possède une mémoire qui garde les mouvements déjà effectués pour éviter d'y revenir et d'être

piégée dans un cycle qui impliquera la convergence vers un optimum local. Cependant une fonction d'aspiration existe afin d'autoriser le retour vers une solution tabou. Cela permet d'explorer d'autres espaces de solutions et d'améliorer les solutions.

[101] propose une heuristique du type recherche tabou pour résoudre le problème d'ordonnancement multiprocesseur de n tâches sur m processeurs identiques. L'efficacité de cette heuristique vient du fait que la procédure d'insertion et de suppression des mouvements tabous s'inspire de celle du problème du voyageur de commerce (le problème d'optimisation le plus connu et le plus traité [102]).

Algorithme génétique

Tout comme les algorithmes de colonies de fourmis [103] ou les algorithmes évolutionnaires [104], les algorithmes génétiques font partie des heuristiques d'évolution qui correspondent à des transformations et des coopérations entre les individus qui représentent chacun une solution.

Les principes fondamentaux de l'algorithme génétique ont été instaurés par John Holland [105]. Ces algorithmes s'inspirent du fonctionnement de l'évolution naturelle des espèces, notamment la sélection de Darwin et la procréation selon les règles de Mendel. Les individus, ou chromosomes d'un algorithme génétique, sont des codages des solutions possibles du problème. Comme dans la nature, ces individus forment une population qui va évoluer dans le temps selon des lois de sélection qui vont favoriser les mieux adaptés à se croiser afin de produire des populations meilleures. L'évolution des individus d'une population à une autre se fait à l'aide de la reproduction (problème des minima locaux). Les individus parents se reproduisent pour engendrer des individus fils qui vont être améliorés grâce à la mutation et le croisement. Les reproductions, les mutations et les croisements se font avec une part de hasard. En effet les parents candidats à la reproduction ou les fils candidats au croisement et à la mutation sont choisis d'une manière probabiliste proportionnelle à leurs qualités (par exemple faire reproduire des parents de bonne qualité pour espérer obtenir des fils de qualité encore meilleure). En définitive les individus, d'une génération à une autre, forment une progéniture plus performante qui se rapproche de la solution optimale (par exemple privilégier la reproduction entre les individus de bonne qualité favorise l'obtention d'individus fils de meilleure qualité). L'un des avantages de l'algorithme génétique est que, même si nous n'avons aucune idée de la solution du problème ni de la façon de la construire, la population initiale peut être générée aléatoirement. Par contre, si des solutions initiales sont connues des individus de départ sont créés à partir de celles-ci. Les algorithmes génétiques ont été appliqués aux différents problèmes d'ordonnancement de différentes manières [106, 107]. Pour ne parler que des cas où les tâches sont périodiques, dans [107] Nossal présente un algorithme d'ordonnancement multiprocesseur de tâches périodiques dépendantes, fondé sur l'algorithme génétique. Comme on a pas besoin de savoir comment construire une solution, Nossal se concentre sur l'encodage du problème, i.e., la représentation du problème par les individus et la fonction de sélection.

Colonie de fourmis

Les fourmis ont la particularité d'employer, pour communiquer, des substances volatiles appelées phéromones. Elles utilisent les pistes de phéromone pour marquer leur trajet, par exemple

entre le nid et une source de nourriture. Une colonie est ainsi capable de choisir (sous certaines conditions) le plus court chemin vers une source à exploiter [108], sans que les individus aient une vision globale du trajet. La méthode de la colonie de fourmis simule le comportement de ces insectes qui, lorsque l'on pose un obstacle sur leur trajet, trouvent toujours le chemin le plus court pour le contourner.

Comparée aux autres métaheuristiques citées précédemment, l'optimisation par colonie de fourmis a été moins pratiquée dans le cas de l'ordonnancement multiprocesseur. Néanmoins on trouve dans la littérature des travaux qui se basent sur cette méthode comme dans [103] où il est question d'un problème d'ordonnancement multiprocesseur avec contrainte de précédence et de ressource. Cette méthode permet aussi de prédire le nombre de processeurs nécessaires à l'ordonnançabilité du système.

Optimisation par essaim particulaire

L'optimisation par essaim de particules est une technique d'optimisation globale stochastique développée par [109]. Elle s'inspire du comportement social des individus qui ont tendance à imiter les comportements réussis qu'ils observent dans leur entourage, tout en y apportant leurs variations personnelles. Il existe un ouvrage complet sur les racines sociales de cette métaheuristique et les techniques mathématiques mises en oeuvre [110]. L'optimisation par essaim de particules utilise une population de solutions potentielles (particules) qu'elle fait évoluer à l'aide des échanges d'informations (essaim) entre particules. En fait, chaque particule ajuste sa trajectoire à partir de sa meilleure position dans le passé et vers la meilleure position des particules de son voisinage [111]. La variante globale de ces algorithmes considère la totalité de l'essaim comme voisinage. Les particules profitent ainsi des découvertes et expériences antérieures de toutes les autres particules.

Dans le domaine de l'ordonnancement multiprocesseur temps réel l'optimisation par essaim particulaire a été essentiellement hybridée avec d'autres méthodes approchées. Dans [109] Kong et al. présentent un algorithme résultant de l'hybridation de l'optimisation par essaim particulaire et du recuit simulé. [112] propose une hybridation de cette méthode avec la recherche locale pour résoudre le problème d'ordonnancement des tâches dans les systèmes multiprocesseurs. Les tests réalisés ont démontré l'efficacité de cet algorithme hybride. Dans [113] Boeringer et Werner prouve que l'optimisation par essaim particulaire est aussi performante que l'algorithme génétique mais en étant plus simple à coder.

Comparaison des métaheuristiques

La comparaison des métaheuristiques pose un problème nouveau qui n'est pas encore résolu de manière satisfaisante. En effet ces heuristiques sont itératives, ce qui veut dire que plus longtemps on les laisse travailler meilleures sont les solutions qu'elles produisent. Elles sont également presque toujours non déterministes car elles utilisent un générateur pseudo-aléatoire. Cela signifie que si on les exécute plusieurs fois on observera des solutions différentes. Il est clair que la comparaison d'heuristiques itératives doit se faire sur la qualité des solutions produites en fonction de l'effort de calcul consenti. Ce dernier est traditionnellement un temps d'exécution sur une machine donnée. Malheureusement cette mesure est à la fois imprécise et variable. En effet le temps de

calcul [91] est fortement dépendant du style de programmation, du compilateur et de ses options, de la charge de calcul au moment où les tests sont réalisés, etc.

Dans le cas où le problème combinatoire consiste à optimiser une fonction unique, il est facile de comparer la qualité de deux solutions. Talbi et Mountean [114] proposent, pour le problème d'ordonnancement multiprocesseur, une étude comparative de trois algorithmes qui sont : le recuit simulé, l'algorithme génétique et la méthode de descente [115](une variante de la recherche locale). Les critères de performance considérés sont la qualité de la solution et le temps de recherche pris par l'algorithme. Par ailleurs si on est confronté à un problème, comme c'est le cas dans ce manuscrit, qui combine la satisfaction de contraintes et l'optimisation (minimisation du makespan) les solutions qu'on doit comparer, selon la longueur du makspan, sont uniquement celles qui respectent toutes les contraintes.

1.6.3.2 Algorithmes gloutons

Le plus gros inconvénient des algorithmes sous-optimaux, après le manque de connaissance sur la qualité de leurs résultats, est leur complexité. Cette complexité est, certes, loin d'atteindre celles des algorithmes exacts, néanmoins, pour les applications de grandes tailles le temps de calcul des algorithmes sous-optimaux demeure très long (plusieurs heures voire plusieurs jours). Tellement long, qu'il n'est pas envisageable d'utiliser ces algorithmes pour un ordonnancement en-ligne (), ni même pour un ordonnancement hors-ligne ou l'utilisateur ne dispose pas du temps nécessaire a ces algorithmes [116].

Un algorithme glouton donne très rapidement un résultat qui est rarement la solution optimale. Du coup les solutions produites par ces algorithmes sont utilisées par des méthodes plus lentes comme solutions initiales [117].

Le principe d'une méthode gloutonne est très simple. La solution est construite incrémentalement en rajoutant à chaque pas un élément selon un critère glouton, c'est-à-dire que celui qui paraît "localement" le meilleur (un choix à court terme). La construction de la solution est donc souvent très simple, le problème étant plutôt de justifier que la solution construite est une bonne solution.

La différence entre les algorithmes glouton et les algorithmes qu'on vient de voir est que la solution est construite directement sans être mise en cause, c'est-à-dire que toutes les décisions prises au cours de l'algorithme glouton sont définitives et on ne revient jamais en arrière. De ce fait, une seule branche de l'arbre représentant l'espace des solutions est parcourue contrairement aux autres algorithmes qui se servent de techniques d'exploration des voisinages des solutions.

L'algorithme EDF multiprocesseur qu'on a vu dans la section 1.4.1 est un algorithme glouton destiné à résoudre des problèmes d'ordonnancement de tâches préemptives périodiques. Cependant EDF peut être utilisé dans le cas non préemptif mais l'optimalité n'est plus assurée pour des tâches périodiques concrètes et les conditions d'ordonnançabilité ne sont plus valables [118].

1.6.3.3 Algorithmes d'approximation

Dans ce cas on est capable de caractériser ou d'évaluer les solutions qu'on obtient en utilisant de tels algorithmes relativement aux solutions que proposeraient des algorithmes optimaux. C'est l'objectif des algorithmes d'approximation qui sont la plupart du temps du type glouton. Ainsi,

pour un problème d'optimisation, on ne peut pas garantir l'obtention systématique d'une solution optimale avec une telle méthode, mais on pourra affirmer que la solution fournie par un algorithme d'approximation ne peut pas être au-delà d'un certain pourcentage d'une solution optimale, tout en conservant un temps de calcul dont la croissante est limitée par un polynôme en la taille des données considérées (le nombre de tâches et le nombre de processeurs).

Un algorithme d'approximation A produit en temps polynomial une solution de valeur $A(i)$ proche de l'optimum. Un algorithme d'approximation a une garantie de performance α (qu'on appelle aussi algorithme α-approché) si : quelque soit l'instance *ins* du problème appartenant à l'ensemble des solutions I alors $max(\frac{A(ins)}{OPT(ins)}, \frac{OPT(ins)}{A(ins)}) \leq \alpha$ ($\frac{A(ins)}{OPT(ins)}$ correspond à un prolème de minimisation et $\frac{OPT(ins)}{A(ins)}$ à un problème de maximisation). $OPT(i)$ est la valeur de la fonction objectif pour la solution optimale (la fonction objectif désigne l'expression qu'on recherche l'extremum).

L'importance de concevoir de tels algorithmes s'est imposée à mesure que la conjecture (P≠NP) devenait plus vraisemblable. La définition d'algorithmes d'approximation a été posée formellement par Garey, Graham, Ullman et Johnson [119, 120].

Plusieurs problèmes d'ordonnancement multiprocesseur de tâches périodiques sont présentés dans [121]. Après avoir prouvé qu'ils sont NP-difficiles, Korst et al. proposent des algorithmes d'approximation pour les résoudre.

1.6.4 Approche probabiliste

Estimant que le raisonnement déterministe est limité et ne suffit pas à résoudre tous les problèmes d'ordonnancement et d'analyse d'ordonnançabilité, des chercheurs se sont tournés vers des approches probabilistes [122]. Dans la littérature l'approche probabiliste est largement utilisée dans le cas temps réel souple mais on la trouve aussi dans le temps réel dur, dit probabiliste, où il n'est pas impératif de respecter toutes les deadlines mais où des garanties probabilistes qui s'approchent des 100 % suffisent [123]. En ordonnancement multiprocesseur où la complexité du problème est extrême, comme dans [124] l'analyse stochastique produit des résultats intéressants et surtout elle considère tous les scénarios possibles d'exécution ce qui permet, entre autre, de connaître les raisons de la non ordonnançabilité.

1.7 Réduction de la consommation d'énergie et gestion de la mémoire

Le problème considéré dans cette thèse est, comme on l'a vu, un problème compliqué à résoudre. D'autre part on a vu qu'il existe des contraintes liées à l'aspect embarqué des applications prises en compte comme la consommation d'énergie ou encore la gestion mémoire. Quasiment aucun des algorithmes cités auparavant ne considèrent ces deux critères et la raison est toute simple : l'ajout de critères supplémentaires à un problème difficile le rend encore plus difficile. Pourtant ces deux critères jouent un rôle capital dans le bon fonctionnement des applications embarquées et il faut bien qu'à un moment ou un autre elles soient prisent en compte.

Dans la section 1.3.3.2 il est dit que la minimisation du makespan dans un système multiprocesseur est synonyme de réduction de la consommation d'énergie. En même temps un des moyens les plus efficaces pour y parvenir est la distribution de la charge sur les processeurs. D'un autre côté le problème de la gestion de la mémoire consiste à adapter une application ainsi que la quantité de mémoire dont elle a besoin pour s'exécuter et la quantité de mémoire dont dispose le système où cette application est exécutée. Les tâches doivent être ordonnancées de telle façon que, tout au long de l'exécution de l'application, chaque processeur puisse avoir la quantité mémoire nécessaire pour exécuter les tâches qui y ont été ordonnancées.

La solution au problème qu'on se pose dans cette section serait, une fois que l'heuristique d'ordonnancement respectant toutes les contraintes temps réel ait été exécutée, on exécute un algorithme d'équilibrage (répartition) de charge et de gestion mémoire. En plus d'être très rapide cet algorithme doit effectuer une optimisation bi-critère (équilibrage de charge et de mémoire pour minimiser le makespan et une gestion efficace de la mémoire).

1.7.1 Équilibrage de charge

Une bonne utilisation des machines parallèles nécessite une répartition de la charge que l'application va engendrer au cours de son exécution, de façon à pouvoir la distribuer équitablement sur chaque processeur de l'architecture. La charge (somme des durées d'exécution) se définit en fonction de la nature de l'application. La stratégie utilisée pour le placement des tâches à exécuter a un impact direct sur les performances d'exécution de l'application. Il est de ce fait très important, pour des raisons d'efficacité, de pouvoir gérer l'allocation de la charge produite par l'exécution d'une application sur les différents processeurs de l'architecture : c'est le problème d'équilibrage de charge.

Ce problème a toujours suscité un vif intérêt dans différents domaines de recherche. De nombreuses définitions et taxonomies de modèles d'équilibrage de charge ont été proposées dans la littérature [23, 125, 126, 127].

Les algorithmes d'équilibrage de charge se distinguent par leurs objectifs. Ils peuvent être utilisés dans un système pour répondre à différents critères d'efficacité. Leur utilisation la plus courante consiste à optimiser l'utilisation des ressources de calcul (le partage de la mémoire par exemple) [128] et/ou la minimisation du makespan (par équilibrage de charge) [129] d'une application. Cet objectif peut être complété par le respect des contraintes temporelles dans le cadre d'applications temps réels [130].

1.7.2 Optimisation multicritère

L'optimisation multicritère a pour but de résoudre des problèmes d'optimisation en présence de critères d'optimalité multiples. À la fin du 19ème siècle, l'économiste Vilfredo Pareto formule le concept d'optimum de Pareto [131] qui constitue les origines de la recherche sur l'optimisation multicritère. Dans un problème multicritère, il existe un équilibre tel que l'on ne peut pas améliorer un critère sans détériorer au moins un des autres. Cet équilibre a été appelé optimum de Pareto. Le choix d'une solution parmi l'ensemble des solutions acceptables n'est pas évident et dépend de la personne en charge du problème (décideur) qui est souvent amené à prendre une décision

de compromis parmi tous les objectifs fixés. Talbi [132] décrit la relation entre le décideur et le mécanisme de résolution d'un problème multicritère en fonction de l'implication du décideur dans le choix des solutions.

Un problème d'optimisation multicritère se pose sous la forme suivante : $Min(F_1(x \in X),...F_n(x \in X))$ (dans le cas d'une minimisation) où x est une solution (un vecteur de nombres réels ou entiers dans le cadre de la programmation mathématique) et X est l'ensemble des solutions réalisables alors que chaque F_i correspond à un critère i différent parmi les n critères qu'on cherche à optimiser.

A priori un tel problème est mal défini puisque, sauf cas très particulier, on ne peut pas simultanément optimiser plusieurs critères à la fois. Par essence même une optimisation multicritère n'a pas de sens mathématique car l'expression mathématique précédente traduit plutôt des outils et méthodes permettant de trouver une ou plusieurs solutions de compromis. Pour ce faire les notions de dominance et d'efficacité sont fondamentales.

Une solution efficace ou Pareto-optimale est une solution non dominée par aucune autre solution. L'ensemble des solutions non-dominées est appelé ensemble efficace ou front de Pareto [133].

Concernant les méthodes de résolution du problème d'optimisation multicritère, il est très difficile de générer un ensemble de solutions non dominées. Jusqu'à maintenant, une méthode largement utilisée est la méthode de pondération des objectifs qui consiste à hiérarchiser le problème multicritère. On définit à priori des coefficients de pondération pour toutes les fonctions-objectif et on remplace une fonction vectorielle par une fonction scalaire représentée par une somme des objectifs pondérés. Cette méthode est très intuitive et permet d'obtenir une seule solution du problème multicritère. Afin de générer plusieurs solutions, il faudrait utiliser plusieurs vecteurs de coefficients de pondération.

Il est possible d'utiliser quelques unes des méthodes de résolution citées dans dans la section 1.6.3 pour traiter ce problème. Les méthodes exactes font partie des candidats plausibles pour des problèmes de petites tailles. Côté heuristiques, les algorithmes évolutionnaires sont considérés comme les méthodes les plus efficaces et de loin les plus utilisées [134, 135]. L'avantage principal de cette approche est le fait que les algorithmes évolutionnaires n'utilisent pas un seul point mais toute une population de points dans l'espace de solutions admissibles pendant le processus itératif d'optimisation. Ainsi à la fin des itérations, on obtient un ensemble de solutions du problème d'optimisation multicritère.

Faire appel à l'optimisation bi-critère pour résoudre des problèmes d'ordonnancement avec la gestion de la mémoire et la minimisation du makespan [136] ou encore d'autres contraintes telle la fiabilité [137] est très courant dans la littérature.

Dans ce chapitre on a tenter de faire un état de l'art aussi complet que possible sur le problème d'ordonnancement des systèmes temps réel ainsi que les différentes approches capables de les résoudre. Dans le chapitre suivant nous allons reprendre quelques une des termes introduit pour énoncer notre modèle ainsi que les méthodes de résolution les plus adaptées à notre problématique.

Chapitre 2

Ordonnancement temps réel multiprocesseur avec contraintes de précédence et de périodicité

Dans ce chapitre on s'intéresse à l'ordonnancement temps réel multiprocesseur avec contraintes de précédence et de périodicité. Les tâches considérées sont non-préemptives et les coûts de communication inter-processeurs sont pris en compte. Ce même problème a été traité par Cucu dans [58] où plusieurs résultats sont présentés. Toutefois nous fûmes dans l'impossibilité de reprendre ces résultats pour des raisons de suppositions non réalistes comme celle où le nombre de tâches est égal au nombre de processeurs par exemple.

On a vu dans le premier chapitre, section 1.5, que ce problème est NP-difficile au sens fort et qu'il existe plusieurs approches susceptibles, soit de le résoudre, soit de chercher des solutions approchées. Par ailleurs les problèmes qui nous intéressent sont liés aux systèmes embarqués (voir la section 1.2) et les algorithmes les résolvant, pour les besoins du prototypage rapide (cette question est largement abordée dans la deuxième partie du manuscrit), doivent avoir un temps d'exécution court. C'est pourquoi le premier critère sur lequel l'heuristique est évaluée est le temps qu'elle met pour s'exécuter (en fonction de la taille des données du problème en entrée). De plus, en raison de la contrainte de période, on est confronté à un problème de décision qui consiste à déterminer si une solution au problème existe car comme on peut le lire dans l'état de l'art (voir la section 1.4.2) la solution n'existe pas toujours. Seuls les algorithmes exacts sont capables d'affirmer qu'un problème, tel qu'on le considère, est ordonnançable ou pas, leurs solutions sont fiables à 100%. Donc l'autre critère sur lequel cette heuristique est évaluée est la qualité des résultats obtenus en les comparant aux résultats d'un algorithme exact. Enfin deux contraintes sont prises en compte : la précédence et la minimisation du makespan.

Considérant tous ces éléments du problème, les principaux objectifs qu'on s'est fixés sont :

– tout d'abord, parmi les approches citées dans l'état de l'art, les heuristiques gloutonnes (voir la section 1.6.3.2) sont celles qui produisent des ordonnancements en un temps court même si c'est, parfois, au dépend de la qualité de la solution. Par conséquent l'heuristique, au départ, sera du type glouton, c'est-à-dire à la fois rapide et de bonne qualité. Il est évident que proposer une heuristique très rapide mais qui en revanche ne trouve jamais d'ordonnance-

ments est un exercice plutôt facile. Ainsi, un des défis de cette thèse, est de proposer une heuristique gloutonne très rapide et astucieuse à la fois (en se basant sur des conditions rapides à vérifier) pour pouvoir garantir un certain taux de fiabilité dans l'ordonnançabilité. Le taux de fiabilité dans l'ordonnançabilité, qui est détaillé dans la section 2.4.1, traduit la capacité de l'heuristique à trouver des solutions. Cette capacité sera comparée avec celle d'une méthode connue pour être plus performante mais qui est largement plus lente. L'heuristique proposée devra respecter, en outre, la précédence en garantissant les transferts de données entre les tâches. D'autre part elle cherchera à minimiser le makespan même si ce dernier objectif est considéré comme secondaire par rapport aux deux autres. Une série de tests sera nécessaire pour évaluer l'heuristique proposée ;

– ensuite nous allons chercher à améliorer, en termes de fiabilité dans l'ordonnançabilité, les résultats obtenus avec la première heuristique proposée. Cette amélioration passe par un compromis, qu'il va falloir trouver, entre le fait de gagner en ordonnançabilité et d'accepter de perdre en termes de rapidité. Pour y parvenir l'analyse d'ordonnançabilité est primordiale où, contrairement à la première version, des conditions plus affinées doivent être trouvées ;
– finalement et une fois que les deux premiers objectifs seront atteints, on cherche à minimiser encore le makespan et à réaliser un équilibrage de la mémoire. Ceci représente comme il est indiqué dans le section 1.7.2 un problème d'optimisation bi-critère qu'il convient de résoudre toujours à l'aide d'approches gloutonnes.

Le reste du chapitre est organisé comme suit : la section 2.1 expose le modèle flot de données classique d'abord pour passer ensuite au modèle flot de données multipériode qui est utilisé dans ce manuscrit. La section 2.2 s'étale sur l'heuristique d'ordonnancement proposée en détaillant les différentes étapes ainsi que l'étude d'ordonnançabilité réalisée. La section 2.3 présente l'algorithme exact qu'on a utilisé pour tester l'heuristique proposée. La section 2.4 rapporte les résultats des tests qu'on a effectué. Et pour finir on s'intéresse dans la section 2.5 à l'équilibrage de charge et de mémoire.

2.1 Modèles

2.1.1 Modèle d'algorithme

Nous allons d'abord présenter les modèles flot de donnée classique.

2.1.1.1 Modèle flot de donnée classique

Le concept des flux de données a été abordé par Karp et Miller [138] pour décrire leurs graphes de calcul. Plus tard Dennis [139] définit le modèle flot de données.

La notion de flot vient du fait que l'algorithme est représenté comme un réseau véhiculant un flux de données [62]. Chaque noeud du réseau (sommet) consomme le ou les flux des données entrants et produit le ou les flux de données sortants. Ainsi la description d'un algorithme avec le modèle flot de données nécessite de définir les traitements atomiques comme les sommets de

ce réseau et ensuite de préciser les liens raccordant ces sommets. La représentation graphique est alors naturelle, chaque sommet (raccordement) appelé tâche est un traitement atomique et chaque arc (lien) est soit une dépendance de donnée, soit une précédence. Une précédence entre deux tâches signifie que l'une ne doit être exécutée que lorsque la tâche qui la précède est exécutée. Alors qu'une dépendance, en plus d'être une précédence, signifie que les deux tâches échangent des données. Pour qu'une tâche en relation de précédence de donnée s'exécute et que le transfert de données s'effectue convenablement il est impératif que :

- toutes ses données d'entrée soient disponibles,
- toutes les données qu'elle est sensée produire soient produites et envoyées,
- une tâche qui s'exécute consomme des données sur chacune de ses dépendances de données entrantes et produit des données sur chacune de ses dépendances de données sortantes.

2.1.1.2 Modèle flot de données multipériode

À la différence du modèle précédent une information supplémentaire est ajoutée aux tâches (sommets), il s'agit de la période. La périodicité des tâches est la conséquence de la présence des capteurs et actionneurs qui imposent leurs périodes aux tâches. Cette nouvelle information permet aux tâches de s'exécuter à des périodes différentes, chacune d'elles possède sa propre période alors que l'exécution du système incluant toutes les tâches possède à son tour une période qu'on appelle l'"Hyper-période" et qui est le Plus Petit Commun Multiple (*PPCM*) de toutes les périodes des tâches du graphe [140]. Par conséquent chaque tâche s'exécute à plusieurs reprises relativement aux autres périodes des autres tâches, chaque exécution est appelée instance ou répétition et le nombre de ces instances est calculé à partir de la période de la tâche et de l'hyper-période. Il est important de noter que, étant donné qu'en général le nombre des capteurs et actionneurs est relativement petit dans les systèmes, la valeur de l'hyper-période est plus réduite.

Tout comme dans le précédent modèle les tâches (sommets), même en étant périodiques, sont reliées par des arcs qui symbolisent les dépendances de données. La plupart du temps les tâches dépendantes ont des périodes multiples [83], de plus cette restriction permet de s'assurer que les données sont transférées en totalité. La précédente restriction ne veut certainement pas dire que toutes les tâches doivent avoir des périodes multiples, seulement que les tâches qui ont des périodes non multiples ne peuvent échanger des données au risque d'en perdre (sauf si c'est voulu par l'utilisateur, ce qui n'est pas traité dans ce manuscrit).

Prenons l'exemple simple de deux tâches dépendantes t_a et t_b ($T(t_a)$ et $T(t_b)$ sont, respectivement, les périodes des tâches t_a et t_b). Avoir $T(t_b) = qT(t_b)$ tel que la tâche t_b consomme ce que la tâche t_a produit s'interprète par le fait que les besoins de la tâche t_b en données, pour être exécutée, dépassent ce que produit la tâche t_a en une seule exécution. Il lui faut, exactement, q fois ce que produit une exécution de la tâche t_a. Dès lors pour qu'une tâche s'exécute et que le transfert de données s'effectue convenablement il est impératif que :

- toutes ses données d'entrée soient disponibles. C'est-à-dire que toutes les données qui proviennent des instances (répétitions) de la ou les tâches productrices soient disponibles ;
- toutes les données qu'elle est sensée produire soient produites et envoyées,

– une tâche qui s'exécute consomme toutes les données issues de chacune de ses dépendances de données entrantes et se répète (s'exécute plusieurs fois) afin de produire assez de données pour chacune de ses dépendances de données sortantes.

Ce modèle n'est pas très différent du modèle SDF (Synchronous Data Flow) [141], étant donné que la seule différence se situe dans le fait que le nombre de données (jetons dans le modèle SDF) que consomme et produit chaque tâche n'est plus déterminé au préalable par l'utilisateur comme dans le modèle SDF mais par les périodes des tâches.

Pour conclure, dans cette thèse l'algorithme est modélisé par un graphe flot de données périodique, où les n sommets sont les n tâches de l'algorithme et les arcs sont les dépendances de données entre les tâches.

2.1.2 Modèle d'architecture

L'architecture est quand à elle est du type hétérogène et est décrite à l'aide d'un graphe non orienté où chaque noeud est un processeur ou un médium de communication et où chaque arc est une connexion entre un processeur et un médium [142]. Le nombre de processeurs est égal à m. Un médium de communication peut être une liaison point à point, une liaison multi-point (bus) avec ou sans broadcast, ou une mémoire partagée. Ce modèle de représentation des architectures distribuées est une extension des modèles classiquement utilisés pour spécifier les architectures parallèles ou distribuées que sont les PRAM (Parallel Random Access Machines) et les DRAM (Distributed Random Access Machines) [143]. Il permet de décrire des architectures utilisant les deux types de communication que sont les mémoires partagées (PRAM) et le passage de message (DRAM).

Des extensions de ce modèle permettant de raffiner la description de l'architecture, par exemple en décrivant les différentes unités de calcul (ALU, FPU, etc.) et interfaces de communication (DMA, convertisseur série/parallèle, etc.) d'une carte DSP, sont proposées dans [142] et [144]. Les heuristiques proposées doivent être compatibles avec les différentes architectures qui existent. Quelque soit le type de médium de communication utilisé (bus, mémoire, ...) le coût de communication doit être pris en compte. Par exemple si c'est une mémoire tous les coûts d'accès mémoire, de lecture et d'écriture doivent être pris en compte, et si c'est un bus c'est le coût du transfert qui est pris en compte.

2.1.3 Modèle temporel

Une tâche t_a est caractérisée par :

– une période $T(t_a)$ indépendante du processeur,
– l'échéance est égale à la période ($T(t_a) = D(t_a)$),
– une durée d'exécution $C(t_a)$ pour un processeur donné,
– une date de début d'exécution $S(t_a)$ pour un processeur donné,
– une date de fin d'exécution $C(t_a)$ égale à $S(t_a) + C(t_a)$ pour un processeur donné,
– les tâches sont non-concrètes.

Une tâche t_a est souvent représentée par sa période et sa durée d'exécution de la manière suivante : $(t_a : C(t_a), T(t_a))$.
Pour qu'une tâche $(t_a : C(t_a), T(t_a))$ puisse respecter sa période il est nécessaire que [58]:

$$C(t_a) \leq T(t_a) \tag{2.1}$$

Dans la suite on fait toujours cette hypothèse quand on définit une tâche.

On admet que les périodes et les durées d'exécution sont des multiples d'une unité de temps $U = 1$. Cela signifie que les valeurs des périodes et des durées d'exécution représentent par exemple quelques cycles d'horloge du processeur. Si on dit qu'une tâche t_a d'une durée d'exécution $C(t_a)$ commence son exécution à la date $S(t_a)$ alors t_a commence son exécution au début de l'unité de temps $S(t_a)$ et termine son exécution avant le début de l'unité de temps $S(t_a) + C(t_a)$. Ainsi l'intervalle de temps où la tâche t_a s'exécute est donné par $[S(t_a), S(t_a) + C(t_a)[$.

Il est important de noter que nous considérons des ordonnancements sans temps creux. En effet chaque tâche est ordonnancée immédiatement après la fin de l'exécution de la tâche qui a été ordonnancée juste avant, afin d'éviter de rallonger le makespan.

2.1.4 Périodicité stricte vs. périodicité classique

Ces deux contraintes ont été introduites dans la section 1.2.2.1. Comme on l'a précisé à plusieurs reprises, la contrainte de périodicité prise en compte dans ce manuscrit est du type strict. Ce choix de modèle n'est certainement pas anodin et encore moins sans conséquences. Il est vrai qu'à l'opposé de la périodicité classique la périodicité stricte est peu abordée dans la littérature (que ce soit dans le cas monoprocesseur ou le cas multiprocesseur). Néanmoins dans la réalité il existe des systèmes dont certaines tâches doivent être à périodicité stricte. Il s'agit des tâches qui interagissent avec l'environnement pour lesquelles on doit garantir qu'aucune donnée échangée avec l'environnement n'est perdue [145]. C'est le cas lorsque les données en entrée du système sont fournies strict-périodiquement par un convertisseur analogique/numérique (A/D pour analog/digital) et les données en sortie sont consommées strict-périodiquement par un convertisseur D/A. Dans [146] il est prouvé que si les capteurs et les actionneurs du système s'exécutent à des périodes strictes alors aucune donnée n'est perdue. De tels systèmes sont parfaitement prédictibles.

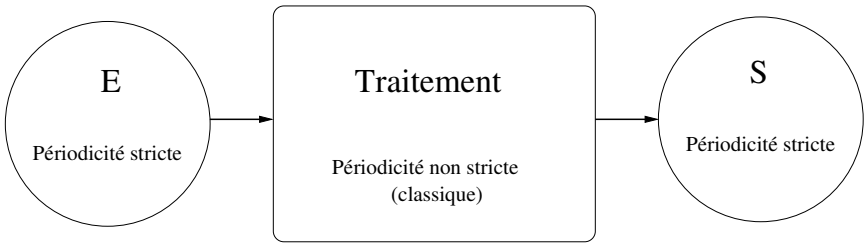

FIG. 2.1 – *Graphe d'algorithme d'un système sous la contrainte de périodicité stricte*

Le graphe de la figure 2.1 représente un système dont les tâches d'entrée et de sortie ont des contraintes de périodicité stricte et la partie traitement, même si elle hérite ses périodes de celles

des entrées, a une contrainte de périodicité non stricte. Afin de satisfaire la périodicité stricte des sorties, les périodes des tâches du traitement doivent s'exécuter à des périodes strictes comme le démontre le théorème suivant.

THÉORÈME 2.1
Pour satisfaire le bon fonctionnement des systèmes correspondant à la figure 2.1 où les périodes des entrées et des sorties sont strictes il est nécessaire que les périodes du traitement reliant ces deux parties soient, elles aussi, strictes.

Preuve
Pour prouver le théorème 2.1 on suppose que, dans un système, les périodes des entrées et des sorties sont strictes alors que les périodes du traitement sont non strictes (classiques). Sans perte de généralité on suppose que toutes les tâches ont la même période T car cette preuve demeure valable dans le cas où il existe plusieurs périodes.

Soient t_e et t_s deux tâches appartenant, respectivement, aux entrées et aux sorties. On en déduit que :

$\forall q \in \mathbb{N}, S(t_{e_q}) - S(t_{e_{(q-1)}}) = T,$

et

$\forall q \in \mathbb{N}, S(t_{s_q}) - S(t_{s_{q-1}}) = T.$

Soient t_a et t_b deux tâches du traitement telles que t_a est la première tâche du traitement et t_b est la dernière tâche du traitement. On en déduit que :

$\forall q \in \mathbb{N}, C(t_a) \leq S(t_{a_q}) - S(t_{a_{q-1}}) \leq 2T - C(t_a),$

et

$\forall q \in \mathbb{N}, C(t_b) \leq S(t_{b_q}) - S(t_{b_{q-1}}) \leq 2T - C(t_a).$

Etant donné que l'ordonnancement est sans temps creux on peut écrire :

1. $S(t_{a_{q-1}}) = S(t_{e_{q-1}}) + C(t_{e_{q-1}})$
2. $S(t_{a_q}) = S(t_{e_q}) + C(t_{e_q})$

ainsi que :

3. $S(t_{s_{i-1}}) = S(t_{b_{i-1}}) + C(t_{b_{i-1}})$
4. $S(t_{s_i}) = S(t_{b_i}) + C(t_{b_i})$

quand on soustrait (1) à (2) on a : $S(t_{a_i}) - S(t_{a_{i-1}}) = S(t_{e_i}) + C(t_{e_i}) - (S(t_{e_{i-1}}) + C(t_{e_{i-1}}))$
et donc :
$S(t_{a_i}) - S(t_{a_{i-1}}) = S(t_{e_i}) - S(t_{e_{i-1}}) = T$
de la même façon, quand on soustrait (3) à (4) on a :
$S(t_{s_i}) - S(t_{s_{i-1}}) = S(t_{b_i}) + C(t_{b_i}) - (S(t_{b_{i-1}}) + C(t_{b_{i-1}}))$
et donc :
$T = S(t_{s_i}) - S(t_{s_{i-1}}) = S(t_{b_i}) - S(t_{b_{i-1}})$

Ces équations contredisent les suppositions faites au départ au sujet de la périocité non stricte des tâches t_a et t_b $\square$

Comme nous traitons des systèmes temps réel critiques où au moins les tâches d'entrées ont des périodicités strictes, nous avons choisi de considérer que toutes les tâches à périodicité non stricte étaient à périodicité stricte. Cela permet de garantir que les tâches à périodicité non strictes respecterons leur contraintes. Alors que si on avait fait l'inverse, c'est-à-dire si on avait considéré les tâches à périodicité stricte comme non stricte, il n'aurait pas été possible de garantir que leur périodicité ait été respectée.

Comme il existait au début de nos travaux une littérature abondante sur les systèmes dont toutes les tâches ont des périodicités non strictes, alors qu'il y avait peu de travaux sur les systèmes dont certaines tâches ont des périodicités strictes, nous avons choisi de traiter, dans cette thèse, ce problème en profondeur afin de constituer une base pour résoudre le problème plus général combinant les deux types de périodicité.

Bien sûr le principal inconvénient de ce modèle avec toutes les tâches avec périodicité stricte est qu'en terme d'ordonnançabilité les possibilités sont réduites comparées à celles des modèles avec périodicité non stricte. C'est-à-dire qu'il existe des systèmes ordonnançables si les périodes de toutes les tâches sont non strictes qui deviennent non ordonnançables dès que les périodes des tâches sont considérées comme strictes, tandis que la réciproque est fausse. L'exemple le plus simple est l'ordonnancement de deux tâches avec des périodes premières entre elles, en périodicité stricte ces deux tâches ne sont ordonnançables que sur deux processeurs différents alors qu'en périodicité classique ces deux mêmes tâches sont ordonnançables sur le même processeur (voir l'exemple 2.1 page 60).

2.2 Heuristique d'ordonnancement

Afin de satisfaire toutes les contraintes imposées, à savoir la prise en compte des tâches périodiques et dépendantes, l'heuristique opère en trois parties. Elle est donc constituée de trois algorithmes dont deux sont indépendants et un troisième qui exploite les résultats des deux autres algorithmes pour produire l'ordonnancement recherché. Le premier qu'on appelle "Algorithme d'assignation" reprend les résultats obtenus pas l'analyse d'ordonnançabilité qui a été faite et concrétise l'approche par partitionnement qu'on a choisi pour résoudre notre problème. Le deuxième qu'on appelle "Algorithme de déroulement" transforme le graphe initial décrivant les tâches périodiques et dépendantes en un autre graphe où les tâches sont répétées suivant leurs périodes en ajoutant des dépendances afin d'établir les transferts de données correspondants, ce qui permet de faire abstraction de la périodicité des tâches. Le troisième, appelé "Algorithme d'ordonnancement" exploite les résultats des deux premiers algorithmes pour produire un ordonnancement des tâches du graphe d'algorithme sur les processeurs du graphe d'architecture en respectant les dépendances et les périodes strictes des tâches tout en cherchant à minimiser le makespan.

Nous allons, à présent, développer chaque partie en détail.

2.2.1 Assignation

Entre les deux approches connues pour la résolution des problèmes d'ordonnancement multiprocesseur, exposées dans l'état de l'art (voir la section 1.4.2), nous avons opté pour la méthode

du partitionnement. Par conséquent la première étape consiste à partitionner les tâches sur les processeurs de l'architecture afin de les ordonnancer par la suite. Nous appelons cette première étape "l'assignation". Ce terme, dans la littérature, peut avoir plusieurs sens, c'est pourquoi nous allons donner la signification employée dans ce manuscrit.

Assigner une tâche signifie lui associer un processeur de façon à ce que cette tâche, ainsi que toutes les tâches qui sont déjà assignées à ce même processeur soient ordonnançables. Les tâches seront ordonnancées dans la dernière partie de l'heuristique d'ordonnancement sur les processeurs où elles ont été assignées. Si une tâche est ordonnançable sur plusieurs processeurs alors elle est assignée à chacun de ces processeurs, c'est dans la dernière partie de l'heuristique que cette tâche, assignée à plusieurs processeurs, sera ordonnancée définitivement sur un processeur suivant des critères qu'on évoquera dans la suite.

Cette étape sert à déterminer si le système est ordonnançable avant de commencer l'ordonnacement effectif des tâches (celui-ci se fait dans la troisième partie). La différence entre l'assignation et l'ordonnancement est que la première s'intéresse à l'ordonnançabilité en essayant de trouver une assignation qui respecte toutes les contraintes, alors que la deuxième attribue des dates de départ à chaque tâche sur le processeur où elle a été assignée, tout en essayant de minimiser le makespan en opérant sur les tâches qui ont été éventuellement assignées à plusieurs processeurs. L'assignation exploite l'étude d'ordonnançabilité que l'on détaille plus loin.

2.2.1.1 Étude d'ordonnançabilité

Comme nous l'avons déjà évoqué dans l'état de l'art et étant donné la complexité du problème, il n'existe aucun résultat qui permet de dire en un temps polynomial si un ensemble de tâches périodiques et dépendantes est ordonnançable sur un ensemble de processeurs. L'avantage de l'approche par partitionnement est que le problème d'ordonnancement multiprocesseur se transforme en un ensemble de problèmes monoprocesseur. Au lieu de chercher une condition multiprocesseur on s'intéresse plutôt à une condition monoprocesseur qui permet de vérifier qu'un ensemble de tâches est ordonnançable sur un seul processeur, néanmoins ce nouveau problème n'est pas polynomial pour autant dans le cas non-préemptive [33]. C'est pourquoi la condition monoprocesseur qui nous intéresse, en premier lieu, considère l'assignation d'une seule tâche à la fois. Ensuite, nous examinons la possibilité éventuelle de l'améliorer. Cette condition monoprocesseur sera testée à chaque assignation d'une tâche pour permettre de lui trouver le processeur qui convient. "Convenir" veut dire que cette tâche, ainsi que les tâches qui ont déjà été assignées à ce processeur, sont ordonnançables sur ce processeur.

Pour commencer nous énonçons dans le théorème 2.2 la condition nécessaire et suffisante d'ordonnançabilité dans le cas basique de deux tâches. Ensuite nous prouvons avec les théorèmes 2.3 et 2.4 que la condition du théorème 2.2 n'est pas généralisable pour plus de deux tâches. Dans le but d'avoir une condition plus générale une série de résultats intermédiaires sont présentés. Tout d'abord le théorème 2.5 donne une condition nécessaire et suffisante d'ordonnançabilité d'un ensemble de tâches dont les périodes, deux à deux, ont le même PGCD (plus grand commun diviseur). Plus tard l'étude s'oriente vers un problème de calcul, c'est-à-dire qu'une fois qu'une première tâche est ordonnancée, on cherche une formule qui permet de calculer le nombre de tâches, identiques à une deuxième tâche, pourrait être ordonnancées (théorème 2.6). Deux tâches

sont identiques lorsqu'elles ont la même période et le même WCET bien qu'elles ne réalisent pas la même fonction. Ce résultat sera, ensuite, étendu au cas d'un ensemble de tâches dont les périodes, deux à deux, ont le même PGCD (théorème 2.7). Pour finir le théorème 2.8 et le corollaire 1 proposent un résultat qui permet d'assigner une tâche à un processeur quelle que soient les tâches qui ont déjà été assignées à celui-ci. Le théorème 2.9 prouve que la contrainte de précédence est respectée dans tous les précédents théorèmes.

Le théorème suivant donne une condition nécessaire et suffisante d'ordonnançabilité de deux tâches.

THÉORÈME 2.2
Deux tâches $(t_a : C(t_a),T(t_a))$ *et* $(t_b : C(t_b),T(t_b))$ *sont ordonnançables si et seulement si*

$$C(t_a)+C(t_b) \leq PGCD(T(t_a),T(t_b)) \tag{2.2}$$

Preuve
Soit $g = PGCD(T(t_a),T(t_b))$. Nous commençons par prouver que l'équation (2.2) est une condition suffisante. On suppose que t_a et t_b sont ordonnançables et que $S(t_a) = 0$ et $S(t_b) = C(t_a)$. Donc chaque instance de la tâche t_a est exécutée dans un intervalle appartenant à l'ensemble d'intervalles I_1, avec I_1=$\{\forall k \in \mathbb{N},[kT(t_a),kT(t_a)+C(t_a)[\}$ et chaque instance de la tâche t_b est exécutée dans un intervalle appartenant à l'ensemble d'intervalles I_2, avec I_2=$\{\forall k \in \mathbb{N},[kT(t_b)+C(t_a),kT(t_b)+C(t_a)+C(t_b)[\}$.

Nous allons introduire g dans les intervalles I_1 et I_2. I_1 peut être réécrit en I_1=$\{\forall k \in \mathbb{N},[gk\frac{T(t_a)}{g},gk\frac{T(t_a)}{g}+C(t_a)[\}$, et si n est un entier tel que $n = k\frac{T(t_b)}{g}$ alors on obtient I_1=$\{\forall k \in \mathbb{N},[ng,ng+C(t_a)[\}$. De la même manière on trouve que, si n est un entier tel que $m = k\frac{T(t_b)}{g}$ alors I_2=$\{\forall k \in \mathbb{N},[mg+C(t_a),mg+C(t_a)+C(t_b)[\}$. La supposition faite au départ (t_a et t_b sont ordonnançables) implique qu'aucun des intervalles appartenant aux ensembles I_1 et I_2 ne se chevauchent. En observant les débuts des intervalles appartenant à I_1 on constate qu'ils commencent tous à des dates qui sont des multiples de g. D'un autre côté, en observant les fins des intervalles appartenant à I_2, on constate qu'ils finissent tous à des dates qui sont égales à un multiple de $(g+C(t_a)+C(t_b))$. Dire que les intervalles appartenant aux ensembles I_1 et I_2 ne se chevauchent pas signifie que lorsque $(n = m+1)$ on a : $mg+C(t_a)+C(t_b) \leq ng$, ce qui est équivalent à $C(t_a)+C(t_b) \leq g$. Ceci prouve que la la condition est suffisante(2.2).

Pour prouver la nécessité de l'équation (2.2) on montre que si $C(t_a)+C(t_b) < g$ alors les tâches t_a et t_b sont ordonnançables. Ceci est équivalent à montrer que si les tâches t_a et t_b ne sont pas ordonnançables alors $C(t_a)+C(t_b) \leq g$. Sans perte de généralité, on suppose que $S(t_a) = 0$. Les tâches t_a et t_b ne sont pas ordonnançables signifie qu'il existe deux entiers x et y pour lesquels

$$[xT(t_a),xT(t_a)+C(t_a)[\cap$$
$$[S(t_b)+yT(t_b),S(t_b)+yT(t_b)+C(t_b)[\neq \emptyset$$

ce qui est équivalent à

$$[xT(t_a)-yT(t_b),xT(t_a)-yT(t_b)+C(t_a)[\cap$$

$$[S(t_b), S(t_b)+C(t_b)[\neq \emptyset$$

Selon le théorème de Bezout [147], il existe deux entiers p et q pour lesquels $pT(t_a)+qT(t_b) = g$. En prenant $x = zp$ et $y = -zq$, $z \in \mathbb{N}$ on a :

$$[zg, zg+C(t_a)[\cap [S(t_b), S(t_b)+C(t_b)[\neq \emptyset$$

Ceci est vrai si la longueur des intervalles vides entre les intervalles $[zg, zg+C(t_a)[, z \in \mathbb{N}$ (qui est égale à $g - C(t_a)$) est inférieure à la longueur des intervalles $[S(t_b), S(t_b)+C(t_b)[$ (qui est égale à $C(t_b)$). C'est-à-dire que : $g - C(t_a) < C(t_b)$ ce qui est équivalent à $g < C(t_a)+C(t_b)$. Ceci conclut la preuve du théorème 2.2 □

REMARQUE 2.1
La première conclusion qu'on peut tirer du théorème précédent est que si les périodes des deux tâches t_a et t_b sont premières entre elles ($PGCD(T(t_a), T(t_b)) = 1$) alors ces deux tâches ne sont pas ordonnançables (voir le prochain exemple).

EXEMPLE 2.1
Pour illustrer la remarque 2.1 on prend l'exemple de deux tâches qui selon le théorème 2.2 ne sont pas ordonnançables. Soient $(t_a : 1,2)$ et $(t_b : 1,3)$ deux tâches qu'on tente d'ordonnancer sur le même processeur.

Sur la figure 2.2 on remarque que l'intervalle de temps où la troisième instance de la tâche t_a s'exécute est le même que celui de la deuxième instance de la tâche t_b. Comme un processeur n'est pas capable d'exécuter deux tâches en même temps on est obligé de décaler l'une des deux instances. Sauf que dans ce cas la période stricte de la tâche qu'on décale n'est plus respectée, dès lors on peut dire que ce système (deux tâches et un processeur) n'est pas ordonnançable.

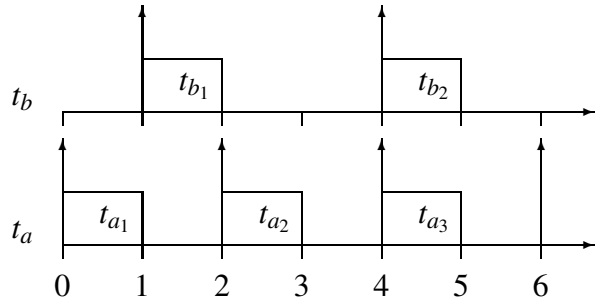

FIG. 2.2 – *Exemple de deux tâches de périodes premières entre elles*

La condition du théorème 2.2 n'est pas pratique puisque, comme cela est démontré par le théorème 2.3, elle ne peut pas être généralisée aux cas de plus de deux tâches. De plus, comme le démontre le théorème 2.4, le fait de vérifier l'ordonnançabilité des tâches deux à deux ne garantit pas l'ordonnançabilité de l'ensemble des tâches.

Le théorème suivant prouve que la généralisation de la condition du théorème 2.2 donne une condition seulement suffisante.

THÉORÈME 2.3
Si

$$\sum_{i=1}^{n} C(t_i) \leq PGCD(\forall i, T(t_i)) \tag{2.3}$$

alors l'ensemble des tâches $\{i \in \mathbb{N}^*, i \leq n, (t_i : C(t_i), T(t_i))\}$ *est ordonnançable.*

Preuve
Soit $g = PGCD(\forall i, T(t_i))$. Pour prouver que l'équation 2.3 est une condition suffisante on procède de la même manière que pour le théorème 2.2. Supposons que les tâches de l'ensemble $\{i \in \mathbb{N}^*, i \leq n, t_i\}$ sont ordonnançables et que $S(t_1) = 0$, et $S(t_i) = \sum_{j=1}^{i-1} C(t_j)$. Donc chaque instance de la tâche t_i (avec $i \in \mathbb{N}^*, i \leq n$) est exécutée dans un intervalle appartenant à l'ensemble d'intervalles I_i=$\{\forall k \in \mathbb{N}, [kT(t_i) + S(t_i), kT(t_i) + S(t_i) + C(t_i)[\}$. I_i peut être réécrit en I_i=$\{\forall k \in \mathbb{N}, [gk\frac{T(t_i)}{g} + S(t_i), k\frac{T(t_i)}{g} + S(t_i) + C(t_i)[\}$. Si n est un entier tel que $n = k\frac{T(t_i)}{g}$ alors on obtient I_i=$\{\forall k \in \mathbb{N}, [ng + S(t_i), ng + S(t_i) + C(t_i)[\}$. La supposition faite au départ (les tâches t_i sont ordonnançables) implique qu'aucun des intervalles appartenant aux ensembles I_i (avec $i \in \mathbb{N}^*, i \leq n$) ne se chevauchent. En observant les débuts des intervalles appartenant à I_o on constate qu'ils commencent tous à des dates qui sont des multiples de g. D'un autre côté, en observant les fins des intervalles appartenant à $I_i, (1 \leq i \leq n)$, on constate qu'ils finissent tous à des dates qui sont égales à un multiple de g plus $\sum_{i=1}^{n} C(t_i)$. Dire que les intervalles appartenant aux ensembles I_1 et I_2 ne se chevauchent pas signifie que pour $S(t_i)$ maximal, $ng + S(t_i) + C(t_i) \leq (n+1)g + S(t_1)$, ce qui est équivalent à $S(t_i) + C(t_i) \leq g$. Comme $\max(S(t_i), i \in \mathbb{N}^*)$=$S(t_n)$ on en déduit que $\sum_{i=1}^{n} C(t_i) \leq g$. Ceci prouve que la condition (2.3) est une condition suffisante□

EXEMPLE 2.2
Afin de vérifier que la condition (2.3) n'est pas une condition nécessaire il suffit de prendre comme contre exemple le système suivant : un processeur et trois tâches t_a, t_b *et* t_c *de périodes 2, 4 et 4 avec la valeur 1 comme durée d'exécution pour les trois tâches. Ces trois tâches sont ordonnançables même si la somme de leurs durées d'exécution, qui est égale à 3, est supérieure au PGCD de toutes les tâches (*$PGCD(4,4,2) = 2$*).*

Sur la figure 2.3 on remarque que ce système est ordonnançable vu que sur un intervalle de temps égal à deux fois l'hyper-période (dans ce cas elle est égale à 4) on arrive à ordonnancer les trois tâches ainsi que leurs instances.

THÉORÈME 2.4
Le fait que n tâches satisfassent deux à deux la condition du théorème 2.2 n'implique pas l'ordonnançabilité des n tâches

Preuve
Pour prouver cette non implication il suffit de proposer un contre exemple 2.3.

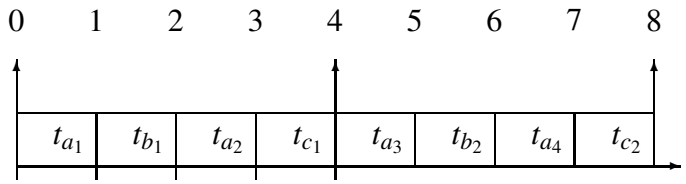

FIG. 2.3 – *Exemple de trois tâches ordonnançables*

EXEMPLE 2.3
Prenons l'exemple de trois tâches de périodes : 2, 4 et 6 avec 1 comme durée d'exécution pour les trois tâches. Prises deux à deux ces tâches vérifient la condition du théorème 2.2. Seulement, comme le montre la figure 2.4, ces trois tâches ne sont pas ordonnançables simultanément (t_{c_2} et t_{b_3} sont ordonnancées sur le même intervalle de temps)

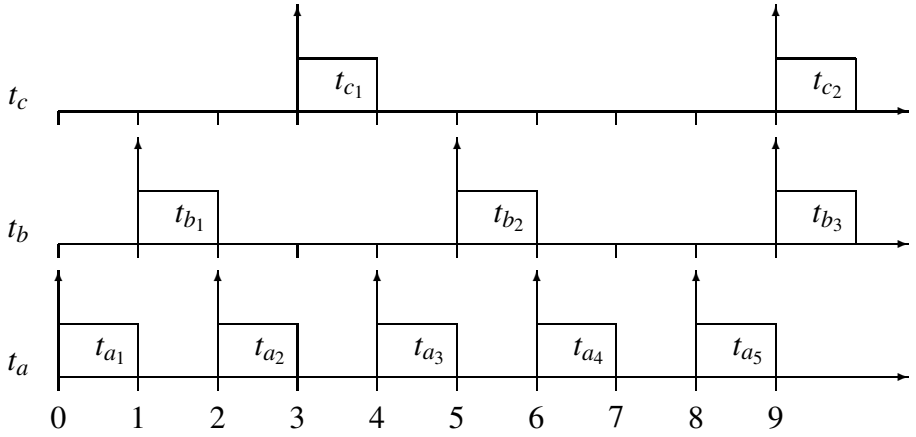

FIG. 2.4 – *Exemple de trois tâches non ordonnançables sur le même processeur*

La première étape consiste à chercher une condition qui prend en compte le plus de tâches possibles. Le théorème suivant introduit une condition nécessaire et suffisante qui permet de considérer un ensemble de tâches partageant la même propriété.

PROPRIÉTÉ 2.1
On dit que les n tâches de l'ensemble $\{\forall i \in \mathbb{N}^*, i \leq n, (t_i : C(t_i), T(t_i))\}$, tel que $g = PGCD(\forall i \in \mathbb{N}^*, i \leq n, t_i)$, satisfont la propriété 2.1 si : quelle que soit la paire de tâches $((t_a : C(t_a), T(t_a)), (t_b : C(t_b), T(t_b)))$ le $PGCD(T(t_a), T(t_b))$ est égal à g.

THÉORÈME 2.5
Soit $\{\forall i \in \mathbb{N}^*, i \leq n, (t_i : C(t_i), T(t_i))\}$ un ensemble de n tâches qui satisfont la propriété 2.1. Les

tâches de cet ensemble sont ordonnançables si et seulement si :

$$\sum_{i=1}^{n} C(t_i) \leq g \tag{2.4}$$

Preuve

La condition 2.4 est une condition suffisante a été prouvée par le théorème 2.3.

Nous prouvons que la condition 2.4 est une condition nésessaire en montrant que si $\sum_{i=1}^{n} C(t_i) \leq g$ alors les tâches de l'ensemble $\{\forall i \in \mathbb{N}^* \text{ et } i \leq n, t_i\}$ sont ordonnançables. Ce qui est équivalent à montrer que si les tâches de l'ensemble $\{\forall i \in \mathbb{N}^* \text{ et } i \leq n, t_i\}$ ne sont pas ordonnançables alors $g < \sum_{i=1}^{n} C(t_i)$.

On utilisera la preuve par induction.

Le cas de base : pour un ensemble de deux tâches $\{t_1, t_2\}$ la condition 2.4 a été prouvée condition nécessaire dans le théorème 2.2.

L'étape inductive : maintenant on montre que si la condition est valable pour l'ensemble $\{\forall i \in \mathbb{N} \text{ et } i \leq (n-1), t_i\}$ alors elle reste valable pour l'ensemble $\{\forall i \in \mathbb{N} \text{ et } i \leq n, t_i\}$.

Les tâches de l'ensemble $\{\forall i \in \mathbb{N} \text{ et } i \leq n, t_i\}$ ne sont pas ordonnançables signifie que des entiers $x_1, x_2, \ldots, x_n$ existent pour lesquels :

$$\begin{gathered}([S(t_1)+x_1T(t_1), S(t_1)+x_1T(t_1)+C(t_1)[\cup\ldots\cup \\ [S(t_{n-1})+x_{n-1}T(t_{n-1}), \\ S(t_{n-1})+x_{n-1}T(t_{n-1})+C(t_{n-1})[)\cap \\ [S(t_n)+x_nT(t_n), S(t_n)+x_nT(t_n)+C(t_n)[\neq \emptyset\end{gathered}$$

ceci est équivalent à

$$\begin{gathered}([S(t_1)+x_1T(t_1), S(t_1)+x_1T(t_1)+C(t_1)[\cap \\ [S(t_n)+x_nT(t_n), S(t_n)+x_nT(t_n)+C(t_n)[)\cup\ldots\cup \\ ([S(t_{n-1})+x_{n-1}T(t_{n-1}), \\ S(t_{n-1})+x_{n-1}T(t_{n-1})+C(t_{n-1})[\cap \\ [S(t_n)+x_nT(t_n), S(t_n)+x_nT(t_n)+C(t_n)[)\neq \emptyset\end{gathered}$$

on peut le réécrire aussi de la façon suivante :

$$\begin{gathered}([S(t_1)+x_1T(t_1)-x_nT(t_n), \\ S(t_1)+x_1T(t_1)-x_nT(t_n)+C(t_1)[\\ \cap[S(t_n), S(t_n)+C(t_n)[)\cup\ldots\cup \\ ([S(t_{n-1})+x_{n-1}T(t_{n-1})-x_nT(t_n), \\ S(t_{n-1})+x_{n-1}T(t_{n-1})-x_nT(t_n)+C(t_{n-1})[\cap\end{gathered}$$

$$[S(t_n),S(t_n)+C(t_n)[\neq\emptyset$$

Par ailleurs, suivant le théorème de Bezout, il existe des paires d'entiers (p_i,q_i) pour lesquelles $p_iT(t_i)+q_iT(t_n)=g$ $(i=1,...,n-1)$, en posant $x_i=l_ip_i$ et $x_n=-l_iq_i$ $(l_i,...,l_n\in\mathbb{N})$, on obtient :

$$([S(t_1)+l_1g,S(t_1)+l_1g+C(t_1)[\cap$$
$$[S(t_n),S(t_n)+C(t_n)[)\cup...\cup$$
$$([S(t_{n-1})+l_{n-1}g,S(t_{n-1})+l_{n-1}g+C(t_{n-1})[\cap$$
$$[S(t_n),S(t_n)+C(t_n)[\neq\emptyset$$

Ce qui peut être réécrit en :

$$([S(t_1)+l_1g,S(t_1)+l_1g+C(t_1)[\cup...\cup$$
$$[S(t_{n-1})+l_{n-1}g,S(t_{n-1})+l_{n-1}g+C(t_{n-1})[)$$

$$\cap[S(t_n),S(t_n)+C(t_n)[\neq\emptyset$$

Cette dernière est vraie si la somme des longueurs des intervalles vides entre les intervalles $([S(t_1)+l_1g,S(t_1)+l_1g+C(t_1)[\cup...\cup[S(t_{n-1})+l_{n-1}g,S(t_{n-1})+l_{n-1}g+C(t_{n-1})[)$ (qui est égale à $(g-\sum_{i=1}^{n-1}C(t_i))$), est inférieure à la longueur des intervalles $[S(t_n),S(t_n)+C(t_n)[$ (qui est égale à $C(t_n)$). C'est-à-dire que : $(g-\sum_{i=1}^{n-1}C(t_i))<C(t_n)$, ce qui est équivalent à $g<\sum_{i=1}^{n}C(t_i)$. Ceci complète la preuve par induction et la preuve du théorème$\boxdot$

Le condition du théorème 2.5 permet, comme c'était l'objectif, de prendre en compte plus de deux tâches. C'est-à-dire que si toutes les tâches dont on dipose satisfont la propriété 2.1 alors il suffit d'appliquer la condition du théorème 2.5 pour savoir si elles sont ordonnançables ou pas. Par contre cette condition ne sert plus à déterminer l'ordonnançabilité de toutes les tâches si celles-ci ne satisfont pas la propriété 2.1. Dire qu'on est incapable de déterminer l'ordonnançabilité de tout un ensemble de tâches signifie qu'on ne peut pas assigner, simultanément, toutes les tâches de cet ensemble à un processeur.

À défaut de viser une condition qui regroupe toutes les tâches nous avons tenté de trouver un autre moyen pour vérifier le respect des contraintes. L'idée est de définir une condition qui permet d'assigner une tâche à un processeur où un ensemble de tâches a déjà été assigné. On appelle la tâche qui est sur le point d'être assignée la tâche candidate à l'assignation et dès qu'elle est assignée une autre tâche, parmi les tâches qui n'ont pas encore été assignées, devient la tâche candidate à l'assignation. Pour qu'elle soit assignée à un processeur la tâche candidate à l'assignation, ainsi que l'ensemble des tâches assignées auparavant à ce même processeur, doivent être ordonnançables sur un processeur. Par conséquent, comme les tâches sont assignées les unes après les autres, on est assuré que toutes les tâches considérées comme assignées au même processeur sont ordonnançables sur le même processeur.

Il est évident que pour la toute première tâche candidate à l'assignation, le nombre d'intervalles pouvant servir à ordonnancer cette tâche est infini puisque aucune tâche n'a été assignée avant. Cependant, au fur et à mesure que des tâches sont assignées, le nombre d'intervalles pouvant servir à ordonnancer une tâche candidate à l'assignation diminue jusqu'à ce qu'il soit égal à 0, ce qui signifie que cette tâche n'est pas ordonnançable, sur le même processeur, avec les autres tâches qui ont été assignées avant. L'exemple suivant illustre cette idée avec un cas simple.

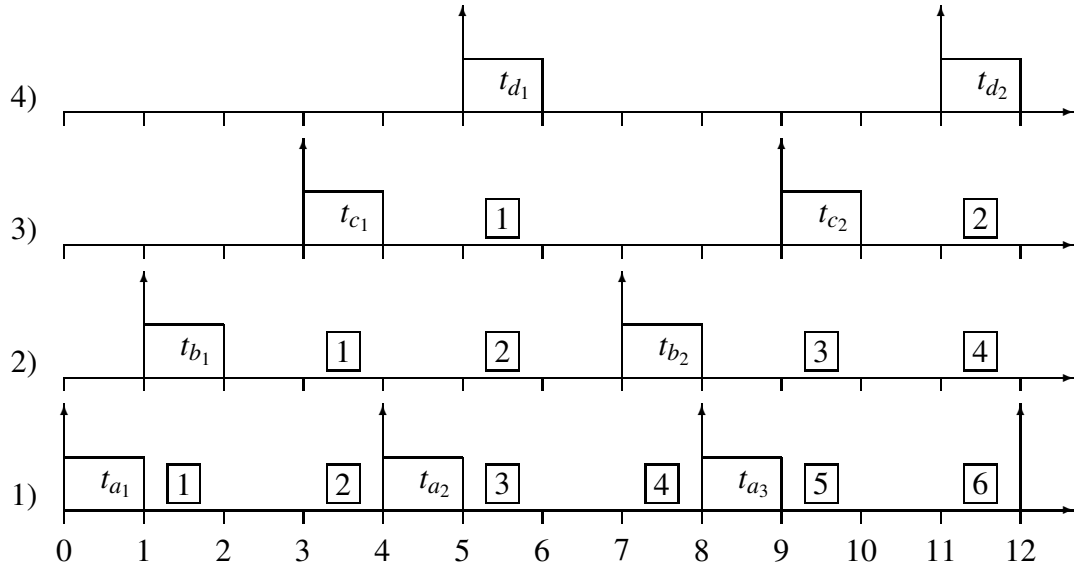

FIG. 2.5 – *(Exemple 2.4) Ordonnancement des tâches par étape*

EXEMPLE 2.4

La question est : une fois qu'une tâche t_a est ordonnancée et en prenant en compte tous les intervalles occupés par les instances de cette tâche, combien de tâches identiques à une tâche t_b peut-on ordonncer ? Si on considère que $(t_a : 1,4)$ alors l'ordonnancement de ses instances laisse 6 intervalles où une tâche $(t_b : 1,6)$ peut être ordonnancée. Les six intervalles où la tâche t_b peut être ordonnancée sont indiqués sur la figure 2.5-1). Ainsi, Si la tâche t_b est ordonnancée sur un des intervalles vides autres que les six indiqués alors l'ordonnançabilité n'est pas satisfaite puisque une des instances de t_b s'exécute sur le même intervalle qu'une des instances de t_a.

Une fois que la tâche t_b est ordonnancée, il reste 4 intervalles où une tâche $(t_c : 1,6)$ peut être ordonnancée (voir la figure 2.5-2). Après que la tâche t_c ait été ordonnancée il reste 2 intervalles où une tâche $(t_d : 1,6)$ peut être ordonnancée (voir la figure 2.5-3). Pour finir, après que la tâche t_d ait été ordonnancée on constate qu'il ne reste plus d'intervalle vide où une tâche $(t_e : 1,6)$ peut être ordonnancée (voir la figure 2.5-4). Notons que les tâches t_c et t_d sont identiques à la tâche t_b (la même durée d'exécution et la même période). Ce qui signifie qu'au total trois tâches identiques à t_b peuvent être ordonnancées en même temps.

Ceci dit l'objectif n'est certainement pas d'ordonnancer plusieurs fois la tâche t_b mais uniquement pour savoir si il existe un moyen de calculer le nombre maximal de tâches identiques à t_b qui peuvent être ordonnancées (3 dans cette exemple).

REMARQUE 2.2
D'après l'exemple précédent, une fois qu'une tâche t_a est ordonnancée alors on peut non seulement ordonnancer la tâche candidate t_b mais aussi d'autres tâches qui sont identiques à celle-ci. Le but du théorème qui suit est d'introduire une formule qui permet de calculer le nombre de tâches qui peuvent être ordonnancées (ce nombre comprend la tâche candidate t_b plus quelques tâches identiques à t_b). Dans la suite nous appelons ce nombre le nombre de fois que t_b peut être ordonnancée et il est noté par $\Psi(t_b)$.

THÉORÈME 2.6
Soient $(t_a : C(t_a),T(t_a))$ et $(t_b : C(t_b),T(t_b))$ deux tâches qui satisfont la condition 2.2 et soit $g = PGCD(T(t_a),T(t_b))$. Une fois que la tâche t_a est ordonnancée alors le nombre de fois que t_b peut être ordonnancée est donnée par :

$$\Psi(t_b) = \frac{T(t_b)}{g}\left\lfloor \frac{(g - C(t_a))}{C(t_b)} \right\rfloor$$

Preuve
$\left\lfloor \frac{(g-C(t_a))}{C(t_b)} \right\rfloor$ est, dans un intervalle de longueur g le nombre de fois que t_b peut être ordonnancée sachant qu'une tâche t_a a déjà été ordonnancée.

Une fois ce nombre trouvé, il est multiplié par le nombre d'intervalles de longueur g dans un intervalle de longueur $T(t_b)$ qui est égal à $\frac{T(t_b)}{g}$. La raison est que, quand une tâche t_b est ordonnancée, elle va certainement se répéter après $T(t_b)$ unités de temps (périodicité stricte). Ce qui signifie que l'intervalle, où une instance de t_b s'exécute, ne peut pas être utilisé pour ordonnancer une autre tâche t_b $\boxdot$

Le théorème précédent peut être généralisé pour toutes les tâches qui satisfont la propriété 2.1.

THÉORÈME 2.7
Soit $(t_{cdt} : C(t_{cdt}),T(t_{cdt}))$ la tâche candidate à l'assignation et soit $\{\forall i \in \mathbb{N}^*$ et $i \leq n,(t_i : C(t_i),T(t_i))\}$ un ensemble de n tâches tel que les n tâches ainsi que t_{cdt} satisfont la propriété 2.1. Le nombre de fois que la tâche t_{cdt} peut être ordonnancée est donné par :

$$\Psi(t_{cdt}) = \frac{T(t_{cdt})}{PGCD(g,T(t_{cdt}))}\left\lfloor \frac{PGCD(g,T(t_{cdt})) - \sum_{i=1}^{n} C(t_i)}{C(t_{cdt})} \right\rfloor$$

Preuve
Comme pour le théorème précédent $\left\lfloor \frac{PGCD(g,T(t_{cdt})) - \sum_{i=1}^{n} C(t_i)}{C(t_{cdt})} \right\rfloor$ représente le nombre de fois que t_{cdt} pourrait être ordonnancée dans un intervalle de longueur g. $\frac{T(t_{cdt})}{PGCD(g,T(t_{cdt}))}$ représente le nombre d'intervalles de longueur g dans un intervalle de longueur $T(t_{cdt})$ $\boxdot$

EXEMPLE 2.5
Soient $(t_a : 1,4)$, $(t_b : 1,12)$ et $(t_c : 1,16)$ trois tâches déjà ordonnancées. On cherche à assigner une tâche $(t_d : 1,20)$. En utilisant le résultat du théorème 2.7 on va calculer le nombre de fois que t_d peut être ordonnancée.

Avant de commencer le calcul on s'assure que les tâches t_a, t_b, t_c et t_d vérifient la condition du théorème 2.5 : $PGCD(T(t_a),T(t_b)) = PGCD(T(t_b),T(t_c)) = PGCD(T(t_c),T(t_a)) = PGCD(T(t_c),T(t_d)) = PGCD(T(t_b),T(t_d)) = PGCD(T(t_a),T(t_d)) = 4$, $C(t_a)+C(t_b)+C(t_c)+C(t_d) = 4 \leq 4$.

En appliquant le résultat du théorème 2.7 le nombre qu'on obtient est : $\Psi(t_d) = \frac{20}{4} * \lfloor \frac{4-3}{1} \rfloor = 5$.

Sur la figure 2.6 sont indiqués les cinq intervalles où la tâche t_d peut être ordonnancée (les cadres numérotés de un à cinq). On y indique aussi les intervalles qui, même en étant vides, ne permettent pas d'ordonnancer t_d et, à chaque fois, la tâche qui cause la non-ordonnançabilité. Cette non-ordonnançabilité signifie qu'une instance de la tâche t_d et une instance de la tâche à l'origine de la non-ordonnançabilité doivent être ordonnancées sur le même intervalle. Il faut noter que pour réaliser cet exemple il a fallu aller jusqu'à la date 240 *(ce qui représente l'hyper-période) alors que sur la figure 2.6, pour les raisons qui sont citées dans le prochain paragraphe, on ne va pas plus loin que la date* 24.

Vérifions, sur la figure 2.6 cette fois, le résultat qu'on a obtenu avec le précédent calcul. On remarque que si on divise l'axe des temps qui représente le processeur en intervalles de taille égale au $PGCD = 4$, *ensuite on divise chacun de ces intervalles en quatre intervalles de taille* 1 *alors c'est toujours le quatrième intervalle qui peut être utilisé pour ordonnancer la tâche t_d. En d'autres termes c'est l'ordre établi dans le premier intervalle de la taille du* $PGCD = 4$ *qui se répète de la même manière dans les intervalles de la taille du* $PGCD = 4$ *qui suivent. D'ailleurs même si il n'y a pas d'instance de la tâche t_c ordonnancée à la date* 6 *on ne peut pas ordonnancer la tâche t_d à cette date parce que la tâche t_c occupe le troisième intervalle de taille* 1 *dans le premier intervalle de la taille du* $PGCD = 4$ *et l'intervalle entre les dates* 6 *et* 7 *correspond, en effet, au troisième intervalle de taille* 1 *dans le deuxième intervalle de la taille du* $PGCD = 4$. *Quand on arrive au sixième intervalle de la taille du* $PGCD = 4$ *il se trouve qu'à l'intérieur de celui-ci c'est la première tâche t_d qui s'exécute une deuxième fois suivant sa période (la date de début d'exécution de celle-ci est* 3 *et comme sa période est égale à* 20, *elle s'exécute une deuxième fois à la date* 23*). On en déduit qu'on ne peut pas ordonnancer une sixième tâche t_d et que le nombre maximal de tâches identiques à t_d qu'on peut ordonnancer est équivalent à* 5.

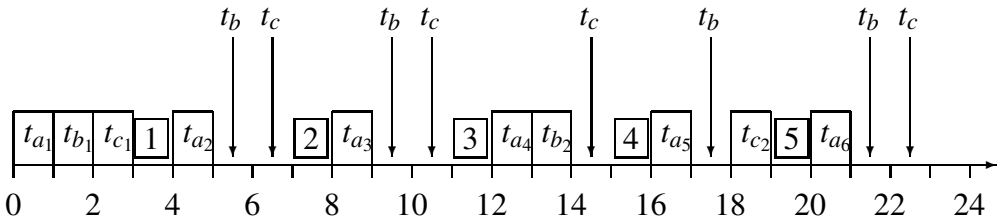

FIG. 2.6 – *(Exemple 2.5) Les différentes possibilités d'ordonnancement de la tâche t_d*

Notations

On note par Ω l'ensemble des tâches qui ont déjà été assignées à un processeur. Cet ensemble est divisé en plusieurs sous-ensembles $\{\Omega_1,\Omega_2,...,\Omega_v\}$ tel que les tâches de chaque sous-ensemble ainsi

que la tâche candidate t_{cdt} satisfont la propriété 2.1. Suite à cette division chaque sous-ensemble sera caractérisé par un PGCD et on considère que Ω_1 est l'ensemble avec le plus petit PGCD (qui est en même temps le PGCD de toutes les tâches de l'ensemble Ω). On note ce plus petit PGCD par BG (pour PGCD de base) et $C(\Omega_j) = \sum_{t_i \in \Omega_j} C(t_i)$. Finalement on note par Q le résultat de $(BG - C(\Omega_1))$.

À présent pour savoir si la tâche candidate t_{cdt} est ordonnançable ou pas sur un processeur où d'autres tâches ont déjà été assignées on applique l'algorithme suivant :

1. Construire les ensembles $(\forall j)$, Ω_j,
2. Calculer $\Psi_{\Omega_1}(t_{cdt})$. Cela représente le nombre de fois que t_{cdt} peut être ordonnancée en prenant en considération Ω_1 uniquement (ceci peut être effectué en utilisant le résultat du théorème 2.7),
3. Comme les autres ensembles $(j > 1)$, Ω_j n'ont pas été pris en compte, ôter au nombre obtenu à l'étape précédente, pour chaqu'un de ces ensembles le nombre $\Gamma_{\Omega_j}(t_{cdt})$. C'est-à-dire, le nombre de fois où : si t_{cdt} est ordonnancée alors une ou plusieurs tâches appartenant à l'ensemble Ω_j ne respecteront pas leurs périodes (l'équation utilisée pour ce calcul est donnée par le prochain théorème) ;
4. Suivant le résultat obtenu on décide d'assigner la tâche candidate ou bien de tenter l'assignation sur un autre processeur.

Maintenant on cherche à déterminer $\Gamma_{\Omega_j}(t_{cdt})$ (où $j \neq 1$) qui représente, parmi le nombre de fois que t_{cdt} peut être assignée, le nombre de fois que la tâche candidate ne peut être assignée à cause de sa non ordonnançabilité avec des tâches appartenant à l'ensemble Ω_j tel que$(j > 1)$.

THÉORÈME 2.8
Soit t_{cdt} la tâche candidate et soit Ω=$\{\Omega_1,\Omega_2,...,\Omega_v\}$ l'ensemble des ensembles de tâches déjà assignées. Si Ω_j tel que $(j > 1)$ est un des ensembles appartenant à Ω et si $PGCD(T(t_{cdt}),\forall t_i \in \Omega_j\ T(t_i)) = g$ alors le nombre $\Gamma_{\Omega_j}(t_{cdt})$ de fois que t_{cdt} ne peut pas être ordonnancée relativement à Ω_j est donné par :

$$\Gamma_{\Omega_j}(t_{cdt}) = \frac{T(t_{cdt})}{g}\left[\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor \left\lfloor \frac{C(\Omega_j)}{Q} \right\rfloor + \left(\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor - \left\lfloor \frac{C(\Omega_j) - C(\Omega_j) \bmod Q}{C(t_{cdt})} \right\rfloor \right)\right]$$

Preuve
$\frac{T(t_{cdt})}{g}$ est le nombre d'intervalles de longueur g dans un intervalle de longueur $T(t_{cdt})$. Ce nombre est multiplié par $\left[\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor \left\lfloor \frac{C(\Omega_j)}{Q} \right\rfloor + \left(\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor - \left\lfloor \frac{C(\Omega_j) - C(\Omega_j) \bmod Q}{C(t_{cdt})} \right\rfloor \right)\right]$ qui représente, dans un intervalle de longueur g, le nombre de fois que t_{cdt} ne peut pas être ordonnancée. La raison vient du fait que si la tâche t_{cdt} est ordonnancée sur l'un de ces intervalles une ou plusieurs tâches appartenant à l'ensemble Ω_j ne pourraient se répéter suivant leurs périodes□

Désormais il ne reste plus qu'à rassembler les différents résultats pour obtenir une formule qui permet de calculer le nombre de fois que la tâche candidate à l'assignation peut être ordonnancée et du coup en déduire si elle est assignable ou pas. Le corollaire suivant introduit ce calcul.

COROLLAIRE 1

Soit t_{cdt} la tâche candidate et soit $\Omega=\{\Omega_1,\Omega_2,...,\Omega_v\}$ l'ensemble des ensembles de tâches déjà assignées. Le nombre de fois que la tâche t_{cdt} peut être ordonnancée est donné par :

$$\Psi_{\Omega_1}(t_{cdt}) - \sum_{j=2}^{v} \Gamma_{\Omega_j}(t_{cdt}) \tag{2.5}$$

avec

$$\Psi_{\Omega_1}(t_{cdt}) = \frac{T(t_{cdt})}{BG} \left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor$$

Preuve

L'équation 2.5 peut être réécrite en :

$$\frac{T(t_{cdt})}{BG} \left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor - \sum_{j=2}^{v} \frac{T(t_{cdt})}{PGCD(T(t_{cdt}),\forall t_k \in \Omega_j\ T(t_k))} \left[\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor \left\lfloor \frac{C(\Omega_j)}{Q} \right\rfloor + \left(\left\lfloor \frac{Q}{C(t_{cdt})} \right\rfloor - \left\lfloor \frac{C(\Omega_j) - C(\Omega_j) \bmod Q}{C(t_{cdt})} \right\rfloor \right) \right]$$

Ce corollaire est le résultat direct des deux derniers théorèmes. En effet le théorème 2.7 nous permet de calculer le nombre de fois que la tâche t_{cdt} peut être ordonnancée en prenant en compte, uniquement, les tâches de l'ensemble Ω_1. Notons que les tâches de l'ensemble Ω_1 sont celles qui se répètent le plus parmi toutes les tâches déjà assignées. Une fois ce nombre obtenu, on y soustrait les tâches qui affectent la périodicité des tâches appartenant à chacun des ensembles Ω_j tel que $1 < j \leq v$ (ceci est calculable en utilisant le théorème 2.8). À partir de là si le résultat obtenu est supérieur à 0 alors on en déduit le nombre de tâches identiques à t_{cdt} pouvant être assignées sinon, si le résultat est égal à 0 alors on en déduit que la tâche t_{cdt} ne peut pas être assignée au processeur en question et qu'il va falloir tenter de l'assigner à un autre processeur $\boxdot$

REMARQUE 2.3

On aurait pu se contenter d'une condition qui, comme prévu, ne permet d'assigner qu'une seule tâche à la fois. Cependant, au cours de cette étude on s'est rendu compte qu'on est capable d'assigner plusieurs tâches, à la fois, à condition qu'elles soient identiques (la même durée d'exécution et la même période). D'un autre côté, dans un système temps réel, il est courant que des tâches soient identiques sans qu'elles aient la même fonction. Dès lors ce résultat nous permet d'accelerer l'algorithme d'assignation en assignant à chaque fois autant de tâches que le résultat de l'équation du corollaire 1.

Par ailleurs un autre intérêt de ce résultat est que si il reste des tâches identiques à la tâche candidate mais qui n'ont pas pu être assignées, en même temps qu'elle, parce que le nombre total des tâches identiques à la tâche candidate est supérieur au résultat de l'équation du corollaire 1 alors on peut, d'ores et déjà, en déduire que les tâches restantes sont non assignables sur ce processeur et qu'il faut tenter de les assigner sur les autres processeurs.

EXEMPLE 2.6

Pour illustrer le corollaire précédent nous proposons d'appliquer son résultat à une tâche candidate à l'assignation et un processeur où un ensemble ω de tâches, ordonnançables, a déjà été assigné.

Soient $(t_x : 2{,}30)$ *la tâche candidate à l'assignation et* $\omega = \{(t_a : 2{,}5),(t_b : 1{,}10),(t_c : 1{,}20),(t_d : 1{,}30),(t_e : 1{,}60)\}$ *l'ensemble des tâches déjà assignées. Dans ce cas*
BG = PGCD $(T(t_a),T(t_b),T(t_c),T(t_d),T(t_e),T(t_x))$=5.

- *les 3 sous-ensembles qu'on forme à partir de l'ensemble* ω *suivant la propriété 2.1 sont :*
  - $\Omega_1 = \{t_x, t_a\}$ *avec un PGCD égal à 5,*
  - $\Omega_2 = \{t_x, t_b, t_c\}$ *avec un PGCD égal à 10,*
  - $\Omega_3 = \{t_x, t_d, t_e\}$ *avec un PGCD égal à 30.*
- *selon le corollaire 1,* $\Psi_{\Omega_1} = \frac{30}{5}\left\lfloor \frac{5-2}{2} \right\rfloor = 6$,
- $\Gamma = \Gamma_{\Omega_2} + \Gamma_{\Omega_3}$,
  - $\Gamma_{\Omega_2} = (\frac{30}{10}(\left\lfloor \frac{5-2}{2} \right\rfloor \left\lfloor \frac{2}{5-2} \right\rfloor + (\left\lfloor \frac{5-2}{2} \right\rfloor - \left\lfloor \frac{2-2 \bmod (5-2)}{2} \right\rfloor))) = 3 \times 1 = 3$,
  - *de la même manière on trouve que* $\Gamma_{\Omega_3} = 1$, *donc,* $\Gamma = 3 + 1 = 4$,
- *ainsi* $(\Psi_{\Omega_1} - \Gamma) = 6 - 4 = 2$, *on en déduit que la tâche* t_x *ainsi qu'une tâche identique à elle sont assignables au processeur où les tâches de l'ensemble* Ω *ont déjà été assignées. De plus si il existe d'autres tâches identiques à* t_x *alors celles-ci ne sont pas assignables sur ce processeur.*

REMARQUE 2.4
Tout au long de cette étude la contrainte de précédence n'a jamais été évoquée alors qu'il en est question dans le modèle. Cela est justifié dans le prochain théorème.

Effectivement un système de tâches avec contraintes de précédence uniquement (sans contraintes de périodicité) est toujours ordonnançable. Le théorème suivant démontre que, pour un ensemble de tâches périodiques ordonnançables (une assignation de toutes les tâches sur un processeur est possible), il existe au moins un ordonnancement - de ces tâches - qui respecte les précédences entre elles.

THÉORÈME 2.9
Soit Ω *un ensemble de tâches qui ont été assignées au même processeur. Quelles que soient les contraintes de précédences entre ces tâches il existe au moins un ordonnancement qui les satisfait toutes.*

Preuve
Une fois que les n tâches de l'ensemble Ω ont été prouvées ordonnançables, on peut les ordonnancer de $n!$ manières correspondant à toutes les permutations possibles des éléments de l'ensemble Ω. Parmi ces $n!$ ordonnancements il en existe au moins un qui satisfait les contraintes de précédence car les conditions d'assignation proposées tout au long de cette étude sont indépendantes de l'ordre entre les tâches (on rappelle que les tâches sont non préemptives) $\square$

À ce stade de l'étude on est capable de produire un algorithme qui permet d'assigner les tâches aux processeurs tout en respectant les différentes contraintes. Cependant la condition que l'on utilise est une condition monoprocesseur qui s'applique à chaque fois que l'on cherche à assigner

une tâche (tout en sachant que le test peut être fait plusieurs fois jusqu'à ce que le bon processeur soit trouvé) et cela se traduit par un temps d'exécution de l'algorithme d'assignation plus long. On a, donc, décidé de développer deux algorithmes d'assignation différents et de les comparer par la suite. Ces deux algorithmes sont introduits dans les deux prochaines sections.

2.2.1.2 Algorithme glouton d'assignation

Le premier algorithme d'assignation est du type glouton, c'est-à-dire très rapide. L'analyse d'ordonnançabilité effectuée dans la section précédente étant coûteuse en temps on a proposé, en s'appuyant sur une étude probabiliste, de n'assigner au même processeur que les tâches de même période ou de périodes multiples.

L'étude probabiliste effectuée est la suivante :

À partir du théorème 2.2 on constate que pour deux tâches $(t_a : C(t_a),T(t_a))$ et $(t_b : C(t_b),T(t_b))$ il existe trois cas d'ordonnançabilité :

1. $T(t_a)$ et $T(t_b)$ sont deux entiers premiers entre eux. D'après la remarque 2.1 les tâches t_a et t_b ne sont pas ordonnançables et donc ne peuvent pas être assignées au même processeur ;
2. $T(t_a)$ et $T(t_b)$ sont deux entiers égaux ou multiples. Si $T(t_a) < T(t_b)$ la condition 2.2 devient :

$$C(t_a) + C(t_b) \leq T(t_a) \tag{2.6}$$

3. $T(t_a)$ et $T(t_b)$ ne sont ni multiples ni premiers entre eux. La condition 2.2 reste la même.

Dans l'algorithme d'assignation glouton on ne considère comme ordonnançable que le deuxième cas, c'est à dire que pour cet algorithme le troisième cas est considéré comme non ordonnançable (le premier est évidemment considéré comme non ordonnançable). Les raisons de ce choix sont, premièrement, le besoin de rapidité d'exécution (logiquement, lorsqu'il y a moins de cas à traiter, l'algorithme progresse plus rapidement), ensuite, la comparaison des résultats obtenus en calculant la probabilité de chaque cas pour qu'il soit ordonnançable.

Soit $P(cond_{2.6})$ la probabilité que la condition 2.6 soit satisfaite. $1 \leq C(t_a) \leq T(t_a)$ et $1 \leq C(t_b) \leq T(t_b)$ signifient que la paire $(C(t_a),C(t_b))$ appartient à l'ensemble $\gamma = \{(1,1),(1,2),(2,1),...,(T(t_a),T(t_b))\}$. Si ψ est l'ensemble des paires $(C(t_a),C(t_b))$ qui satisfont la condition 2.6 ($\psi \subset \gamma$) alors :

$$P(cond_{2.6}) = \frac{|\psi|}{|\gamma|} = \frac{\frac{T(t_a)(T(t_a)-1)}{2}}{T(t_a)T(t_b)} = \frac{T(t_a)^2 - T(t_a)}{2T(t_a)T(t_b)}$$

Soit $P(cond_{2.2})$ la probabilité que la condition 2.2 soit satisfaite. De façon similaire au premier calcul on a :

$$P(cond_{2.2}) = \frac{PGCD(T(t_a),T(t_b))^2 - PGCD(T(t_a),T(t_b))}{2T(t_a)T(t_b)}$$

Il est facile de vérifier que $P(cond_{2.6}) > P(cond_{2.2})$. Ce qui signifie que la probabilité que la condition 2.6 soit satisfaite est plus grande que celle pour laquelle la condition 2.2 le soit.

L'algorithme 1 détaille l'algorithme glouton d'assignation.

Algorithme 1 Algorithme glouton d'assignation

1: Trier les tâches suivant un tri à deux niveaux (ce tri est décrit dans la suite)
2: **tant que** Toutes les tâches n'ont pas été assignées **faire**
3: Sélectionner une tâche
4: Assigner cette tâche à un processeur suivant la condition 2.6 (on assigne au même processeur les tâches de même période ou de périodes multiples)
5: Si la tâche n'est assignée à aucun processeur on dit que le système n'est pas ordonnançable
6: **fin tant que**

En ce qui concerne le tri utilisé dans cet algorithme on a d'abord opté pour un tri classique qui consiste en un ordre croissant de périodes. Cependant les premiers tests nous ont montré ses limites puisqu'il ne fait que trier les tâches sans se préoccuper de la condition d'ordonnançabilité (on n'assigne, sur le même processeur, que les tâches avec la même période ou avec des périodes multiples). Pour y remédier on a introduit un tri à deux niveaux qui prend en compte, en même temps, l'ordre croissant des périodes et l'ordre de priorité. Ce dernier est attribué à chaque tâche relativement au nombre de tâches dont la période divise la période de la tâche en question. Après avoir calculé la priorité de chaque tâche, un tri suivant l'ordre des priorités est effectué et quand les priorités sont les mêmes, le tri se fait suivant l'ordre croissant des périodes. D'un côté ce tri permet d'avantager les tâches dont les périodes ont un nombre réduit de diviseurs parmi les autres périodes des tâches, faute de quoi ces tâches ne seraient pas assignées par manque de processeurs et de tâches de périodes divisant sa période. D'un autre côté il permet de pénaliser les tâches dont les périodes ont un nombre important de diviseurs parmi les autres périodes des tâches et qui ont plusieurs possibilités d'assignation. Il est important de noter qu'une tâche peut être assignée à plusieurs processeurs, toutefois le processeur où cette tâche sera ordonnancée est déterminé par l'algorithme d'ordonnancement qui est présenté dans la section 2.2.3.

EXEMPLE 2.7
Les tableaux ci-dessous présentent un système de tâches qu'on tente d'assigner avec la méthode du tri classique (première série de tableaux) ainsi qu'avec la méthode du tri à deux niveaux (deuxième série de tableaux). Ce système est composé de quatre tâches t_a, t_b, t_c, t_d avec, respectivement, quatre valeurs de périodes différentes : 2, 3, 6 et 8. Le graphe d'architecture est composé de deux processeurs (P_1 et P_2) connectés avec un médium de communication. Sur le premier tableau de la première série les tâches sont triées en ordre croissant de périodes. On observe que la tâche de période 8 n'est assignée à aucun des deux processeurs puisque 8 est ni multiple de 6 ni multiple de 3. Sur le premier tableau de la deuxième série les tâches sont triées suivant le tri à deux niveaux. Le niveau de priorité des périodes 2 et 3 est 0 (ces deux valeurs sont premières), la période 8 a un niveau de priorité égal à 1 (8 est multiple de 2) et la période 6 a un niveau de priorité égal à 2 (6 est multiple de 2 et de 3). De cette façon on arrive à assigner toutes les tâches.

1. *le tri classique :*

L'ordre après un tri croissant	*2*	*3*	*6*	*8*
L'ordre des tâches	t_a	t_b	t_c	t_d

(a) *t_a est assignée à P_1*

Processeur P_1	t_a
Processeur P_2	

(b) *t_b ne peut être assignée à P_1 (3 n'est pas un multiple de 2) donc elle est assignée à P_2*

Processeur P_1	t_a
Processeur P_2	t_b

(c) *t_c est assignée à P_1 (6 est un multiple de 2)*

Processeur P_1	t_a, t_c
Processeur P_2	t_b

(d) *t_d n'est pas assignée (8 n'est ni un multiple de 3 ni un multiple de 6)*

Processeur P_1	t_a, t_c
Processeur P_2	t_b

2. *le tri à deux niveaux :*

L'ordre après un tri croissant	*2*	*3*	*6*	*8*
Les niveaux de priorité	*0*	*0*	*2*	*1*
L'ordre final	*2*	*3*	*8*	*6*
L'ordre des tâches	t_a	t_b	t_d	t_c

(a) *t_a est assignée à P_1*

Processeur P_1	t_a
Processeur P_2	

(b) *t_b ne peut être assignée à P_1 (3 n'est pas un multiple de 2) donc elle est assignée à P_2*

Processeur P_1	t_a
Processeur P_2	t_b

(c) *t_d est assignée à P_1 (8 est un multiple de 2)*

Processeur P_1	t_a, t_d
Processeur P_2	t_b

(d) *t_c ne peut être assignée à P_1 (6 n'est pas un multiple de 8) donc elle est assignée à P_2 (6 est un multiple de 3)*

Processeur P_1	t_a, t_d
Processeur P_2	t_b, t_c

À partir de l'exemple 2.7 et quelques autres exemples de base nous avons constaté que le tri à deux niveaux améliore considérablement le taux de fiabilité dans l'ordonnançabilité sans pour autant augmenter le temps d'exécution de l'algorithme. Cette constatation se confirme en comparant les deux complexités, elle passe de $O(n(m+logn)) \simeq O(nlogn)$ avec un tri croissant classique à $O(n(n+m+logn)) \simeq O(n^2)$ avec le tri à deux niveaux (n est le nombre de tâches et m le nombre de processeurs).

Cet algorithme s'est avéré très intéressant en termes de fiabilité dans l'ordonnançabilité dans le cas des systèmes de tâches de périodes similaires ou multiples, par contre il ne considère pas les cas où les tâches périodiques ne sont pas toutes multiples.

2.2.1.3 Algorithme d'assignation de type "Recherche Locale"

En plus du premier algorithme d'assignation présenté dans la section précédente on a proposé un algorithme d'assignation de type RL (Recherche Locale) plus performant en termes de fiabilité dans l'ordonnançabilité quelle que soit les tâches, sachant que cette amélioration peut influer sur le temps d'exécution de l'heuristique d'ordonnancement. L'algorithme d'assignation de type RL est basé donc sur la méthode RL définie dans l'état de l'art, à la section 1.6.3, et il utilise les résultats de l'analyse d'ordonnançabilité de la section 2.2.1.1. À la différence de l'algorithme glouton d'assignation, la recherche locale prend en compte tous les cas possibles d'ordonnançabilité. D'autre part si on arrive à un point où une tâche ne peut être assignée, on remet en cause l'assignation des tâches faite auparavant en désassignant des tâches (les considérer comme non assignées alors qu'elles l'étaient auparavant) jusqu'à ce qu'une assignation soit trouvée pour la tâche en question et les tâches désassignées seront assignées par la suite. De cette manière on passe d'une possibilité à une autre jusqu'à ce qu'une solution qui offre une assignation à toutes les tâches soit trouvée. Les critères d'arrêt de l'algorithme RL sont, soit de trouver une solution regroupant toutes les tâches, soit de revenir à une configuration déjà explorée avant, et dans ce cas on considère que l'assignation n'existe pas et que le système traité est non ordonnançable.

L'algorithme 2 détaille l'algorithme de la recherche locale.

Algorithme 2 Algorithme d'assignation de type "Recherche Locale"

```
1: Trier les tâches suivant l'ordre croissant des périodes
2: tant que Toutes les tâches n'ont pas été assignées faire
3:    Sélectionner une tâche
4:    Assigner cette tâche à tous les processeurs qui satisfont la condition du corollaire 1
5:    si Aucun processeur ne satisfait la condition du corollaire 1 alors
6:       si L'assignation actuelle a déjà été visitée alors
7:          Sortir du programme : aucune assignation n'est possible avec cet algorithme
8:       sinon
9:          Désassigner la dernière tâche assignée à chaque processeur
10:         Aller à l'étape 4
11:      fin si
12:   fin si
13: fin tant que
```

REMARQUE 2.5
Comme c'est le cas dans l'algorithme glouton d'assignation, une tâche peut être assignée à plusieurs processeurs. La dernière partie de l'heuristique d'ordonnancement, qui est l'algorithme de l'ordonnancement, décidera du choix du processeur où une tâche multi-assignée sera ordonnancée.

2.2.2 Déroulement

Cette partie consiste à répéter chaque tâche t_i du système r_i fois où $r_i = hp/T(t_i)$ (hp étant l'hyper-période) [148]. Ce déroulement permet de transformer le graphe d'algorithme initial en un deuxième graphe qui comprend toutes les tâches (non-périodiques cette fois) qui doivent être ordonnancées sur un intervalle de temps égal à l'hyper-période. De cette façon on considère le système périodique tel qu'il se répète réellement [140], c'est-à-dire que dans ce type de système les tâches s'exécutent de la même manière suivant un intervalle de temps dont la longueur est égale à l'hyper-période. Cependant ce graphe déroulé ne sera achevé qu'après l'ajout des dépendances telles qu'elle apparaissent dans le graphe initial. Cette opération est détaillée dans la prochaine section.

2.2.2.1 Contrainte de périodicité et transfert de données

Conformément à ce qui a été énoncé dans la section 2.1.1.2 on a opté pour un modèle sans perte de données. Lorsque deux tâches sont dépendantes et ne possèdent pas la même période deux cas de figure se présentent :

1. si la période de la tâche consommatrice est égale à q fois la période de la tâche productrice alors la tâche productrice doit être exécutée q fois pour une seule exécution de la tâche consommatrice. De plus une instance de la tâche consommatrice ne peut démarrer son exécution qu'après l'exécution des q instances de la tâche productrice ;
2. si la période de la tâche productrice est égale à q fois la période de la tâche consommatrice alors la tâche consommatrice doit être exécutée q fois pour une seule exécution de la tâche productrice.

Afin de satisfaire ces deux situations il faut ajouter des arcs :

- de précédence entre les instances d'une même tâche,
- de dépendance entre les instances d'une tâche productrice et les instances d'une tâche consommatrice de la façon suivante : si $T(t_p)$ est la période de la tâche productrice et $T(t_c)$ est la période de la tâche consommatrice alors :
  - si $T(t_c) > T(t_p)$ alors : $T(t_c)/T(t_p)$ instances de la tâche productrice seront connectées à une instance de la tâche consommatrice,
  - si $T(t_p) > T(t_c)$ alors : $T(t_p)/T(t_c)$ instances de la tâche consommatrice seront connectées à une instance de la tâche productrice.

2.2.2.2 Algorithme du déroulement

L'algorithme 3 détaille le déroulement.

EXEMPLE 2.8
Pour illustrer l'algorithme du déroulement nous proposons de l'exécuter sur le graphe d'algorithme de la figure 2.7. Les périodes des tâches t_a, t_b, t_c et t_d sont les suivantes : $T(t_a) = 2$, $T(t_b) = 4$, $T(t_c) = 6$ et $T(t_d) = 12$.

Algorithme 3 Algorithme de déroulement

1: Dupliquer chaque tâche suivant la valeur du rapport entre l'hyper-période et sa période
2: Ajouter une précédence entre chaque paire d'instances successives de la même tâche
3: Ajouter les dépendances entre les instances des tâches productrices et celles des tâches consommatrices

La valeur de l'hyper-période est, donc, égale à 12. n_a=12/2=6, n_b=12/4=3, n_c=12/6=2 *et* n_d=12/12=1 *sont respectivement le nombre d'instances des tâches* t_a, t_b, t_c *et* t_d. *Après calcul on trouve qu'une instance de la tâche* t_B *dépend de 2 instances de la tâche* t_a, *une instance de la tâche* t_c *dépend de 3 instances de la tâche* t_a *et l'instance de la tâche* t_d *dépend de 3 instances de la tâche* t_b *ainsi que de 2 instances de la tâche* t_c. *Une fois que tous les arcs sont ajoutés on obtient le graphe déroulé de la figure 2.8.*

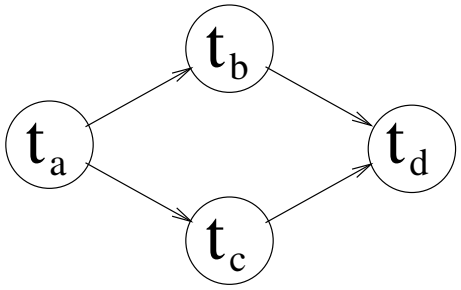

FIG. 2.7 – *(Exemple 2.8) Graphe d'algorithme*

2.2.3 Ordonnancement

L'algorithme d'ordonnancement exploite les résultats des algorithmes d'assignation et de déroulement. Il a pour but d'ordonnancer les tâches du graphe déroulé - en respectant les dépendances entre les tâches - sur les processeurs où elles ont été assignées. On rappelle que toutes les instances d'une même tâche seront ordonnancées sur le processeur où cette tâche a été assignée. Les tâches assignées à plusieurs processeurs seront ordonnancées sur le processeur qui permet de minimiser le makespan.

Cet algorithme est fondé sur la méthode de liste (abordé dans l'état de l'art dans la section 1.4.2). À chaque étape de l'algorithme une tâche est sélectionnée dans un ensemble de tâches candidates pour être ordonnancées. Cet ensemble regroupe les tâches, dans le graphe déroulé, qui n'ont pas de prédécesseurs ou dont tous leurs prédécesseurs ont déjà été ordonnancés. De cette façon l'ordonnancement respecte la contrainte de précédence puisqu'une tâche n'est ordonnancée qu'après que tous ses prédécesseurs aient été ordonnancés. En outre pour minimiser le makespan, l'algorithme d'ordonnancement utilise une fonction de coût qui prend en compte les durées d'exécution des tâches et les coûts des communications (entre les tâches dépendantes ordonnancées sur des processeurs différents). La fonction de coût est détaillée à la section 6.1.4.2.

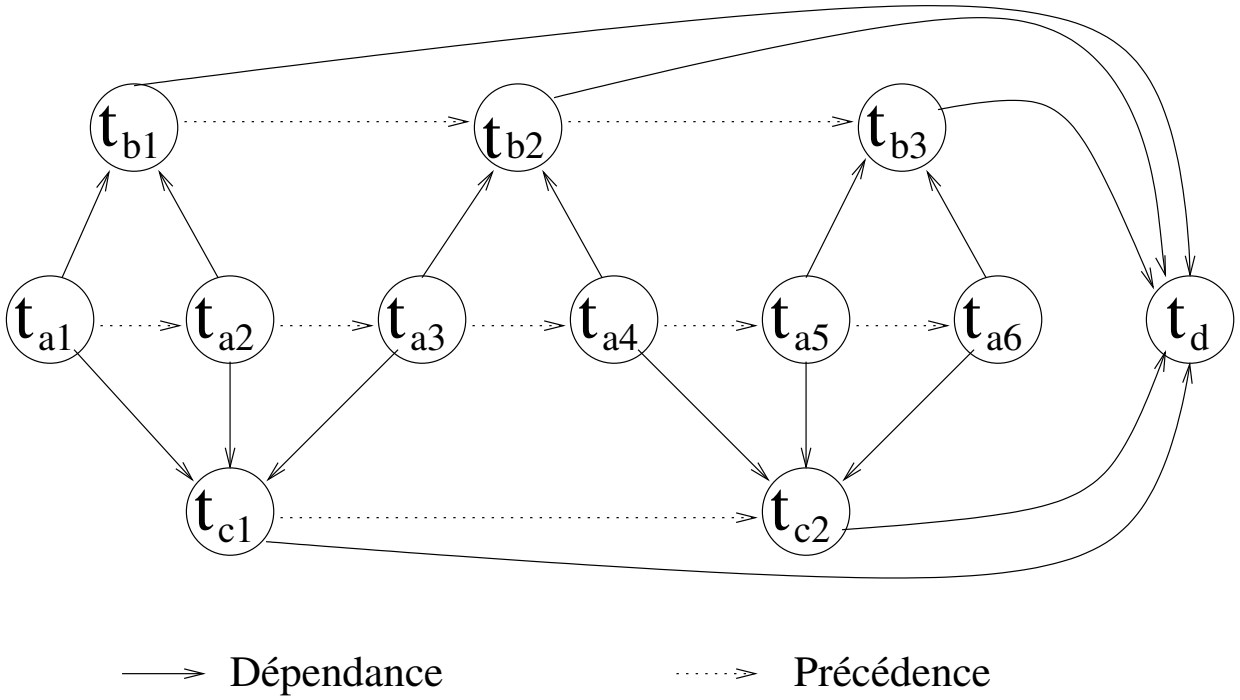

FIG. 2.8 – *(Exemple 2.8) Graphe déroulé résultant*

En plus de la contrainte de précédence et de la minimisation du makespan, prises en compte par l'algorithme de liste, on devait modifier cet algorithme pour s'assurer du respect de la contrainte de périodicité. En effet l'algorithme du déroulement nous permet d'avoir le bon nombre d'instances pour chaque tâche, néanmoins, elle ne garantit pas que les dates de début d'exécution de ces instances respectent la contrainte de périodicité stricte. Deux ajouts ont été nécessaires :

1. dès qu'une tâche t_i est ordonnancée toutes ses instances prennent des dates de début d'exécution fixes calculées en fonction de $T(t_i)$ (la période de la tâche) et q le numéro de répétition de l'instance t_{i_q}, tel que :
$$S(t_{i_q}) = S(t_{i_1}) + T(t_i)(q-1)$$
2. pour qu'elle soit ordonnancée une tâche doit satisfaire une condition qui est détaillée dans le théorème qui suit.

Le théorème suivant introduit une condition qui n'est testée que pour les premières instances des tâches et qui permet de vérifier si cette instance ainsi que les instances suivantes peuvent être ordonnancées aux dates définies par l'algorithme d'ordonnancement. Si la condition n'est pas satisfaite alors la date de début d'exécution de cette tâche sera retardée le temps que l'algorithme lui trouve un intervalle où la condition sera satisfaite. Comme les conditions d'assignation, vues précédemment, sont satisfaites on est certain de trouver un intervalle pour chaque tâche, néanmoins il se peut que l'algorithme effectue plusieurs tentatives pour parvenir à les ordonnancer provoquant à chaque fois un retard.

THÉORÈME 2.10
Soient $\{(t_i : C(t_i),T(t_i),S(t_i)),i = 1..n\}$ n tâches déjà ordonnancées sur un processeur. La tâche $\{(t_j : C(t_j),T(t_j)\}$ est ordonnançable à la date de début d'exécution $S(t_j)$ sur ce processeur si et seulement si :

$$\forall i,\ S(t_i) \leq (S(t_j) - S(t_i)) \bmod g_i \leq (g_i - C(t_j)) \tag{2.7}$$

où g_i =PGCD$(T(t_i),T(t_j))$.

Preuve
Pour prouver le théorème 2.10 il suffit de prouver que deux tâches $\{(t_a : C(t_a),T(t_a),S(t_a))\}$ et $\{(t_b : C(t_b),T(t_b),S(t_b))\}$ sont ordonnançables sur le même processeur si et seulement si :

$$C(t_a) \leq (S(t_b) - S(t_a)) \bmod g \leq (g - C(t_b)) \tag{2.8}$$

où $g = PGCD(T(t_a),T(t_b))$.

Sans perte de généralité on admet que $S(t_a) = 0$. Nous démontrons que la condition 2.8 est une condition suffisante de la manière suivante : soit I l'ensemble d'intervalles suivant $I = \{\forall l \in \mathbb{N},[0 + lg,g + lg[\}$, les premières $C(t_a)$ unités de temps de chacun de ces intervalles peuvent être utilisées pour exécuter t_a ainsi que ses instances avec une exécution tous les $T(t_a)/g$ intervalles. Il va subsister $g - C(t_a)$ unités de temps pour exécuter t_b tous les $T(t_b)/g$ intervalles. Si la condition 2.8 est vérifiée alors les $g - C(t_a)$ unités de temps suffisent à exécuter t_b et donc les deux tâches sont ordonnançables.

La nécessité de 2.8 est démontrable de la manière suivante : comme pour la première partie, soit I l'ensemble d'intervalles $I = \{\forall l \in \mathbb{N},[0 + lg,g + lg[\}$. Supposons que la condition 2.8 n'est pas vérifiée et que t_a et t_b sont ordonnançables quand même. Dire que la condition 2.8 n'est pas vérifiée signifie que l'intervalle de longueur $C(t_b)$, où t_b est exécutée, chevauche les $C(t_a)$ premières unités d'un intervalle dans T tous les $T(t_b)/g$ intervalles. D'un autre côté on note que les $C(t_a)$ unités de temps d'un intervalle dans I tous les $T(t_a)/g$ intervalles sont utilisées pour exécuter t_a. Comme le $PGCD(T(t_a)/g,T(t_b)/g) = 1$ il y a au moins un intervalle dans l'ensemble I où t_a et t_b sont toutes les deux exécutées. Par conséquent si 2.8 n'est pas vérifiée alors t_a et t_b ne sont pas ordonnançables sur le même processeur, ce qui contredit la supposition de départ. Ceci complète la preuve du théorème□

REMARQUE 2.6
Le théorème 2.2 peut être considéré comme un corollaire du théorème 2.10.

En introduisant la condition d'ordonnançabilité du théorème 2.10 dans l'algorithme de liste on obtient un algorithme d'ordonnancement de tâches périodiques sous la contrainte de précédence avec la minimisation du makespan. L'algorithme 4 détaille l'ordonnancement (l'urgence pour un ordonnancement est définie à la section 6.1.4.2).

Lorsqu'une tâche est ordonnancée sur un processeur, si cette tâche (qu'on appelle la tâche consommatrice) dépend d'une ou de plusieurs tâches productrices déjà ordonnancées sur un autre processeur, alors des tâches réceptrices (en nombre égal aux dépendances qu'a la tâche consommatrice) sont créées. D'un autre côté une tâche émettrice est créée pour chaque tâche productrice.

Algorithme 4 Algorithme d'ordonnancement

1: Initialisation de l'ensemble Δ des tâches candidates à l'ordonnancement
2: **tant que** Δ n'est pas vide **faire**
3: Sélectionner la tâche t_i dont l'ordonnancement est le plus urgent parmi les tâches candidates de l'ensemble Δ
4: **si** La tâche t_{i_1} est la première instance de la tâche t_i **alors**
5: **si** La tâche t_i a été assignée à un seul processeur **alors**
6: Ordonnancer l'instance t_{i_1} sur ce processeur si elle satisfait la condition du théorème 2.10 sinon elle est remise dans l'ensemble Δ pour être ordonnancée plus tard
7: Calculer les dates de début d'exécution des autres instances de cette tâche

$$S(t_{i_q}) = S(t_{i_1}) + T(t_i)(q-1)$$

8: **sinon**
9: Ordonnancer la tâche t_i sur le processeur où elle satisfait la condition du théorème 2.10 (si cette condition n'est possible sur aucun processeur alors elle est remise dans l'ensemble Δ pour être ordonnancée plus tard) et si plusieurs processeurs peuvent être choisis alors cette instance est ordonnancée sur celui pour lequel la fonction de coût, utilisée pour minimiser le makespan, est minimale
10: **fin si**
11: Calculer les dates de début d'exécution des autres instances de cette tâche
12: **sinon**
13: Ordonnancer cette tâche directement à la date de début d'exécution qui a été calculée au moment de l'ordonnancement de la première instance
14: **fin si**
15: Supprimer l'instance t_{i_1} de l'ensemble Δ
16: Actualiser l'ensemble Δ en ajoutant les nouvelles tâches candidates
17: **fin tant que**

Le transfert de données associé à cette dépendance est réalisé en échangeant des données entre les tâches réceptrices et les tâches émettrices à travers le médium de communication qui relie les deux processeurs. Le coût de communication est déterminé par le temps qui s'écoule du début d'exécution de la tâche émettrice à la fin d'exécution de la tâche réceptrice. Ces deux tâches correspondent, dans le cas d'une communication par mémoire partagée, à une lecture mémoire pour la tâche émettrice et une écriture mémoire pour la tâche réceptrice. La durée d'exécution d'une tâche émettrice (resp. réceptrice) est ajoutée à la durée d'exécution de la tâche productrice (resp. consommatrice). Pour expliquer ce principe on prend un exemple de trois tâches t_a, t_b et t_c telles que t_a et t_b produisent des données pour t_c. On admet que t_a et t_b ont été ordonnancées sur le processeur P_j alors que t_c a été ordonnancée sur un le processeur P_i. La figure 2.9 décrit la façon dont

ces trois tâches ont été ordonnancées ($C(t_a)$=$C(t_b)$=$C(t_c)$=1 et le coût d'une communication entre processeurs est égal à 1 aussi). On remarque que la tâche t_c ne commence son exécution qu'une fois que toutes les données, dont elle a besoin pour s'exécuter, sont disponibles.

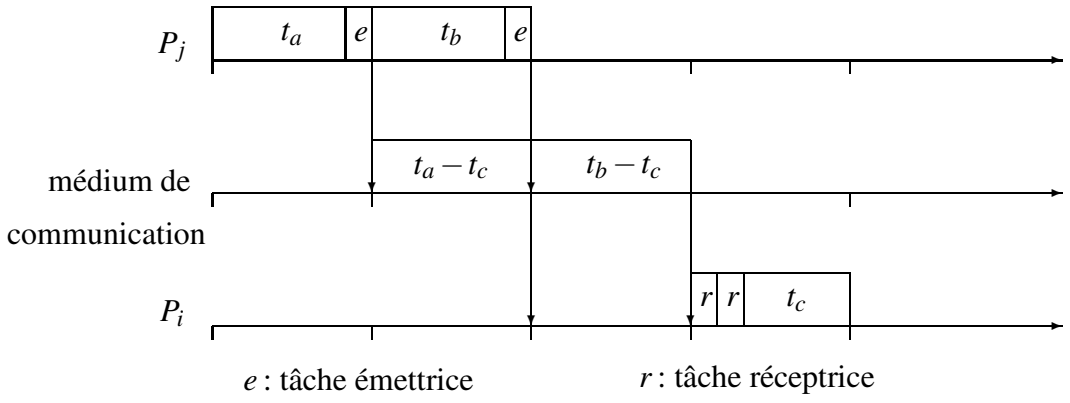

FIG. 2.9 – *Ordonnancement de tâches dépendantes sur des processeurs distincts*

Désormais les parties qui composent l'heuristique d'ordonnancement ont été détaillées. On propose de reprendre chacune de ces parties par le biais d'un exemple.

EXEMPLE 2.9
Pour illustrer l'heuristique d'ordonnancement proposée on utilise le système de la figure 2.10. Ce système est composé d'un graphe d'algorithme et d'un graphe d'architecture. Le graphe d'algorithme est composé de quatre tâches définies comme suit : $(t_a : 2,1)$, $(t_b : 4,1)$, $(t_c : 6,1)$ *et* $(t_d : 12,1)$. *Le graphe d'architecture est composé de deux processeurs* P_1 *et* P_2 *connectés par le médium de communication "med" (le coût d'une communication entre* P_1 *et* P_2 *est égal à 1).*

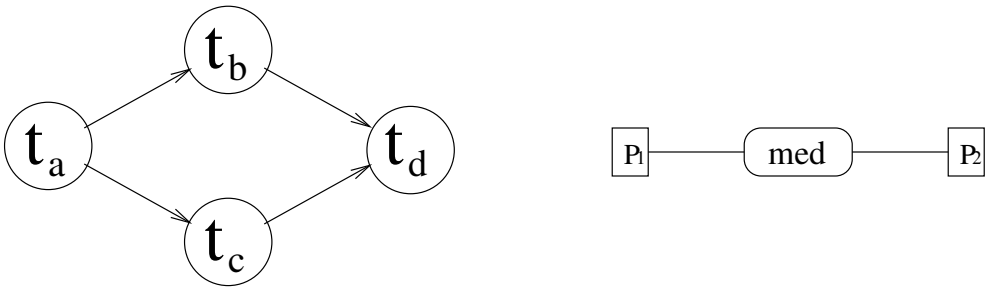

FIG. 2.10 – *(Exemple 2.9) Système temps réel composé d'un graphe d'algorithme et d'un graphe d'architecture*

L'heuristique fonctionne de la manière suivante :

– *l'exécution de l'algorithme d'assignation a produit le résultat suivant : les tâches* t_a *et* t_b *ont été assignées au processeur* P_1, *la tâche* t_c *a été assignée au processeur* P_2 *et finalement* t_d *a*

été assignée aux deux processeurs P_1 et P_2,

- *le déroulement a été effectué dans l'exemple 2.8 et le résultat est donné par la figure 2.8,*
- *l'algorithme d'ordonnancement avec fonction de coût (Algorithme 4) opère comme suit :*
  - *au départ $\Delta = \{t_{a_1}\}$, la tâche t_{a_1} est ordonnancée sur le processeur P_1 où elle a été assignée,*
  - *comme la date de début d'exécution de t_{a_1} est connue $S(t_{a_1}) = 0$, les dates de début d'exécution des instances suivantes de t_a qu'on obtient après calcul sont : $S(t_{a_2}) = 2$, $S(t_{a_3}) = 4$, $S(t_{a_4}) = 6$, $S(t_{a_5}) = 8$, $S(t_{a_6}) = 10$,*
  - *la tâche t_{a_1} est supprimée de l'ensemble Δ qui est, ensuite, actualisé en incluant la tâche t_{a_2},*
  - *la tâche t_{a_2} est sélectionnée et ordonnancée sur le processeur P_1 à la date $S(t_{a_2}) = 2$,*
  - *la tâche t_{a_2} est supprimée de l'ensemble Δ qui est, ensuite, actualisé en incluant la tâche t_{b_1},*
  - *la tâche t_{b_1} est sélectionnée et ordonnancée sur le processeur P_1 où elle a été assignée, les dates de début d'exécution des instances suivantes de t_b qu'on obtient sont : $S(t_{b_2}) = 3$, $S(t_{b_3}) = 7$, $S(t_{b_4}) = 11$,*
  - *la tâche t_{b_1} est supprimée de l'ensemble Δ qui est, ensuite, actualisé en incluant la tâche t_{b_2} à cet ensemble,*
  - *de la même manière les tâches $t_{c_1}, t_{a_2}, t_{b_2}, t_{a_3}, t_{a_4}, t_{c_1}, t_{b_3}, t_{a_5}, t_{a_6}, t_{b_4}, t_{c_2}$ sont ordonnancées,*
  - *la tâche t_d est la seule tâche qui a été assignée aux deux processeurs et qui est finalement ordonnancée sur le processeur P_2 en utilisant la fonction de coût dont l'intérêt est montré plus loin.*
  - *l'exécution de l'algorithme 4 produit, ainsi, un ordonnancement de toutes les tâches qui respectent leurs périodes et prend en compte les coûts de communications entre les tâches dépendantes ordonnancées sur des processeurs différents (voir la figure 2.11).*

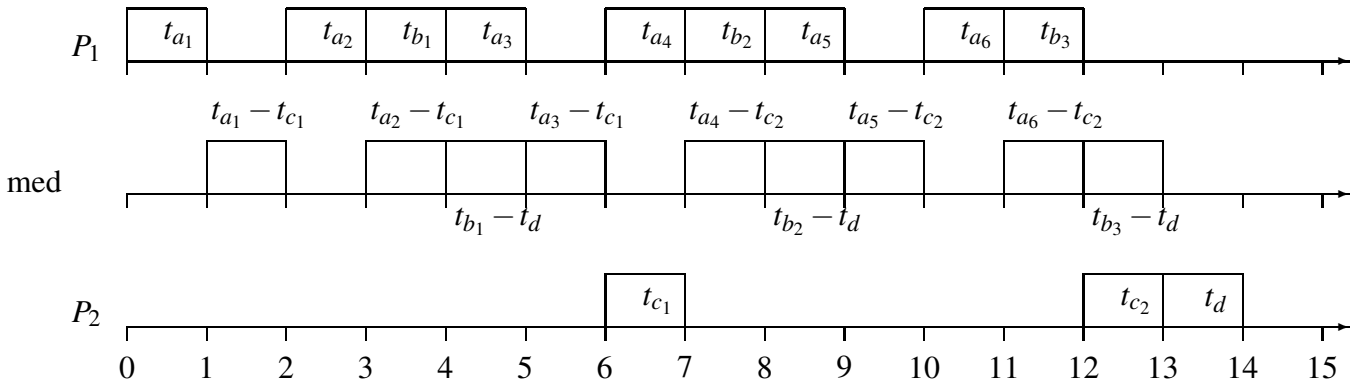

FIG. 2.11 – *Ordonnancement avec l'utilisation de la fonction de coût*

– *l'algorithme d'ordonnancement sans fonction de coût opère similairement sauf que comme la tâche t_d est assignée aux deux processeurs elle est ordonnancée sur l'un d'eux pris au hasard.*

La figure 2.12 expose l'ordonnancement qu'on obtient. On remarque, en ce qui concerne la valeur du makespan résultant qu'elle passe de 14 *dans le cas de l'ordonnancement avec fonction de coût à* 18 *dans le deuxième cas. Notons que $t^2_{a_1}, t^2_{a_2}, t^2_{a_3}$ et $t^2_{b_1}$ sont des instances qui appartiennent à la deuxième exécution du système et la tâche t_d ne peut être exécutée avant que l'algorithme d'ordonnancement, en se servant se la condition du théorème 2.10, ne lui trouve un intervalle où les périodes de t_a et t_b sont satisfaites. On dit, dans ce cas, que l'exécution de t_d a été retardée. La prochaine section détaille ce cas de figure ainsi que ses conséquences.*

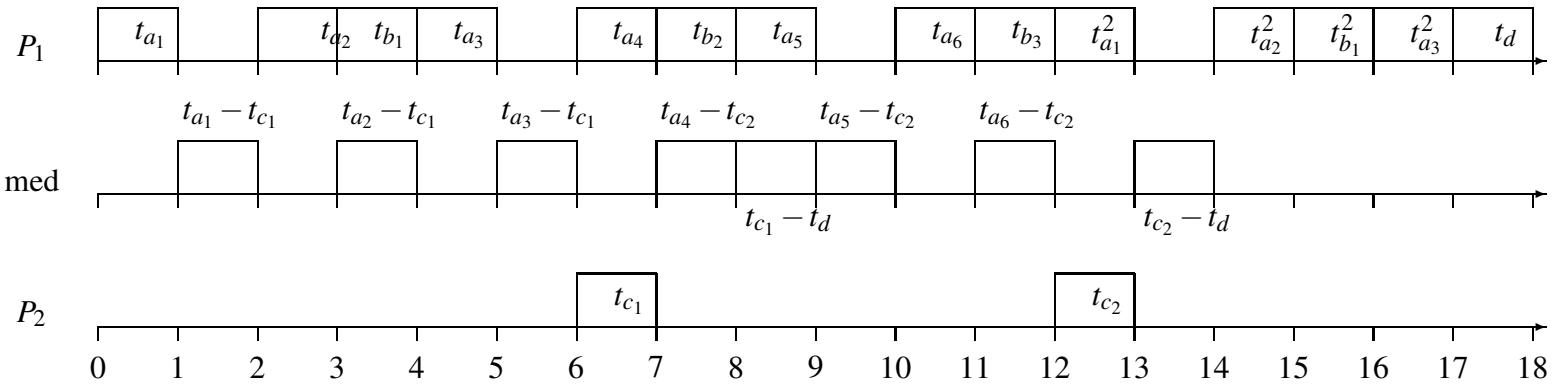

FIG. 2.12 – *Ordonnancement sans l'utilisation de la fonction de coût*

2.2.4 Phase transitoire et phase permanente

À partir de maintenant pour chaque tâche nous utilisons un nouvel indice supérieur (x pour $t^x_{a_q}$) dénotant le numéro d'exécution du système.

Dans l'exemple d'ordonnancement 2.9 la valeur de l'hyper-période est égale à 12 et on voit sur la figure 2.12 qu'elle commence sur le processeur P_1 à la date 0 (la date de début d'exécution de la tâche $t^1_{a_1}$) et finit à la date 12. Cette dernière correspond à la date de début d'exécution de la tâche $t^2_{a_1}$ qui représente, comme c'est le cas pour $t^1_{a_1}$ dans la première exécution du système, la première tâche à s'exécuter dans la deuxième exécution du système. Par ailleurs sur le processeur P_2 l'hyper-période commence à la date 6 (la date de début d'exécution de la tâche $t^1_{c_1}$), à cause des dépendances entre les trois instances de t_a et la tâche $t^1_{c_1}$ et finit à la date 18. De la même manière que sur le premier processeur, la date 18 correspond à la date de début d'exécution de la tâche $t^2_{c_1}$ qui fait partie de la deuxième exécution du système. On en déduit que l'hyper-période est la même sur chaque processeur mais, à cause des communications entre les processeurs, leurs départs peuvent être décalés. L'exécution du système de la figure 2.10 est équivalent à une exécution infinie des tâches du graphe déroulé telles qu'elles sont ordonnancées sur les processeurs P_1 et P_2. On constate

aussi que la tâche t_d s'exécute, pour cause de dépendance avec t_c et de périodicité ($t^2_{a_2}$ et $t^2_{b_1}$ doivent être exécutées respectivement aux dates 14 et 15), à la date 16 alors que l'hyper-période sur ce processeur commence à la date 0 (la date de début d'exécution de la tâche $t^1_{a_1}$) et finit à la date 12 (la date de début d'exécution de la tâche $t^2_{a_1}$).

Ceci est dû au fait que dans certains cas, à cause de la contrainte de périodicité stricte et/ou aux communications intra ou inter-processeurs, au lieu d'avoir toutes les tâches, qui sont assignées au même processeur, exécutées dans un intervalle de temps égal à l'hyper-période, quelques unes peuvent n'être exécutées qu'après cet intervalle.

Prenons un exemple plus simple en monoprocesseur pour mieux observer ce fait. Admettons que les tâches du graphe d'algorithme de la figure 2.13 sont définies comme suit : $(t_a : 1,2)$, $(t_b : 1,4)$ et $(t_c : 1,4)$. L'ordonnancement de ce système est donné par la figure 2.14. On constate qu'au départ il y a une séquence d'ordonnancement qui ne se répète qu'une seule fois et qui ne contient qu'une partie des tâches. Ensuite elle est remplacée par une deuxième séquence, qui contient toutes les tâches et qui se répète indéfiniment.

FIG. 2.13 – *Graphe d'algorithme*

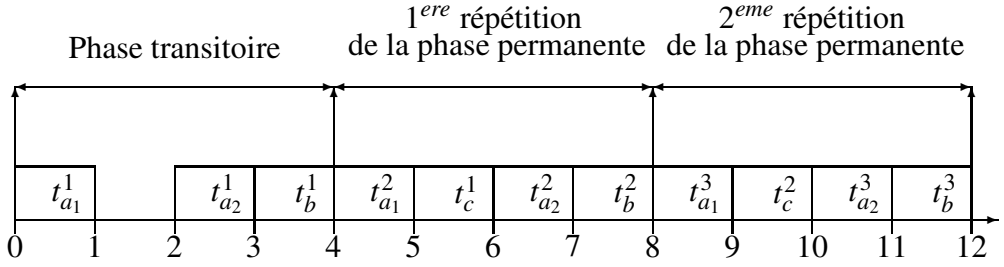

FIG. 2.14 – *Phases d'ordonnancement*

On a donc un ordonnancement en deux phases :

1. la phase transitoire de longueur égale à l'hyper-période et qui représente la séquence d'ordonnancement qui ne se répète qu'une seule fois,
2. la phase permanente de longueur égale aussi à l'hyper-période et qui représente la phase qui se répètent indéfiniment.

Dans [3] Leung et Merill présentent un résultat qui démontre qu'il ne peut y avoir plus de deux phases d'ordonnancement.

2.3 Algorithme exact

L'algorithme exact qui nous a semblé le plus adéquat pour résoudre le problème traité dans ce chapitre est le "Branch and Cut" [149] (B&C). Cet algorithme trouve, si elle existe, une solution quel que soit le problème d'ordonnancement posé. Dans le cas où cet algorithme ne trouve pas de solutions à ce problème aucun autre algorithme n'est capable d'en trouver et, ainsi, on peut dire que ce problème n'a pas de solution. Afin de respecter les précédences, l'algorithme utilise la même méthode que l'heuristique vue précédemment. Cela consiste à regrouper les tâches qui n'ont pas de prédécesseurs ou dont tous leurs prédécesseurs ont déjà été ordonnancés dans un ensemble de candidats qui est mis à jour chaque fois qu'une de ses tâches est ordonnancée. Pour qu'une tâche soit ordonnancée elle doit satisfaire les conditions d'ordonnançabilité du corollaire 1 et du théorème 2.10. La figure 2.15 montre la manière dont l'algorithme explore l'espace de solutions. Pour chaque tâche candidate à l'ordonnancement, l'algorithme teste son ordonnançabilité sur chaque processeur, puis il choisit le premier sur lequel les conditions sont satisfaites. Si la tâche t_i n'est ordonnançable sur aucun processeur alors l'algorithme fait un retour arrière (backtrack), il revient à la tâche t_{i-1} qui a été ordonnancée juste avant et il lui cherche un autre processeur que celui ou elle a été ordonnancée avant. À chaque fois qu'il ordonnance la tâche t_{i-1} il cherche à ordonnancer la tâche t_i, et après avoir testé la tâche t_{i-1} sur tous les processeurs mais il n'arrive toujours pas à ordonnancer la tâche t_i, l'algorithme B&C fait un deuxième retour arrière pour remonter à la tâche t_{i-2}. De cette manière l'espace de recherche représente un arbre de hauteur n (n est le nombre de tâches) et chaque noeud a exactement m fils (m est le nombre de processeurs). L'algorithme B&C effectue un parcours en profondeur sur l'arbre de recherche. Dès qu'il arrive à un noeud qui comporte une solution non ordonnançable il coupe la branche qui se forme à partir de ce noeud pour remonter ensuite, en faisant un retour arrière, dans l'arbre pour parcourir d'autres solutions. Le but est d'arriver à la tâche t_n en évitant d'explorer les branches qui ont comme racine une partie d'une solution non ordonnançable. L'algorithme 5 détaille l'algorithme B&C.

Algorithme 5 Algorithme B&C d'ordonnancement

1: Initialisation de l'ensemble Δ des tâches candidates à l'ordonnancement
2: **tant que** L'ensemble Δ n'est pas vide **faire**
3: Ordonnancer la tâche candidate t_i sur le premier processeur où les conditions du corollaire 1 et du théorème 2.10 sont satisfaites et où elle n'a pas été ordonnancée avant
4: **si** la tâche n'est ordonnançable sur aucun processeur **alors**
5: Remettre la tâche t_{i-1} (la tâche qui a été ordonnancée avant t_i) dans l'ensemble Δ
6: **sinon**
7: Supprimer la tâche t_i de l'ensemble Δ
8: Actualiser l'ensemble Δ en ajoutant les nouvelles tâches candidates
9: **fin si**
10: **fin tant que**

FIG. 2.15 – *Principe du "branch and cut"*

EXEMPLE 2.10
Pour illustrer l'exécution de l'algorithme B&C on l'applique sur le système de la figure 2.16. Ce système est composé d'un graphe d'algorithme et d'un graphe d'architecture. Le graphe d'algorithme est composé de quatre tâches définies comme suit : $(t_a : 2,1)$, $(t_b : 3,1)$, $(t_c : 6,1)$ *et* $(t_d : 8,1)$. *Le graphe d'architecture est composé de deux processeurs* P_1 *et* P_2 *connectés par le médium de communication "med" (le coût d'une communication entre* P_1 *et* P_2 *est égal à 1).*

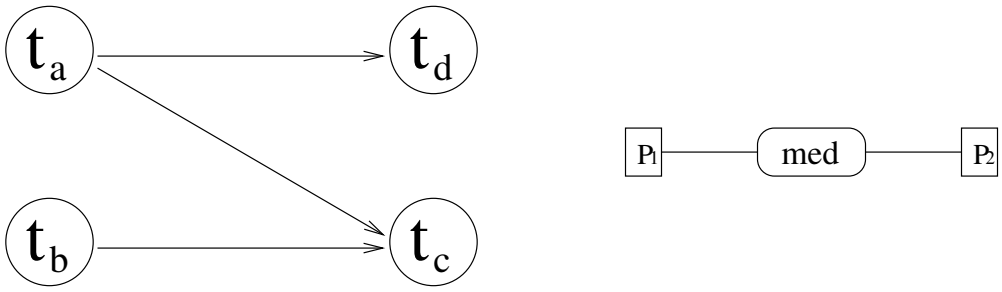

FIG. 2.16 – *(Exemple 2.10) Système temps réel*

La figure 2.17 décrit les huit étapes nécessaires à l'algorithme B&C pour ordonnancer ce système. Ces étapes montrent le parcours que suit l'algorithme B&C dans l'arbre de recherche.

À chaque noeud l'algorithme vérifie les conditions du corollaire 1 et du théorème 2.10 pour décider de l'éventualité de parcourir les fils de ce noeud ou bien de remonter dans l'arbre. Les noeuds où ces conditions sont satisfaites sont marqués avec un smile et ceux où elles ne le sont pas sont marqués avec une croix. L'ordonnancement obtenu est décrit dans la figure 2.18.

2.4 Comparaisons

L'heuristique d'ordonnancement proposée dans ce chapitre a été implémentée dans un logiciel appelé SynDEx (tous les détails sont donnés dans la deuxième partie du manuscrit). Les deux algorithmes d'assignation (glouton et du type recherche locale) ont été implémentés afin de pouvoir tester les performances de chaque approche. Comme on ne dispose d'aucun benchmark sur le sujet qu'on traite on a également implémenté l'algorithme exact qui nous a servi de référence pour des exemples de petite taille étant donné que son temps d'exécution explose rapidement. En définitive nous avons comparé trois algorithmes différents.

Les graphes d'algorithme utilisés pour les tests ont été générés automatiquement en prenant en compte la restriction qui impose que les tâches communiquantes doivent être à la même période ou à des périodes multiples (ceci n'exclut pas la présence de périodes non multiples dans le même graphe). Les périodes des tâches sont, ainsi, générées automatiquement de telle manière que le nombre de différentes valeurs de périodes soit assez réduit relativement au nombre de tâches. Par conséquent chaque tâche a une période en plus d'une durée d'exécution donnée aléatoirement à partir d'une liste de valeurs nécessairement inférieures à la période (voir la condition 2.1). Par ailleurs, pour des raisons de simplification, dans le graphe d'architecture généré chaque paire de

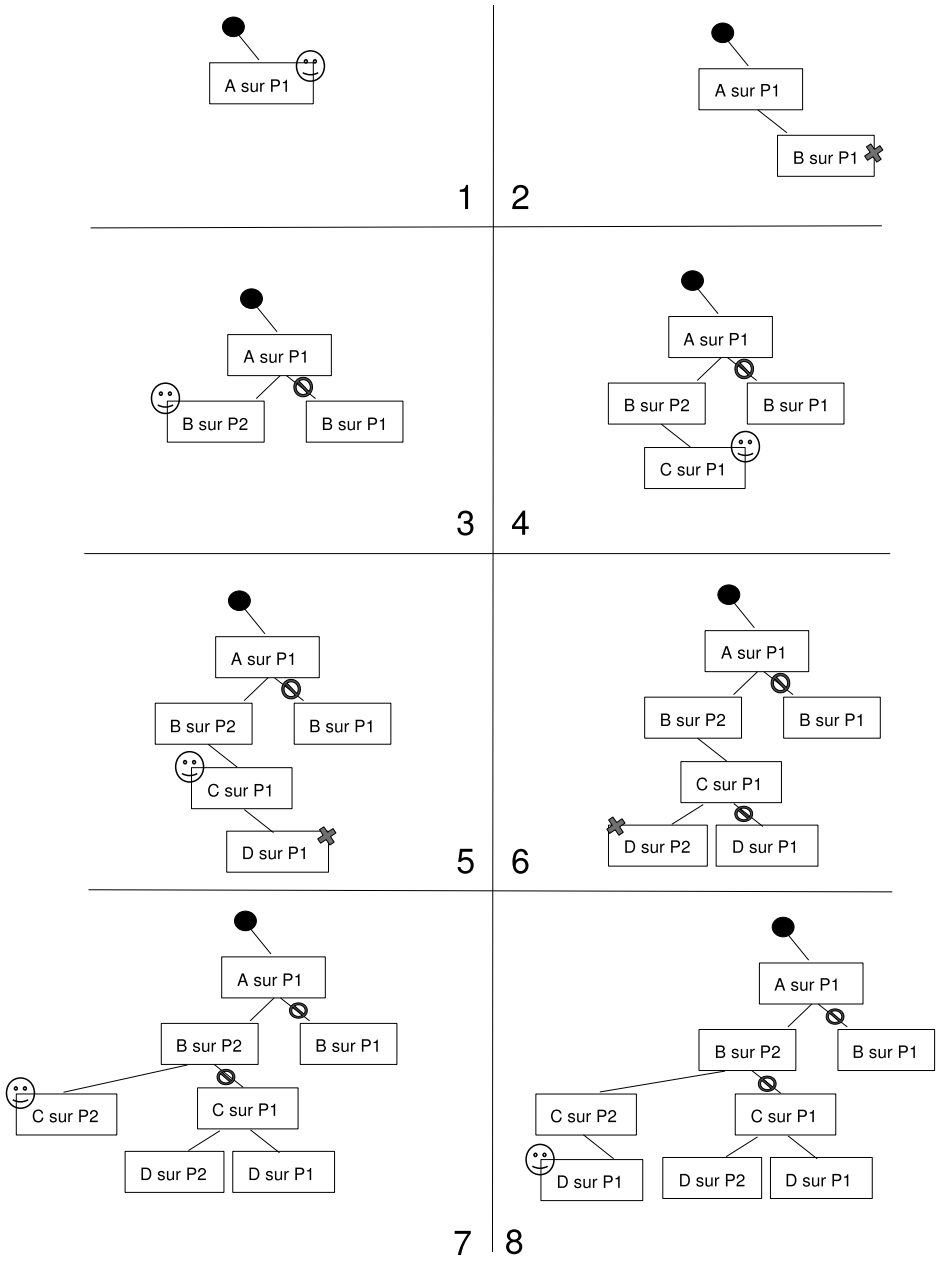

FIG. 2.17 – *(Exemple 2.10) Déroulement de l'exécution de l'algorithme du "branch and cut"*

processeurs est reliée par un médium de communication (pas de routage entre les processeurs). Le nombre de processeurs de l'architecture est fixé en fonction de l'exécution de l'algorithme exact de la manière suivante : tant que l'application n'est pas ordonnançable on rajoute un processeur puis on refait le test.

Puisque le problème traité est en même temps un problème d'optimisation (optimiser le temps d'exécution de l'algorithme) et un problème de décision (le taux de fiabilité de l'algorithme doit

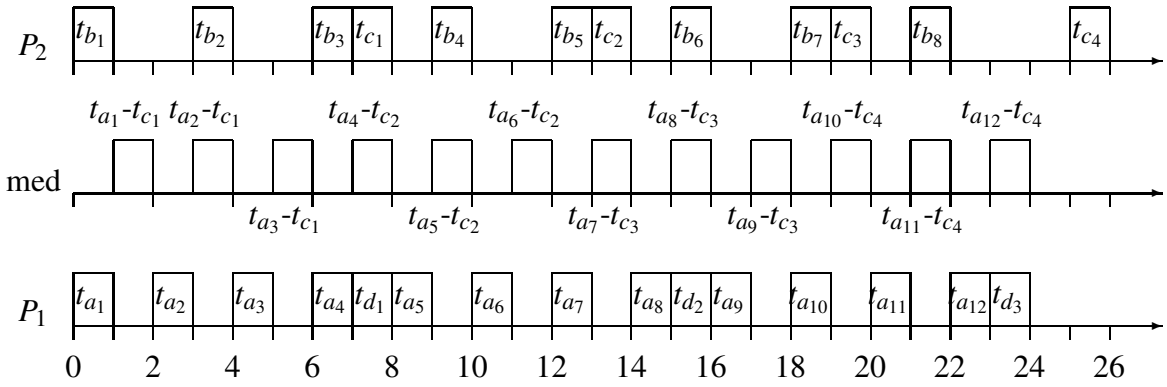

FIG. 2.18 – *Ordonnancement de tâches dépendantes sur des processeurs distincts*

être le plus élevé possible) nous avons réalisé deux catégories de tests. La première catégorie s'intéresse au taux de fiabilité des algorithmes en termes d'ordonnançabilité et la deuxième catégorie s'intéresse au temps d'exécution des algorithmes.

En ce qui concerne les details techniques, le PC utilisé pour les tests est un DELL muni d'un processeur Intel$Core^{TM}2Duo(2{,}66GHz,mmoirecachede12Mo,busfrontalde1333MHz)etunemmoirede4GoDDR$
$unit->float''quiestlimiteaucentimedeseconde(touslesalgorithmesonttimplmentsenOCAML)$.

2.4.1 Comparaison des taux de fiabilité des algorithmes

Pour chaque application générée on calcule la valeur de la fonction θ et on regroupe les applications qui ont la même valeur de θ. La fonction θ définie le rapport entre le nombre de processeurs et le nombre de périodes différentes et non multiples. D'un côté on sait que plus le nombre de processeur est grand plus le système a des chances d'être ordonnançable, d'un autre côté on sait aussi que plus le nombre de périodes non multiple est petit plus le système à des chances d'être ordonnançable. Dès lors l'idée de la fonction θ est d'évaluer les algorithmes suivant ces deux paramètres. On exécute donc successivement les trois algorithmes sur toutes les applications en notant à chaque fois si l'algorithme parvient à l'ordonnancer ou pas. Pour chaque groupe d'applications avec la même valeur de θ et pour chaque algorithme on calcule le taux de fiabilité qui est égal au rapport entre le nombre d'applications que l'algorithme parvient à ordonnancer et le nombre total d'applications dans ce groupe. Dans se cas l'algorithme exact nous sert d'arbitre entre les deux algorithmes approchés puisque c'est grâce à lui qu'on sait si une application est ordonnançable ou pas, d'ailleurs, c'est la raison pour laquelle son taux de fiabilité est maximal sur tous les groupes. Cette catégorie de tests nous permet de se rendre compte des points suivant :

- la disparité entre les deux algorithmes approchés (algorithme glouton et du type recherche locale) en termes de taux de fiabilité dans l'ordonnancement,
- l'impact de l'architecture, en particulier du nombre de processeurs, ainsi que la nature des périodes des tâches sur l'ordonnançabilité.

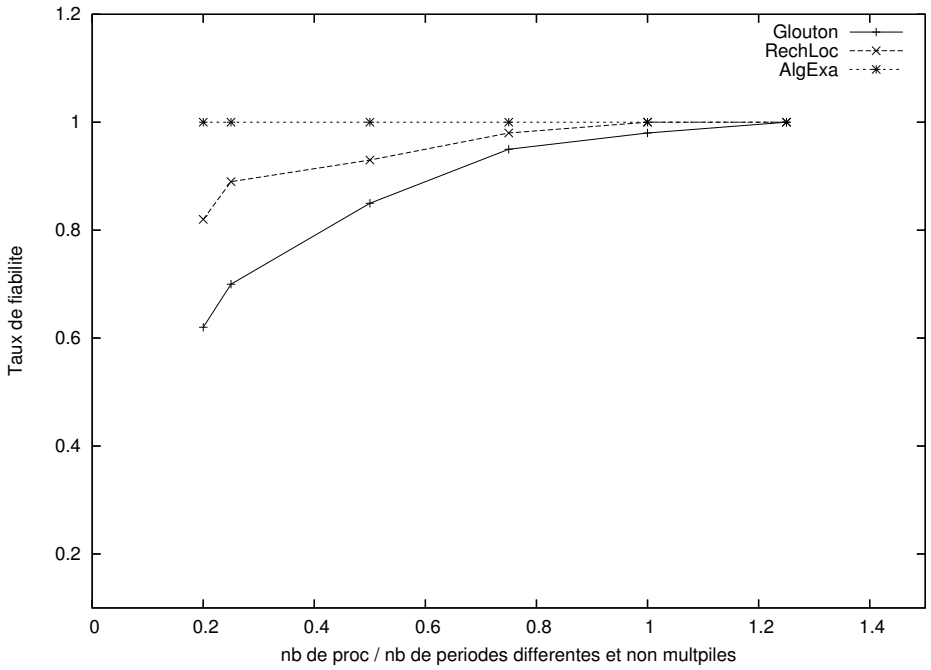

FIG. 2.19 – *Comparaison des taux de fiabilité des algorithmes*

Le diagramme de la figure 2.19 montre l'évolution du taux de fiabilité des algorithmes en fonction de θ. Mise à part l'algorithme exact, on remarque que l'algorithme du type recherche locale affiche des taux de fiabilités très intéressants compte tenu de son temps d'exécution (voir le test suivant). Par ailleurs on constate que l'algorithme glouton est efficace dès que $\theta \geq 1$ ce qui veut dire que son utilisation est relative à la nature de l'application.

2.4.2 Comparaison des temps d'exécution des algorithmes

Cette fois on exécute sur chaque système de tâches généré automatiquement les trois algorithmes et on note le temps d'exécution de chaque algorithme. Seules les applications ordonnançables avec les trois algorithmes ont été utilisées. Pour réaliser ce test on a disposé d'applications (générées automatiquement comme il est expliqué dans la section 2.4) comportant jusqu'à un millier de tâches pour une vingtaine de processeurs.

Chaque courbe, dans la figure 2.20 correspond à la variation du temps d'exécution en fonction des tailles des applications. Signalons que l'axe des ordonnées (temps d'exécution) suit une échelle logarithmique.

Sans surprise le temps d'exécution de l'algorithme exact explose rapidement (on ne l'a exécuté que sur des applications de tailles au-dessous de 100 tâches) alors que les algorithmes approchés suivent une progression plus graduelle. Cependant l'algorithme le plus rapide reste l'algorithme glouton qui affiche des temps d'exécution au-dessous de ceux de l'algorithme du type recherche locale.

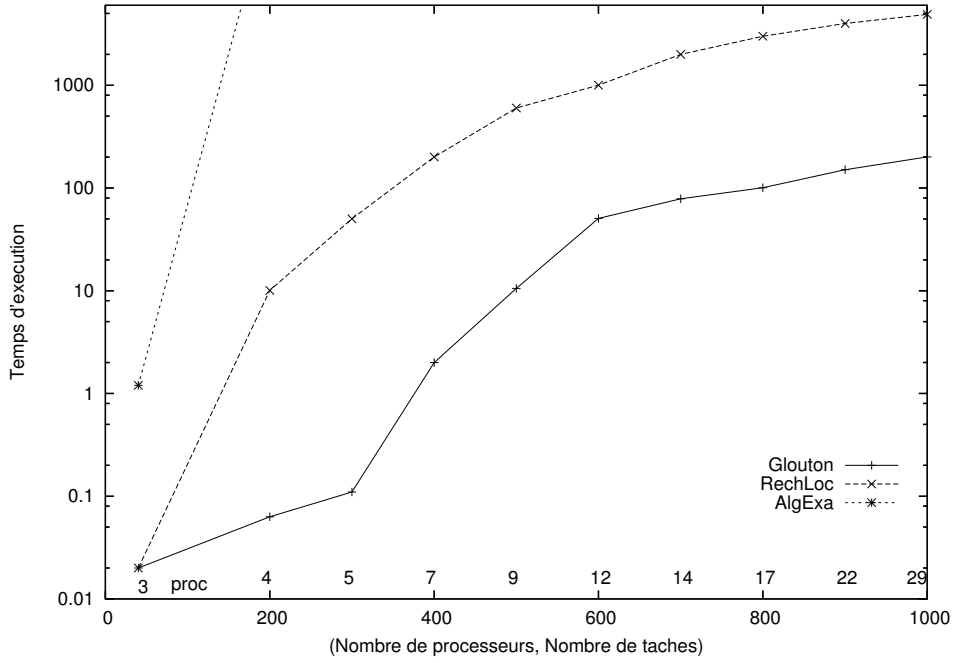

FIG. 2.20 – *Comparaison des temps d'exécution des algorithmes*

2.5 Équilibrage de charge et de mémoire

Cette section présente le travail qui a été nécessaire pour améliorer la minimisation du makespan (même si une première tentative dans ce sens est effectuée par l'heuristique d'ordonnancement proposée avant) ainsi que pour utiliser la mémoire de manière efficace (ce qui n'est pas pris en compte par l'heuristique d'ordonnancement). Dans cette setion on fait l'hypothèse que l'architecture est homogène, c'est-à-dire que tous les processeurs ont la même capacité mémoire et que les coûts de communication entre processeurs sont les mêmes. Une information supplémentaire $M(t_i)$ est associée à chaque tâche qui traduit la quantité de mémoire requise par la tâche t_i. Nous introduisons dans la section 2.5.2 la notion de bloc qui signifie le regroupement de plusieurs tâches qui vérifient certaines conditions.

2.5.1 Motivations

Deux motivations principales nous ont conduit à compléter l'heuristique avec un algorithme d'équilibrage de charge et de mémoire :

– dans la section 1.7.1 de l'état de l'art, il a été vu que l'équilibrage de charge conduit à minimiser le makespan et que minimiser le makespan conduit à minimiser la consommation d'énergie. Cette optimisation - même si elle existe dans l'heuristique d'ordonnancement - n'est appliquée que sur les tâches qui ont été assignées, chacune, à plusieurs processeurs.

Les autres tâches sont ordonnancés telles qu'elles ont été assignées et, ainsi, ne participent pas à la minimisation du makespan. D'un côté ceci entraîne la surcharge d'une partie des processeurs (toutes les instances des tâches qui sont ordonnançables sur le même processeur seront ordonnancées ensemble) et d'un autre côté, ceci entraîne la sous-charge d'une autre partie des processeurs (par exemple les instances d'une tâche avec un nombre premier comme valeur de période auront un processeur à elles toutes seules) ;

– à travers un petit exemple d'ordonnancement, de deux tâches dépendantes avec des périodes différentes, on observe l'importance de l'équilibrage de mémoire pour éviter aux processeurs de dépasser leurs capacités mémoire.

  Soient t_a et t_b deux tâches dépendantes ($T(t_b) = qT(t_a)$) ordonnancées sur deux processeurs différents. La tâche t_a qui est ordonnancée sur le processeur P_1 produit des données pour la tâche t_b qui est ordonnancée sur le processeur P_2 ce qui provoque une communication entre P_1 et P_2. Selon le mécanisme des transferts de données proposé, avant qu'elle ne soit exécutée, la tâche t_b doit recevoir les n données que la tâche t_a produit. L'ordonnancement de ces deux tâches est illustré dans la figure 2.21 ($q = 4$). Les quatre données produites par les exécutions des quatres instances de t_a doivent être stockées dans la mémoire de P_2 jusqu'à ce que la tâche t_b les utilise. Le problème réside dans le fait que la mémoire utilisée pour stocker la donnée produite par la première instance de la tâche t_a ne peut être réutilisée pour stocker la donnée produite par la deuxième, la troisième ou la quatrième instance de la tâche t_a. Ceci montre que la méthode de la réutilisation mémoire, par exemple, n'est pas efficace dans ce cas et qu'il convient de répartir les tâches qui requièrent le plus de mémoire sur des processeurs différents.

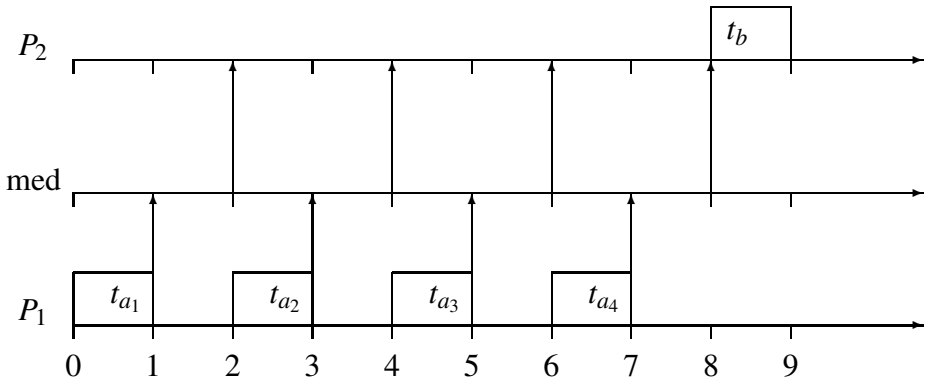

FIG. 2.21 – *Ordonnancement de deux tâches dépendantes, de périodes multiples, sur deux processeurs distincts*

Pour faire cela on a choisi d'exécuter un algorithme d'optimisation bi-critère (makespan,mémoire) sur le résultat obtenu après l'exécution de l'heuristique d'ordonnancement. Cet algorithme prend en entrée l'ordonnancement réalisé par l'heuristique (toutes les tâches ont été ordonnancées sur

les processeurs de l'architecture) et effectuera une nouvelle assignation des tâches sur les processeurs. Dès lors chaque tâche dispose de deux informations supplémentaires qu'elle ne possédait pas avant. En plus d'une période et d'une durée d'exécution elle a une date de début d'exécution et un processeur attribué (où elle a été ordonnancée). La nouvelle assignation est obtenue par des déplacements de tâches de leurs processeurs initiaux - qui sont définis par l'heuristique exécutée avant - vers d'autres processeurs. Elle devra bien évidemment respecter les contraintes de périodicité tout en réalisant un usage efficace de la mémoire et en minimisant le makespan. Pour s'assurer de l'exécution rapide de l'algorithme on a opté encore pour un algorithme glouton.

2.5.2 Algorithme d'optimisation bi-critère (makespan,mémoire)

L'algorithme commence par grouper les tâches en blocs conformément au principe suivant : un bloc se compose d'une ou de plusieurs tâches, dépendantes entre elles, ordonnancées sur le même processeur. En effet si on ordonnançait l'une de ces tâches sur un autre processeur cela produirait une communication inter-processeurs qui augmente le makespan. Un bloc peut, ou non, être déplacé de son processeur initial vers un autre processeur. Lorsqu'un bloc est déplacé, soit il garde la même date de début d'exécution (définie par l'heuristique), soit sa date de début d'exécution diminue, réduisant en même temps le makespan. La durée d'exécution (respectivement la quantité de mémoire requise) d'un bloc est la somme des durées d'exécution (respectivement des quantités de mémoire requises) des tâches qui le composent et sa date de début d'exécution est la date de début d'exécution de la première tâche (celle qui a la date de début d'exécution la plus petite entre les tâches qui composent ce bloc).

Soit bl_a un ensemble de n tâches $\{t_1,..,t_n\}$ ordonnancées sur le même processeur. On dit que bl_a est un bloc si :

$$\forall t_i \in bl_a \text{ alors } \exists t_j \in bl_a \text{ tel que } t_i \neq t_j \text{ et } t_i \prec t_j \text{ ou } t_j \prec t_i \tag{2.9}$$

Nous distinguons deux catégories de blocs :

1. les blocs qui sont composés exclusivement de premières instances des tâches. Cette catégorie représente les blocs dont la date de début d'exécution peut être réduite provoquant ainsi la minimisation du makespan ;
2. les blocs dont l'une des tâches le composant est une autre instance que la première instance. La date de début d'exécution d'un bloc de cette catégorie est réduite seulement si la date de début d'exécution du bloc où se trouve la première instance de la première tâche du bloc en question est réduite.

On note par $\mathbb{A}_{P_1 \to P_2}(bl_a)$ l'amélioration en termes de temps résultant du déplacement du bloc de première catégorie bl_a du processeur P_1 vers le processeur P_2. $S(bl_a)_{P_1}$ est sa date de début d'exécution initiale sur le processeur P_1 et $S(bl_a)_{P_2}$ est sa nouvelle date de début d'exécution sur le processeur P_2, $S(bl_a)_{P_2}$ est inférieure ou égale à $S(bl_a)_{P_1}$.

$$\mathbb{A}_{P_1 \to P_2}(bl_a) = S(bl_a)_{P_1} - S(bl_a)_{P_2} \tag{2.10}$$

Pour que la contrainte de périodicité demeure respectée on a introduit une condition qui consiste à vérifier, avant de déplacer un bloc vers un processeur, que les blocs qui ont été déplacés vers ce processeur peuvent s'exécuter suivant l'hyper-période (on a vu dans la section 2.2.2 que toutes les tâches, et par conséquent les blocs, ont la même période qui est l'hyper-période). Pour le vérifier il suffit de s'assurer que le bloc qu'on cherche à déplacer est ordonnancé avant la date à laquelle le premier bloc à être déplacé vers ce processeur, va se répéter suivant l'hyper-période. Le reste des blocs qui ont été déplacés vers ce processeur se répéterons, de toute manière, après la répétition du premier bloc. Si bl_a est le premier bloc à être déplacé vers le processeur P_i et hp est l'hyper-période, alors le bloc bl_b satisfait la contrainte de périodicité des blocs qui ont été déplacés vers P_i auparavant si :

$$S(bl_b)_{P_i} + C(bl_b) \leq S(bl_a)_{P_i} + hp \tag{2.11}$$

L'algorithme que nous proposons est basé sur une fonction de coût λ pour chaque bloc en fonction de son processeur initial et du processeur destination vers lequel il sera déplacé. Cette fonction combine l'amélioration $\mathbb{A}$ qui résulte de ce déplacement ainsi que la somme des quantités mémoire requises par les blocs déjà déplacés vers le processeur destination. Si bl_a est le bloc candidat au déplacement, $M(bl_a)$ est la somme des quantités mémoire des tâches appartenant à bl_a, P_1 et P_2 sont respectivement le processeur initial et destination et bl_1,...,bl_k sont les k blocs qui ont déjà été déplacés vers P_2 alors :

$$\lambda_{P_1 \to P_2}(bl_a) = \begin{cases} \mathbb{A}_{P_1 \to P_2}(bl_a) & \text{si aucun bloc n'a été déplacé vers } P_2 \ (k = 0) \\ \frac{\mathbb{A}_{P_1 \to P_2}(bl_a)+1}{\sum_{j=1}^{k} M(bl_j)} & \text{sinon} \end{cases} \tag{2.12}$$

Comme $\mathbb{A}_{P_1 \to P_2}(bl_a)$ peut être nul, on lui additionne la valeur 1. Si le coût est maximal alors les valeurs de la paire $(\mathbb{A}_{P_1 \to P_2}(bl_a), \frac{1}{\sum_{j=1}^{k} M(bl_j)})$ doivent être les plus larges possibles. On en déduit que la valeur de $\sum_{j=1}^{k} M(bl_j)$ doit être la plus petite possible. Maximiser $\mathbb{A}_{P_1 \to P_2}(bl_a)$ conduit à choisir le processeur qui entraîne une amélioration maximale pour le makespan alors que minimiser $\sum_{j=1}^{k} M(bl_j)$ conduit à choisir, en même temps, le processeur qui dispose, le plus, de réserve mémoire.

Après avoir construit les blocs sur chaque processeur, les blocs sont triés suivant un ordre croissant de dates de début d'exécution. L'algorithme proposé calcule la fonction de coût du bloc candidat au déplacement vers chaque processeur. Outre la condition 2.11 la date de fin d'exécution du dernier bloc, déplacé vers le processeur choisi, doit être inférieure ou égale à la date de début d'exécution du bloc candidat. Le processeur choisi sera celui qui maximise la fonction de coût. Si le bloc appartient à la première catégorie et si $\mathbb{A} > 0$ alors la date de début d'exécution de ce bloc est réduite et le makespan est, ainsi, amélioré. Dans ce cas l'algorithme met à jour les dates de début d'exécution des blocs restants qui contiennent les autres instances de la tâche dont la première instance se trouve dans le bloc qui vient d'être déplacé. L'algorithme 6 détaille l'équilibrage de charge et de mémoire.

Algorithme 6 L'algorithme d'équilibrage de charge et de mémoire

1: **pour** Chaque processeur **faire**
2: Construire les blocs sur ce processeur
3: **fin pour**
4: Trier les blocs suivant un ordre croissant des dates de début d'exécution
5: **pour** Chaque bloc bl_a initialement ordonnancé sur P_x **faire**
6: **pour** Chaque processeur P_y **faire**
7: Calculer la fonction de coût $\lambda_{P_x \to P_y}(bl_a)$ pour chaque processeur P_y qui vérifie la condition 2.11 et dont la date de fin d'exécution du dernier bloc déplacé vers celui-ci est inférieure ou égale à la date de début d'exécution de bl_a
8: **fin pour**
9: Rechercher le processeur P qui maximise $\lambda_{P_x \to P}(bl_a)$
10: **si** $\mathbb{A}_{P_x \to P}(bl_a) > 0$ sur P **alors**
11: Mettre à jour les dates de début d'exécution des blocs restant qui contiennent des instance dont la première est dans bl_a
12: **fin si**
13: Déplacer bl_a vers P
14: **fin pour**

EXEMPLE 2.11

Dans le but d'illustrer l'évolution de l'algorithme 6 nous appliquons tout d'abord l'heuristique d'ordonnancement proposée dans la section 2.2 au système de la figure 2.22. Ce système est composé d'un graphe d'algorithme et d'un graphe d'architecture. Le graphe d'algorithme est composé de cinq tâches définies comme suit : $(t_a : 3,1)$, $(t_b : 6,1)$, $(t_c : 6,1)$, $(t_d : 12,1)$ *et* $(t_e : 12,1)$. *Les quantités de mémoire requises par chaque tâche sont :* $M(t_a) = 4$, $M(t_b) = 1$, $M(t_c) = 1$, $M(t_d) = 2$ *et* $M(t_e) = 2$. *Le graphe d'architecture est composé de trois processeurs* P_1, P_2 *et* P_3 *connectés par le médium de communication "med" (le coût d'une communication entre chaque paire de processeurs est égal à 1).*

Le résultat de l'ordonnancement est décrit par la figure 2.23. On constate que le makespan est égal à 15, *la somme des quantités de mémoire utilisée sur le processeur* P_1 *est égale à* 16, *sur le processeur* P_2 *est égale à* 4 *et sur le processeur* P_3 *est égale à* 4.

Le résultat de la construction des blocs est le suivant :

- *chaque instance de la tâche* t_a *constitue un bloc,*
- *les instances des tâches* t_b *et* t_c *forment les deux blocs* $[t_{b1} - t_{c1}]$ *et* $[t_{b2} - t_{c2}]$,
- *les tâches* t_d *et* t_e *forment le bloc* $[t_b - t_e]$;

Ensuite, suivant un ordre croissant des dates de début d'exécution données par l'ordonnance-

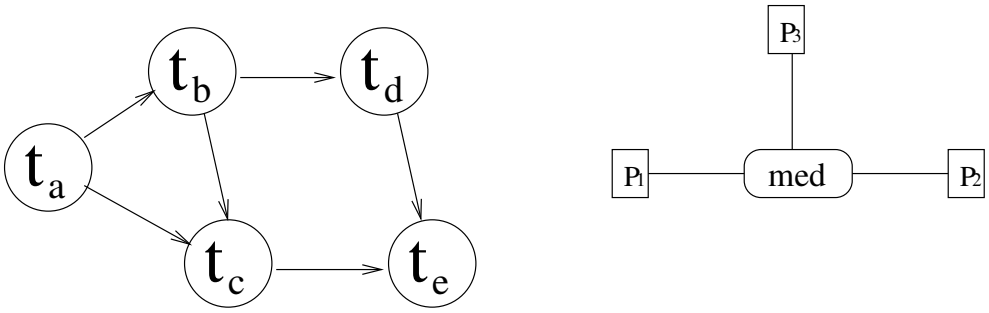

FIG. 2.22 – *(Exemple 2.11) Système temps réel*

ment de la figure 2.23, les blocs sont déplacés de la manière suivante :

1. *le bloc* $[t_{a1}]$ *est sélectionné,*
   - $\mathbb{A}_{P_1 \to P_1,P_2,P_3}([t_{a1}]) = 0$,
   - $\sum_{t_i \in p_2} M(t_i) = \sum_{t_i \in p_3} M(t_i) = \sum_{t_i \in p_1}(t_i) = 0$,
   - $\lambda_{P_1 \to P_1,P_2,P_3}([t_{a1}]) = 0$,
   - *on choisit de garder le bloc* $[t_{a1}]$ *sur le processeur* P_1.
2. *le bloc* $[t_{a2}]$ *est sélectionné,*
   - $\mathbb{A}_{P_1 \to P_1,P_2,P_3}([t_{a2}]) = 0$,
   - $\sum_{t_i \in p_2} M(t_i) = \sum_{t_i \in p_3} M(t_i) = 0$ *et* $\sum_{t_i \in p_1}(t_i) = 4$,
   - $\lambda_{P_1 \to P_1}([t_{a2}]) = 1/4$, $\lambda_{P_1 \to P_2,P_3}([t_{a2}]) = 1$,
   - *alors le bloc* $[t_{a2}]$ *est déplacé vers* P_2 *ou* P_3 *car* $\lambda_{P_1 \to P_2,P_3}([t_{a2}]) = 1 > \lambda_{P_1 \to P_1}([t_{a1}]) = 1/4$ *(*P_2 *est choisi).*
3. *le bloc* $[t_{b1} - t_{c1}]$ *est sélectionné,*
   - $\mathbb{A}_{P_2 \to P_1,P_3}([t_{b1} - t_{c1}]) = 0$ *et* $\mathbb{A}_{P_2 \to P_2}([t_{b1} - t_{c1}]) = 0$,
   - $\sum_{t_i \in p_1} M(t_i) = \sum_{t_i \in p_2} M(t_i) = 4$ *et* $\sum_{t_i \in p_3}(t_i) = 0$,
   - $\lambda_{P_2 \to P_3}([t_{b1} - t_{c1}]) = 0$, $\lambda_{P_2 \to P_1}([t_{b1} - t_{c1}]) = 1/4$ *et* $\lambda_{P_2 \to P_2}([t_{b1} - t_{c1}]) = 1/2$,
   - *le bloc* $[t_{b1} - t_{c1}]$ *est déplacé vers* P_2,
   - *comme* $[t_{b1} - t_{c1}]$ *est un bloc de première catégorie et* $\lambda > 0$ *alors une mise à jour de la date de début d'exécution du bloc* $[t_{b2} - t_{c2}]$ *est nécessaire. La nouvelle date de début d'exécution de* $[t_{b2} - t_{c2}]$ *est* $S([t_{b2} - t_{c2}]) = 10$
4. *le bloc* $[t_{a3}]$ *est sélectionné,*
   - $\lambda_{P_1 \to P_1}([t_{a3}]) = 1/4$, $\lambda_{P_1 \to P_2}([t_{a3}]) = 1/6$ *et* $\lambda_{P_1 \to P_3}([t_{a3}]) = 1$,
   - *alors le bloc* $[t_{a3}]$ *est déplacé vers* P_3.
5. *le bloc* $[t_{a4}]$ *est sélectionné,*
   - $\lambda_{P_1 \to P_1}([t_{a4}]) = 1/4$, $\lambda_{P_1 \to P_2}([t_{a4}]) = 1/6$ *et* $\lambda_{P_1 \to P_3}([t_{a4}]) = 1/4$,
   - *alors le bloc* $[t_{a4}]$ *est déplacé vers* P_1.

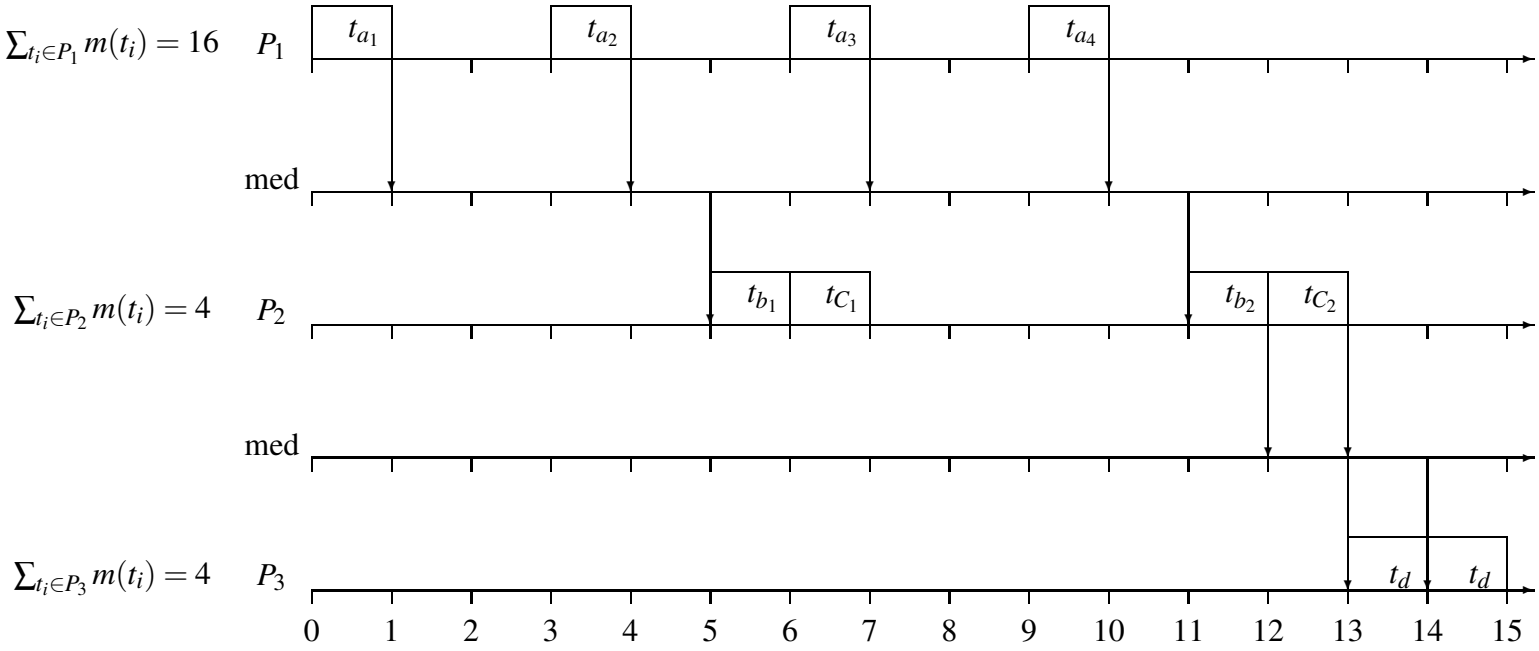

FIG. 2.23 – *(Exemple 2.11) Ordonnancement avant l'exécution de l'algorithme d'équilibrage de charges et d'usage efficace de mémoire*

6. *le bloc* $[t_{b2} - t_{c2}]$ *est sélectionné,*
- $\lambda_{P_2 \to P_1}([t_{b2} - t_{c2}]) = 1/8$, $\lambda_{P_2 \to P_2}([t_{b2} - t_{c2}]) = 0$ *et* $\lambda_{P_2 \to P_3}([t_{b2} - t_{c2}]) = 0$,
- *le bloc* $[t_{b2} - t_{c2}]$ *est déplacé vers* P_1.

7. *le bloc* $[t_b - t_e]$ *est sélectionné,*
- $\lambda_{P_3 \to P_1}([t_b - t_e]) = 1/10$ *mais il ne satisfait pas la condition 2.11 donc cette possibilité est abandonnée,* $\lambda_{P_3 \to P_2}([t_b - t_e]) = 1/4$ *et* $\lambda_{P_3 \to P_3}([t_b - t_e]) = 1/4$,
- *alors le bloc* $[t_b - t_e]$ *est déplacé vers* P_3.

Le résultat de l'algorithme d'équilibrage de charge et de mémoire (Algorithme 6) est décrit par la figure 2.24. On constate que le makespan a été réduit d'une unité (sa valeur passe de 15 *à* 14*) et que l'utilisation de la mémoire est plus équilibrée.*

2.5.2.1 Complexité

Soit n_{blocs} le nombre de blocs construits à partir des n tâches de départ ($n_{blocs} \leq n$). La complexité de l'algorithme d'équilibrage de charge et de mémoire est de $O(mn_{blocs})$.

Comme le nombre de capteurs est petit relativement au nombre de tâches (nous rappelons que se sont les capteurs qui imposent leurs périodes aux tâches) alors le nombre de périodes différentes est petit. De plus, comme les tâches dépendantes sont soit à la même période soit à des périodes multiples, elles sont ordonnancées sur le même processeur ce qui permet d'obtenir des blocs avec un nombre assez grand de tâches et on se retrouve, en définitive, avec un nombre restreint de blocs.

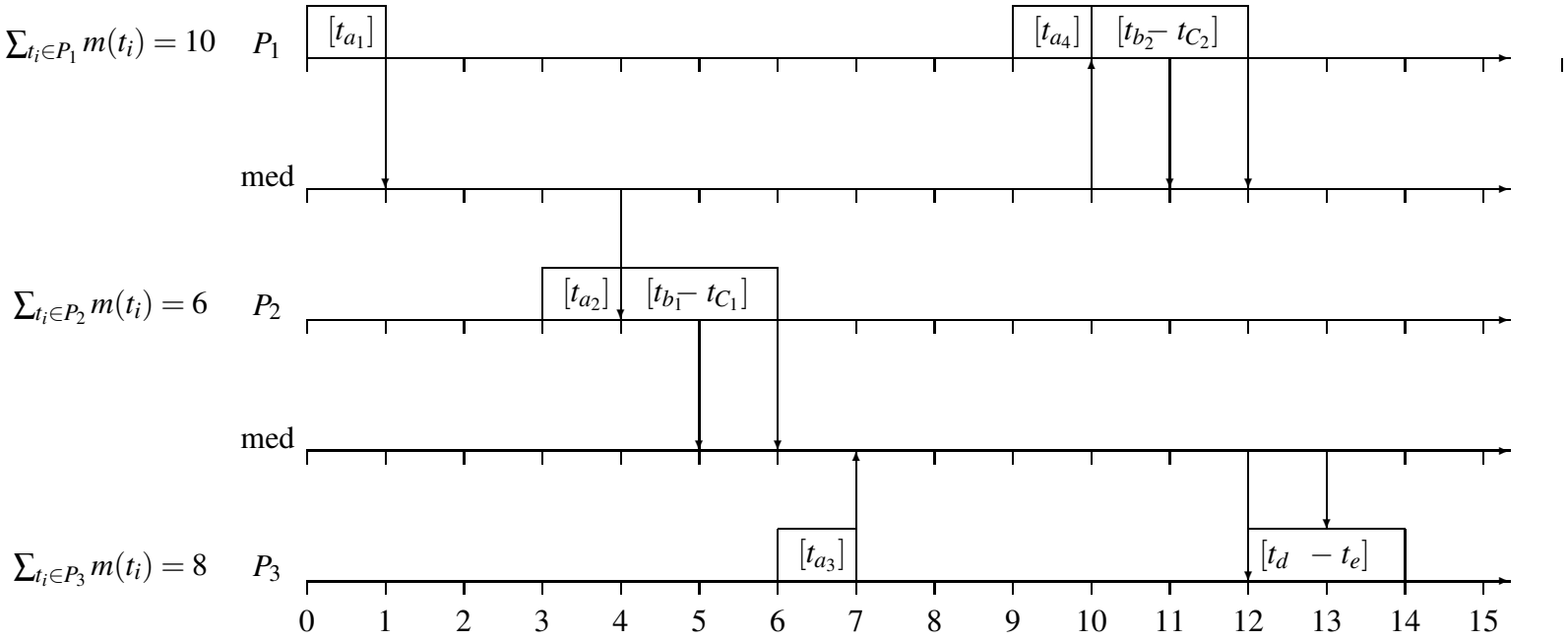

FIG. 2.24 – *(Exemple 2.11) Ordonnancement après l'exécution de l'algorithme d'équilibrage de charges et d'usage efficace de mémoire*

La complexité polynomiale de l'algorithme ainsi que le nombre faible de blocs conduisent à un temps d'exécution très rapide.

2.5.2.2 Étude théorique de performances

Dans le but d'évaluer l'algorithme d'équilibrage de charge et de mémoire nous avons calculé la borne inférieure et la borne supérieure de l'amélioration apportée au makespan, et on a évalué l'α-approximation de l'algorithme en ce qui concerne l'équilibrage de mémoire.

Bornes de l'amélioration du makespan

Si $L_{initial}$ est la longueur du makespan initial (avant d'exécuter l'algorithme d'équilibrage de charge et de mémoire) et $L_{nouveau}$ la longueur du makespan après l'exécution de l'algorithme alors l'amélioration du makespan est définie par :

$$\mathbb{A}_{total} = L_{initial} - L_{nouveau}$$

Supposons que dans le graphe d'architecture chaque couple de processeurs soit connecté par un médium de communication. Le théorème suivant introduit la borne supérieure et la borne inférieure qui délimitent $\mathbb{A}_{total}$. Ces bornes nous permettent, d'un côté de s'assurer que $L_{nouveau}$ est toujours

inférieure ou égale à $L_{initial}$ et, d'un autre côté, de savoir jusqu'à quel point l'algorithme 6 peut améliorer le makespan.

THÉORÈME 2.11
La valeur de $\mathbb{A}_{total}$ est bornée par :

$$0 \leq \mathbb{A}_{total} \leq com \times (M-1)! \tag{2.13}$$

com est le coût d'une communication entre deux processeurs et M est le nombre de processeurs de l'architecture.

Preuve
Commençons par prouver $\mathbb{A}_{total} \leq com(M-1)!$. Lorsqu'un bloc bl_a est déplacé d'un processeur P_1 vers un processeur P_2 la communication avec un coût égal à *com*, qui existait entre le bloc bl_a et un ou plusieurs blocs ordonnancés sur P_2, est supprimée. Il est important de noter que même si il y a d'autres déplacements de blocs, de P_1 vers P_2 ou de P_2 vers P_1, la valeur de $\mathbb{A}_{total}$ n'est pas affectée, c'est-à-dire que le makespan ne diminue pas de la somme de tous les coûts de communication correspondantes. Ceci est dû au fait que les blocs sont ordonnancés séquentiellement sur un processeur. Si la date de début d'exécution d'un bloc diminue, alors les dates de début d'exécution de tous les blocs qui sont ordonnancés après celui-ci diminuent à leur tour. De cette manière le makespan diminue au plus de *com* $\times$ (nombre de paires de processeurs). Le nombre de paires de processeurs distincts est de $(M-1)!$.

Maintenant nous allons prouver que $0 \leq \mathbb{A}_{total}$. L'algorithme proposé cherche à déplacer chaque bloc de telle façon que sa date de début d'exécution diminue, ou au pire qu'il garde sa date de début d'exécution initiale. Ainsi les blocs qui communiquent avec le bloc déplacé peuvent diminuer leurs dates de début d'exécution ou bien gardent leurs dates de début d'exécution initiales. Ceci implique que $\mathbb{A}_{total}$ est supérieure ou égale à 0. Pour résumer, ce théorème montre que dans certains cas toutes les communications peuvent être supprimées et l'amélioration est égale à $com(M-1)!$ et que dans d'autres cas $\mathbb{A}_{total}$ est nulle□

α-approximation de l'équilibrage de mémoire

Maintenant on ne considère que l'équilibrage de mémoire et on suppose que l'amélioration $\mathbb{A}$ est une constante C. Dans ce cas la fonction de coût s'écrit : $\lambda(bl_a) = \frac{C}{\sum_{i=1}^{k} M(bl_i)}$. L'algorithme recherche le processeur qui maximise λ c'est-à-dire celui qui minimise $\frac{C}{\sum_{j=1}^{k} M(bl_j)}$ (k est le nombre de blocs se trouvant sur le processeur vers lequel le bloc bl_a sera déplacé).

Le théorème suivant donne une idée des performances de l'algorithme en ce qui concerne l'équilibrage de mémoire. On rappelle que la α-approximation a été introduite dans la section 1.6.3.3.

THÉORÈME 2.12
Si m est le nombre de processeurs dans l'architecture alors l'algorithme proposé est $(2-\frac{1}{m})$-approché. En d'autres termes si ω_{opt} est la solution optimale et ω est la solution obtenue par l'algorithme proposé (une solution est la quantité de mémoire maximale utilisée par un processeur

parmi toutes les quantités utilisées par les autres processeurs et l'objectif est qu'elle soit la plus petite possible) alors :

$$\frac{\omega}{\omega_{opt}} \leq 2 - \frac{1}{m} \tag{2.14}$$

Preuve

On considère qu'un bloc bl_a est déplacé vers un processeur P_q et que la quantité mémoire que ce bloc requiert est $M(bl_a)$. On note par V la quantité de mémoire requise par les blocs qui ont déjà été déplacés vers P_q. Selon les critères de choix de l'algorithme 6, pour que P_q soit choisi, il faut que V soit la quantité mémoire minimale relativement à toutes les quantités mémoire requises par les blocs déjà déplacés vers les processeurs.

Si ω_{opt} est la solution optimale alors $M(bl_a)$ qui ne représente qu'une partie de la solution est :

$$M(bl_a) \leq \omega_{opt} \tag{2.15}$$

Si N_{blocs} est le nombre de blocs construits par l'algorithme alors :

$$\frac{\sum_{j=1}^{N_{blocs}} M(bl_j)}{N_{blocs}} \leq \omega_{opt} \tag{2.16}$$

D'après la définition de V :

$$V + \frac{M(bl_a)}{m} \leq \frac{\sum_{j=1}^{N_{blocs}} M(bl_j)}{N_{blocs}} \tag{2.17}$$

Des équations 2.16 et 2.17 on a :

$$V + \frac{M(bl_a)}{m} \leq \omega_{opt} \tag{2.18}$$

Donc, la quantité mémoire requise par tous les blocs déplacés vers P_i, après le déplacement de bl_a, est égale à :

$$\omega = V + M(bl_a)$$

En réécrivant ω on a :

$$\omega = V + \frac{M(bl_a)}{m} + M(bl_a)\frac{m-1}{m} \tag{2.19}$$

Des équations 2.15 et 2.18 on a :

$$V + \frac{M(bl_a)}{m} + M(bl_a)\frac{m-1}{m} \leq (1 + \frac{m-1}{m})\omega_{opt}$$

et de l'équation 2.19 on a :

$$\omega \leq (1 + \frac{m-1}{m})\omega_{opt}$$

Par conséquent

$$\frac{\omega}{\omega_{opt}} \leq 2 - \frac{1}{m}$$

ce qui prouve le théorème 2.12□

Dans ce chapitre nous avons abordé le problème d'ordonnancement temps réel multiprocesseur avec contraintes de précédence et de périodicité stricte. Dans le chapitre suivant nous aborderons un autre problème d'ordonnancement avec cette fois la contrainte de latence.

Chapitre 3

Ordonnancement temps réel multiprocesseur avec contraintes de précédence et de latence

Dans ce chapitre on s'intéresse à des systèmes de tâches avec des contraintes de précédence et de latence. Contrairement à la périodicité, la latence est une contrainte peu étudiée et les références sur ce sujet sont moins abondantes. On a commencé par s'intéresser aux travaux réalisés par Cucu [58] qui a défini la latence et a proposé une condition d'ordonnançabilité dans le cas d'un seul processeur. Nous complétons ici cette étude en explorant tous les cas possibles d'ordonnancement. L'étude que nous proposons commence par le cas monoprocesssseur et une seule contrainte de latence et se termine avec le cas multiprocesseur et plusieurs contraintes de latence. On propose également une étude de complexité nous permettant de classer ce problème selon la théorie de la NP-complétude. En conclusion de cette étude nous proposons un algorithme glouton d'ordonnancement après avoir proposé quelques algorithmes intermédiaires.

Ce chapitre est organisé comme suit : dans la section 3.1 on définit la contrainte de latence puis, dans la section 3.2, on présente le modèle qu'on a adopté. La section 3.3 expose l'étude d'ordonnançabilité dans le cas d'une seule contrainte de latence et propose un algorithme glouton d'ordonnancement dans ce cas. Ensuite, dans la section 3.4, on s'intéresse au cas plus général de plusieurs contraintes de latence afin de proposer un algorithme glouton d'ordonnancement. Pour finir la section 3.5 propose le calcul de complexité des algorithmes proposés.

3.1 Définitions

Une contrainte de latence est définie sur deux tâches liées par un chemin dans le graphe. Imposer une contrainte de latence $L(t_a,t_b)$ entre deux tâches t_a et t_b, consiste à s'assurer que le temps d'ordonnancement ou le temps qui s'écoule entre la date de début d'exécution de la tâche t_a et la date de fin d'exécution de la tâche t_b est inférieur ou égal à $L(t_a,t_b)$. Si $S(t_a)$ et $C(t_b)$ sont, respectivement, la date de début d'exécution de la tâche t_a et la date de fin d'exécution de la tâche t_b alors l'ordonnancement de ces deux tâches doit satisfaire la condition suivante :

$$C(t_b) - S(t_a) \leq L(t_a,t_b) \tag{3.1}$$

Dans ce manuscrit la latence est définie exclusivement entre deux tâches reliées dans le graphe des tâches, i.e., il existe au moins un chemin entre t_a et t_b dans le graphe des tâches (voir remarque 3.1).

Lors de la définition d'une contrainte de latence entre deux tâches dépendantes t_a et t_b, les données produites par la tâche t_a doivent être consommées par la tâche t_b en un temps inférieur ou égal à $L(t_a,t_b)$ (on inclut dans ce temps les durées d'exécution de t_a et t_b).

REMARQUE 3.1
On ne traite pas le cas d'une contrainte définie entre deux tâches indépendantes, i.e., aucun chemin les relie. D'après notre modèle, en imposant une contrainte de latence à deux tâches, celles-ci deviennent reliées par une contrainte, donc dépendantes.

La contrainte de latence, tout comme celle de périodicité, fait du problème d'ordonnancement un problème de décision où la condition 3.1 n'est pas toujours satisfaite (aucun ordonnancement qui satisfait cette contrainte n'existe). Par ailleurs la différence entre le respect de la latence et la minimisation du makespan est que cette dernière consiste à optimiser au mieux sans avoir une valeur objectif précise (une borne supérieure à ne pas dépasser) alors que la latence obéit à une valeur bien précise qu'il faut absolument satisfaire au risque de conséquences désastreuses (voir la section 1.2.2.6).

3.2 Modèle

Dans ce modèle d'algorithme qui est toujours représenté par un graphe de flot de données il n'y a pas de contrainte de périodicité. En revanche une ou plusieurs contraintes de latence peuvent être imposées entre des paires de tâches. Il nous faut introduire quelques notions nécessaires à la compréhension de la suite.

On note par $\mathcal{V}(t_a,t_b)$ l'ensemble des tâches appartenant à tous les chemins de t_a à t_b dans le graphe y compris les tâches t_a et t_b. En imposant une contrainte de latence $L(t_a,t_b)$ toutes les tâches appartenant à l'ensemble $\mathcal{V}(t_a,t_b)$ sont dites concernées par la latence $L(t_a,t_b)$. Les tâches t_a et t_b sont, respectivement, la source et la destination de la contrainte de latence $L(t_a,t_b)$. On appelle le temps d'ordonnancement entre la tâche source et la tâche destination le temps global d'ordonnancement.

Étant donné que la latence est définie entre deux tâches reliées soit par un seul chemin, soit par plusieurs alors, parmi tous ces chemins, on note par $\mathcal{C}h_{pl}(t_a,t_b)$ l'ensemble des tâches appartenant au chemin le plus long en termes de somme des durées d'exécution des tâches qu'il contient.

Dans un même graphe on peut donc définir plusieurs contraintes de latences qui peuvent être reliées les unes aux autres. Dans un premier temps on étudie le cas d'une seule latence où la difficulté sera d'ordonnancer les tâches appartenant à $\mathcal{V}(t_a,t_b)$ de telle manière que le temps global d'ordonnancement n'excède pas $L(t_a,t_b)$.

L'architecture est décrite de la même façon que celle du chapitre précédent. On suppose que le graphe d'architecture est un graphe complet et que les coûts de communication sont les mêmes entre chaque paire de processeurs.

3.3 Étude du cas d'une seule contrainte de latence

3.3.1 Cas monoprocesseur

Dans le cas monoprocesseur la condition d'ordonnançabilité est simple à trouver ainsi qu'à vérifier puisque il suffit que le temps d'ordonnancement de toutes tâches soit inférieur à la valeur de la latence [58]. Si une contrainte de latence $L(t_a,t_b)$ est définie entre les deux tâches t_a et t_b alors la condition nécessaire et suffisante d'ordonnançabilité est la suivante :

$$\sum_{t_i \in \mathcal{V}(t_a,t_b)} C(t_i) \leq L(t_a,t_b) \tag{3.2}$$

3.3.2 Cas multiprocesseur

Le fait d'ordonnancer deux tâches dépendantes sur deux processeurs distincts provoque une communication entre ces deux processeurs qui, par conséquent, rallonge le temps d'ordonnancement entre ces deux tâches. La première idée consiste à ordonnancer toutes les tâches concernées par la latence sur le même processeur. De cette façon aucune communication n'est provoquée, cependant la valeur de la latence doit être suffisante pour satisfaire la condition 3.2. D'un autre côté, comme le montre l'exemple 3.1, le fait d'ordonnancer ces tâches sur plusieurs processeurs ne signifie pas, nécessairement, que le temps d'ordonnancement obtenu est plus large que celui qu'on obtient en monoprocesseur.

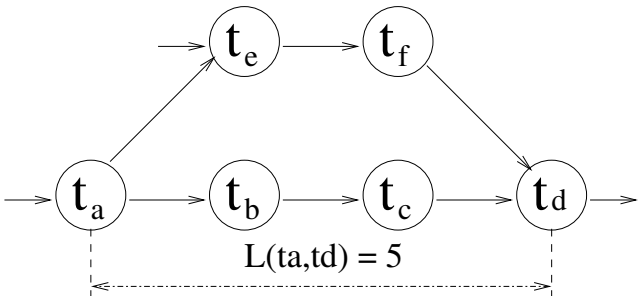

FIG. 3.1 – *(Exemple 3.1) Partie du graphe d'algorithme soumise à la contrainte de latence L*

EXEMPLE 3.1
Sur la figure 3.1 on ne laisse apparaître que la partie du graphe concernée par la contrainte de latence $L(t_a,t_d)$. La durée d'exécution est la même pour toutes les tâches et elle est égale à 2 et le coût d'une communication entre les processeurs est égal à 1. La figure 3.2 expose le résultat

de l'ordonnancement du sous-graphe de la figure 3.1 sur un processeur alors que la figure 3.3 fait de même dans le cas d'une architecture à deux processeurs. Comme la valeur de la contrainte latence $L(t_a,t_d)$ est égale à 5 alors l'architecture monoprocesseur ne satisfait pas cette contrainte ($\sum_{t_i \in \{t_a,t_b,t_c,t_d,t_e\}} C(t_i) \leq 5$). Par ailleurs une architecture avec deux processeurs satisfait parfaitement la contrainte bien qu'il y ait, en plus, un coût de deux fois une communication. On en déduit que, comme pour le problème d'ordonnançabilité du chapitre précédent (périodicité et précédence), le nombre de processeurs est une donnée qui influe sur l'ordonnançabilité.

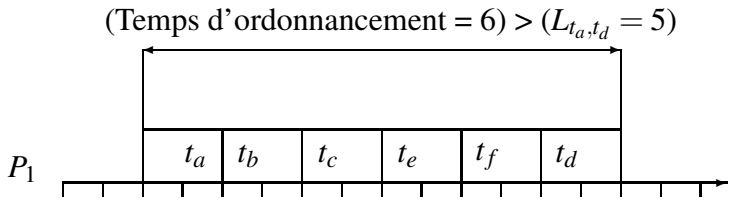

FIG. 3.2 – *(Exemple 3.1) Ordonnancement monoprocesseur*

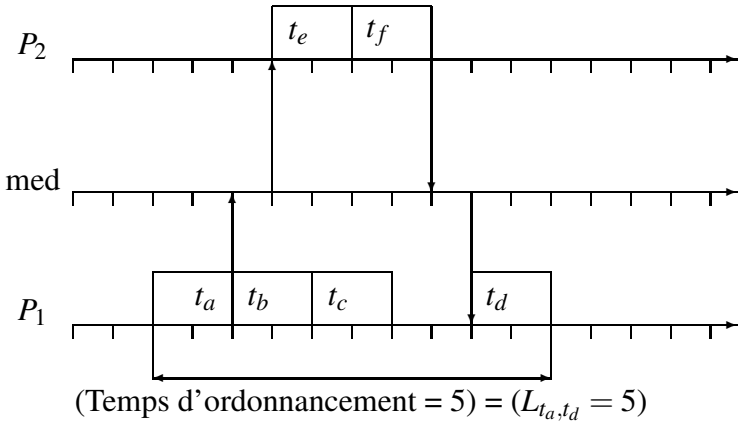

FIG. 3.3 – *(Exemple 3.1) Ordonnancement sur une architecture composée de deux processeurs*

Maintenant il nous faut définir, comme pour le cas monoprocesseur, une condition nécessaire et suffisante, si elle existe, dans le cas multiprocesseur qui prendrait en compte la valeur de latence, les durées d'exécution des tâches concernées par cette latence, le nombre de processeurs et les coûts de communication.

Avant d'entamer cette étude il est important d'évaluer la difficulté du problème posé (latence et précédence). Contrairement au problème d'ordonnancement multiprocesseur avec contraintes de précédence et de périodicité qui a été largement étudié et prouvé NP-difficile ([150] ou dans [151]), la NP-complétude du problème avec contrainte de précédence et de latence a été peu étudiée. Nous donnons dans la section suivante une preuve plus détaillée que celle donnée dans [58].

3.3.2.1 Étude de complexité

Dans le cas d'une seule latence définie entre deux tâches t_a et t_b, notre problème consiste à ordonnancer, sur plusieurs processeurs, les tâches appartenant à l'ensemble $\mathcal{V}(t_a,t_b)$ tout en respectant les dépendances entre ces tâches, et en prenant en compte le coût des communications.

L'évaluation de la difficulté se fait en deux étapes : il faut, premièrement, vérifier que notre problème appartient à la classe NP et, deuxièmement, montrer qu'il est NP-difficile.

Nous allons prouver la NP-complétude d'un cas particulier de notre problème dans lequel les précédences ne sont pas prises en compte. Si ce cas particulier est NP-difficile, alors notre problème l'est aussi. Le problème qu'on considère est donc :

Instance de notre problème:

- un ensemble de tâches $\mathcal{G}$,
- une fonction (durée d'exécution) $E : \mathcal{G} \rightarrow \mathbb{N}$,
- un entier (nombre de processeurs) m,
- un entier (la valeur de latence) L.

Question :
Existe-t-il un ordonnancement $\mathcal{E}$ des tâches de l'ensemble $\mathcal{G}$ sur les m processeurs de telle manière que sur chaque processeur le temps qui s'écoule entre la date de début d'exécution de la première tâche ordonnancée et la date de fin d'exécution de la dernière tâche ordonnancée est inférieure ou égale à L ?

Appartenance à NP ?

Premièrement, pour prouver que ce problème appartient à la classe NP, nous devons prouver qu'il existe un algorithme polynomial qui, étant donné une solution au problème, peut vérifier que cette solution est bonne (la contrainte de latence est respectée). Pour cela, il suffit de calculer la somme des durées d'exécution des tâches sur chaque processeur et de les comparer avec la valeur de L, ce qui représente une complexité de $O(n)$ où n est le nombre de tâches. L'algorithme qui réalise cette vérification est polynomial. Donc ce problème est dans la classe des problèmes NP.

Problème NP-difficile ?

Pour y arriver il suffit de faire un réduction (transformation) avec un problème qui est déjà connu comme NP-difficile. En optant pour la réduction à partir du problème Partition, on obtient :

Définition du problème Partition :
Instance du problème Partition :

- un ensemble $\mathcal{S} = \{s_1,s_2,...,s_n\}$,
- une fonction de poids $Q : \mathcal{S} \rightarrow \mathbb{N}$.

Question :
Existe-t-il une partition de $\mathcal{S}$ en 2 sous-ensembles $\mathcal{S}_1$ et $\mathcal{S}_2$ telle que :

$$\sum_{s \in \mathcal{S}_1} Q(s) = \sum_{s \in \mathcal{S}_2} Q(s)$$

On définit la réduction $\mathcal{F}(\mathcal{S},Q)$

$$\begin{cases} \mathcal{G} = \mathcal{S} \\ \forall s \in \mathcal{S}, \mathcal{E}(s) = Q(s) \\ L = \frac{\sum_{s \in \mathcal{S}} Q(s)}{2} \\ m = 2 \end{cases}$$

Prenons un exemple simple de la réduction :

- Partition :

  $\mathcal{S} = \{s_1,s_2,s_3,s_4,s_5,s_6\}$

  $Q(s_1,s_2,s_3,s_4,s_5,s_6) = \{6,5,4,1,3,7\}$

  La réponse est oui car :

  $\mathcal{S}_1 = \{s_1,s_6\}$ et $\mathcal{S}_2 = \{s_2,s_3,s_4,s_5\}$

- Ordonnancement multiprocesseur avec contrainte de latence :

  $\mathcal{G} = \{t_1,t_2,t_3,t_4,t_5,t_6\}$

  $\mathcal{E}(t_1,t_2,t_3,t_4,t_5,t_6) = \{6,5,4,1,3,7\}$

  Réponse : comme on peut le constater sur la figure 3.4, la réponse est oui.

Par conséquent l'ordonnancement multiprocesseur avec contrainte de latence appartient à la classe des problèmes NP et il est NP-difficile. Le fait de rajouter la contrainte de précédence impose que l'ordonnancement d'une tâche ne doit jamais précéder l'ordonnancement de ses prédécesseurs ainsi que des communications entre processeurs qu'il faut prendre en compte.

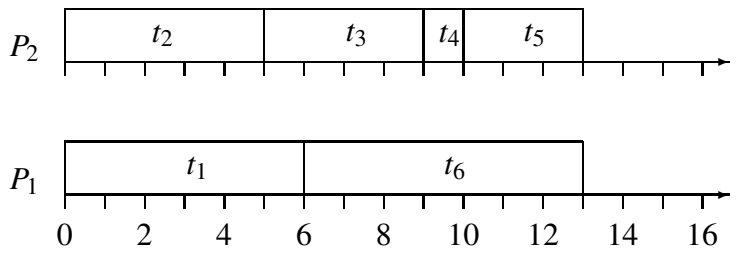

FIG. 3.4 – *Ordonnancement des tâches de l'ensemble $\mathcal{G}$ sur deux processeurs*

3.3.2.2 Étude d'ordonnançabilité

Vu que le problème que nous traitons est NP-difficile, il est impossible de trouver un algorithme qui le résout en un temps polynomial (sauf si NP=P) et ceci reste valable pour la condition d'ordonnançabilité. En d'autres termes on ne peut pas dire, à l'aide d'une condition nécessaire et suffisante, si un système de tâches avec des contraintes de latence et de précédence est ordonnançable en un temps polynomial. Cependant, on peut définir des conditions nécessaires ou suffisantes qui se basent sur le résultat monoprocesseur.

REMARQUE 3.2
La condition nécesssaire et suffisante dans le cas monoprocesseur (3.2) devient une condition suffisante seulement dans le cas multiprocesseur

Pour commencer, le théorème 3.1 introduit une condition nécessaire seulement.

THÉORÈME 3.1
Si une contrainte de latence $L(t_a,t_b)$ est définie entre deux tâches t_a et t_b, alors les tâches appartenant à $\mathcal{V}(t_a,t_b)$ sont ordonnançables si :

$$\sum_{t_i \in Ch_{pl}(t_a,t_b)} C(t_i) \leq L(t_a,t_b) \tag{3.3}$$

Preuve
Raisonnons par l'absurde pour prouver ce théorème. Supposons qu'un ordonnancement qui satisfait les contraintes de précédence et la latence $L(t_a,t_b)$ existe pour les tâches de l'ensemble $\mathcal{V}(t_a,t_b)$ alors que $\sum_{t_i \in Ch_{pl}(t_a,t_b)} C(t_i) > L(t_a,t_b)$. Par définition $Ch_{pl}(t_a,t_b)$ est le chemin le plus long d'entre tous les chemins qui relient la tâche t_a à la tâche t_b et le fait que sa longueur soit plus importante que la valeur de $L(t_a,t_b)$ implique que $(E(t_b) - S(t_a))$ est inévitablement plus large que $L(t_a,t_b)$ (si ces tâches sont ordonnancées sur plusieurs processeurs le résultat de $(E(t_b) - S(t_a))$ s'allonge à cause des communications). Donc, aucun ordonnancement ne peut satisfaire la contrainte $L(t_a,t_b)$. Ce qui contredit la supposition faite au départ et prouve ce théorème□

Une tâche qui n'est pas soumise à une contrainte de latence est ordonnancée suivant la contrainte de précédence seulement, ce qui nous mène à différencier entre les tâches soumises à la contrainte

de précédence seulement et les tâches soumises aux contraintes de latence et de précédence. L'idée est de regrouper ces dernières en une seule super-tâche et d'ordonnancer le contenu de cette super-tâche séparément. Les autres tâches du graphe ainsi que la super-tâche seront ordonnancées avec un algorithme traitant la contrainte de précédence seulement puisque la contrainte de latence, étant à l'intérieur de la super-tâche, est éliminée du graphe d'algorithme initial. La figure 3.5 illustre l'exemple d'une application temps réel où le graphe d'algorithme comporte une contrainte de latence (les tâches concernées par la latence sont en blanc). À partir de ce graphe on a regroupé les tâches concernées par la latence en une super-tâche qui représente une partie du graphe qui va être ordonnancée séparément. Le restant des tâches (en noir) ainsi que la super-tâche (qui sera considérée comme une tâche dont la durée d'exécution est égale au temps d'ordonnancement des tâches qu'elle contient) vont être ordonnancées avec un algorithme de liste (cet algorithme est introduit et détaillé à la section 6.1.4.2 dans la deuxième partie du manuscrit). Il est important de noter que les précédences sont respectées car la super-tâche est le successeur de toutes les tâches du graphe qui sont des prédécesseurs d'une ou plusieurs tâches appartenant à la super-tâche d'un côté, et elle est le prédécesseur de toutes les tâches du graphe qui sont des successeurs d'une ou plusieurs tâches appartenant à la super-tâche de l'autre.

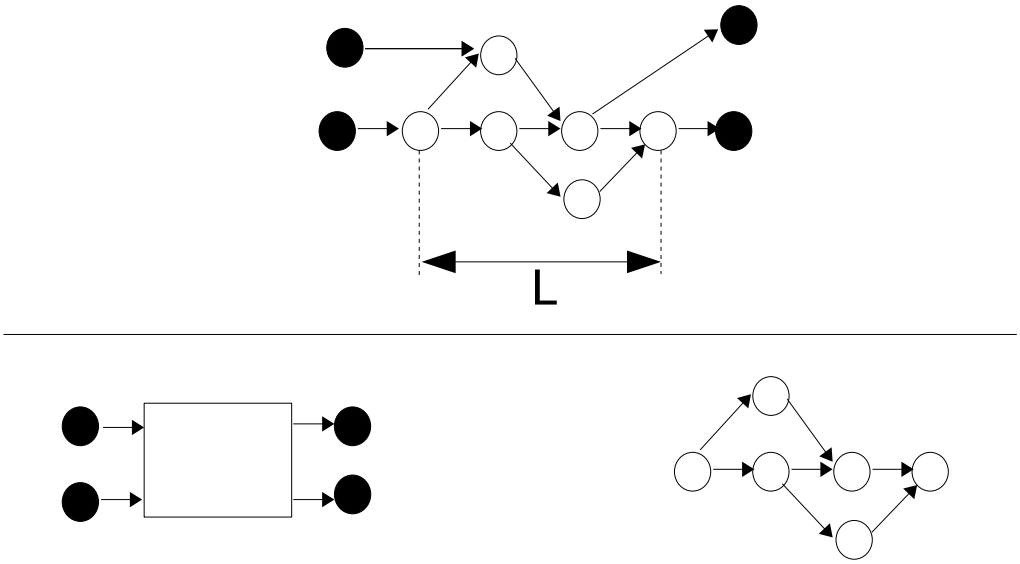

FIG. 3.5 – *Séparation de la super-tâche du graphe d'algorithme*

La difficulté de traiter une contrainte de latence est étroitement liée à la forme du graphe ou plus précisément à la forme de la partie du graphe concernée par la latence. C'est-à-dire que, contrairement au problème du chapitre précédent où la précédence n'influait pas sur l'ordonnançabilité des tâches, dans le cas présent la condition d'ordonnançabilité n'est pas la même si toutes les tâches soumises à la contrainte de latence appartiennent à un seul chemin ou si elles sont réparties sur plusieurs chemins. Afin de s'adapter à la forme de cette partie du graphe on propose d'appliquer

un algorithme de clusterisation.

Clusterisation

La clusterisation commence par déterminer tous les chemins qui vont de la tâche source à la tâche destination. Après les avoir triés par ordre décroissant de longueur (on rappelle que la longueur d'un chemin est la somme des durées d'exécution des tâches qu'il contient) on détermine le chemin le plus long "Ch_{pl}", et puis en allant du plus long au plus court on élimine tout chemin ne comportant que des tâches qui appartiennent déjà à d'autres chemins plus longs que lui. L'ensemble des tâches de chaque chemin est considéré comme un cluster de tâches, et celui issu du chemin le plus long Ch_{pl} est appelé cluster principal. Ainsi chaque cluster contient au moins une tâche qui n'est dans aucun autre cluster. L'algorithme 7 réalise la construction des clusters.

Algorithme 7 Algorithme de construction de clusters

1: Repérer tous les chemins entre la tâche source et la tâche destination
2: **pour** Chaque chemin *ch* obtenu **faire**
3: Calculer $\sum_{t_i \in ch} C(t_i)$
4: **fin pour**
5: Trier les chemins par ordre décroissant de longueur
6: Construire le cluster principal à partir du chemin le plus long Ch_{pl}
7: Initialiser un ensemble ω qui contient toutes les tâches de Ch_{pl}
8: **pour** Chaque chemin *ch* (par ordre décroissant de longueur) **faire**
9: **si** Toutes les tâches du chemin *ch* se trouvent dans ω **alors**
10: Supprimer *ch*
11: **sinon**
12: Ajouter dans ω les tâches qui sont dans *ch* et n'appartiennent pas à ω
13: Construire un cluster à partir de *ch*
14: **fin si**
15: **fin pour**

Pour des graphes avec un nombre important de noeuds et d'arcs le nombre de chemins est très grand. Néanmoins dasn le cas d'un graphe acyclique (comme c'est le cas dans notre modèle) ce prblème n'est pas NP-difficile [152].

D'après [153] il existe différentes approches pour déterminer tous les chemins d'un graphe, parmi lesquelles on peut citer l'approche par sous-graphes ou celle de l'arbre de couvrant.

EXEMPLE 3.2
*La figure 3.6 présente un exemple de construction de clusters pour un sous-graphe d'un graphe plus général non représenté ici. Avec Cl*1 *comme cluster principal, les différentes configurations que les clusters peuvent avoir sont :*

1. les clusters peuvent être indépendants comme le sont *Cl*3 et *Cl*7,
2. les clusters peuvent se superposer de plusieurs manières :
   (a) les clusters peuvent se superposer en pyramide comme le font les clusters *Cl*6 et *Cl*7,
   (b) un cluster peut se placer à cheval sur deux clusters comme l'est *Cl*8 sur *Cl*7 et *Cl*3,
   (c) un cluster peut entourer un autre cluster comme le font les clusters *Cl*2 et *Cl*3 ou *Cl*6 et *Cl*7.
3. les clusters peuvent se croiser comme le font les clusters *Cl*4 et *Cl*5.

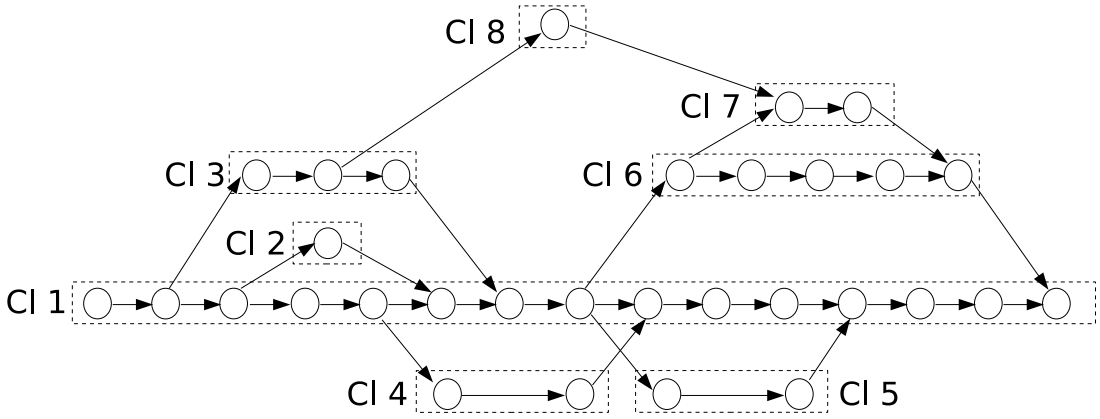

FIG. 3.6 – *Exemple de construction de clusters*

Il est important de préciser que tous les clusters commencent par la tâche source et finissent par la tâche destination. Pour chaque cluster (hormis le principal) nous n'avons mis en pointillé uniquement la partie qui le distingue des autres clusters. Le but d'étudier les différentes configurations que les clusters peuvent avoir est de proposer une condition d'ordonnançabilité relative à chaque configuration. Par exemple si on dispose d'un graphe où les clusters construits sont tous indépendants (première configuration) alors le test d'ordonnançabilité est plus simple que si les clusters sont de la deuxième ou la troisième configuration.

Pour commencer on étudie le cas de deux clusters (le théorème suivant introduit une condition d'ordonnançabilité dans ce cas). Si Cl est un cluster alors on note par $|Cl|$ la longueur de ce cluster. La longueur d'un cluster comprend toutes les durées d'exécution des tâches qu'il contient ainsi que les coûts de communication avec des tâches qui appartiennent aux autres clusters. Il n'y a que le cluster principal qui a une longueur égale à la somme des durées d'exécution de ses tâches seulement puisque on suppose que toutes ses tâches sont ordonnancées sur le même processeur. On note par $\{Cl\}$ l'ensemble des tâches que contient le cluster Cl.

Dans la suite du chapitre nous admettons que la condition nécessaire du théorème 3.1 est vérifiée, ce qui garantit que la longueur du cluster principal est inférieure ou égale à la valeur de

latence. C'est justement pour ne pas provoquer de communications qu'on privilégie l'ordonnancement des tâche de ce cluster sur le même processeur. Le théorème suivant introduit une condition nécessaire et suffisante d'ordonnançabilité dans le cas de basique de deux clusters.

THÉORÈME 3.2
Soit L la latence imposée sur une partie du graphe où deux clusters (Cl_{pl} et Cl_{sd} sont construits) avec $|Cl_{pl}| = \sum_{t_i \in \{Cl_{pl}\}} C(t_i)$. Ce système est ordonnançable si et seulement si :

1. dans le cas où le nombre de processeurs est supérieur à 1 :

$$|Cl_{sd}| \leq L \tag{3.4}$$

2. dans le cas où le nombre de processeurs est égal à 1 :

$$\sum_{t_i \in \{Cl_{sd}\} \setminus (\{Cl_{pl}\} \cap \{Cl_{sd}\})} C(t_i) + |Cl_{pl}| \leq L \tag{3.5}$$

Preuve
– dans le cas où le nombre de processeurs est égal à 1 toutes les tâches doivent être ordonnancées sur le même processeur d'où l'utilisation de la condition monoprocesseur 3.2, l'ensemble $\{Cl_{sd}\} \setminus (\{Cl_{pl}\} \cap \{Cl_{sd}\})$ est l'ensemble des tâches qui appartiennent uniquement au cluster Cl_{sd}
– dans le cas où le nombre de processeurs est supérieur à 1 on traite le cas où chaque cluster est ordonnancé sur un processeur différent car le cas où ils sont ordonnancés sur le même processeur a été vu avant. Comme l'ordonnancement se fait sur deux processeurs le temps global d'ordonnancement est égal au max($|Cl_{pl}|$,$|Cl_{sd}|$). Comme la condition du théorème 3.1 est vérifiée alors : $\sum_{t_i \in \{Cl_{pl}\}} C(t_i) \leq L$. Par ailleurs si l'équation 3.4 est vérifiée alors on peut en déduire que $\max(|Cl_{pl}|,|Cl_{sd}|) \leq L$, ce qui prouve que la condition 3.4 est une condition nécessaire.
Pour prouver que la condition 3.4 est suffisante on suppose que $|Cl_{sd}| \geq L$ et que le système est ordonnançable. D'après la définition d'un cluster on sait que Cl_{sd}, commence par la tâche source et finit par la tâche destination de la latence L. Donc dire que $|Cl_{sd}| \geq L$ signifie que le temps global d'ordonnancement est supérieur à la valeur de la latence et par conséquent que celle-ci n'est plus respectée. Ce qui contredit la supposition du départ et prouve que la condition 3.4 est une condition suffisante ⊡

Ensuite on cherche à concevoir des conditions d'ordonnançabilité plus générales spécifiques à chaque configuration citée avant.

Pour commencer on étudie le premier cas. Le théorème suivant introduit une condition nécessaire et suffisante d'ordonnançabilité dans le cas de n clusters secondaires indépendants comme le montre la figure 3.7.

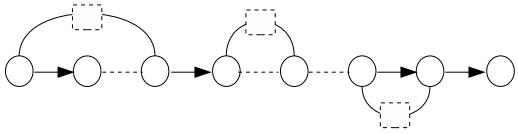

FIG. 3.7 – *Exemple de clusters indépendants*

THÉORÈME 3.3
Soit $\{Cl_1,...,Cl_n\}$ *l'ensemble de n clusters indépendants obtenus, en plus de* Cl_{pl}*, après l'exécution de l'algorithme 7. Dans le cas où le nombre de processeurs est supérieur à* 1 *ce système est ordonnançable si et seulement si :*

$$|Cl_{pl}| + (\sum_{q\in\{1,..,n\}\ tel\ que\ |Cl_q|>|Cl_{pl}|} |Cl_q| - |Cl_{pl}|) \leq L \tag{3.6}$$

Preuve
Au départ (avant d'ordonnancer) le temps global d'ordonnancement est égal à $|Cl_{pl}|$. Si $|Cl_q| > |Cl_{pl}|$ alors le cluster Cl_q rallonge le temps global d'ordonnancement de $|Cl_q| - |Cl_{pl}|$. Donc, et comme tous les clusters sont indépendants, au final le temps global d'ordonnancement est égal à la somme des rallongements relatifs à chaque cluster en plus de sa valeur initiale $|Cl_{pl}|\boxdot$

On s'intéresse à présent au deuxième cas. Le théorème suivant introduit une condition nécessaire et suffisante d'ordonnançabilité dans le cas de *n* clusters qui se superposent comme le montre la figure 3.8.

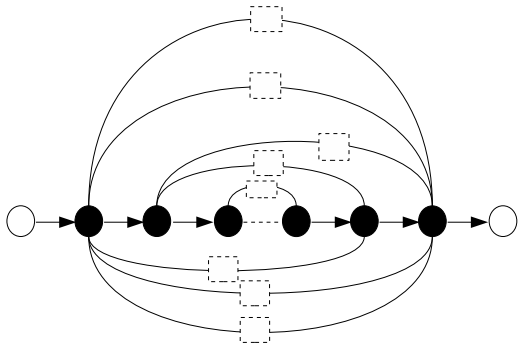

FIG. 3.8 – *Exemple de clusters qui se superposent*

THÉORÈME 3.4
Soit $\{Cl_1,...,Cl_n\}$ *l'ensemble de n clusters qui se superposent. Ce système est ordonnançable si et seulement si (on admet que le nombre de processeurs est supérieur à 1) :*

$$max_{q\in\{1,..,n\}}(|Cl_q|) \leq L \tag{3.7}$$

Preuve
Comme pour la preuve du précédent théorème si $|Cl_q| > |Cl_{pl}|$ alors le cluster Cl_q rallonge le temps global d'ordonnancement de $|Cl_q| - |Cl_{pl}|$. Mais plus précisément Cl_q rallonge le temps d'ordonnancement du chemin qui se situe entre les deux tâches où le ou les clusters qui entourent tous les autres clusters bifurquent (les tâches en noir sur la figure 3.8). Tant que tous les clusters Cl_q rallongent le même chemin, le temps global d'ordonnancement sera égale à la longueur du cluster le plus long, c'est-à-dire celui qui provoque le rallongement le plus important⊡

Enfin on s'intéresse au troisième et dernier cas. Le théorème suivant introduit une condition nécessaire et suffisante d'ordonnançabilité dans le cas de n clusters tel que le q ème croise le $(q+1)$ ème cluster ($q \in N^*, q < n$) comme le montre la figure 3.9.

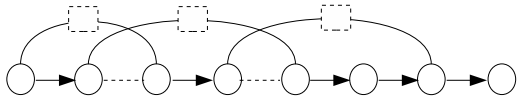

FIG. 3.9 – *Exemple de clusters qui se croisent*

THÉORÈME 3.5
Soit $\{Cl_1,...,Cl_n\}$ l'ensemble de n clusters tel que le q ème croise le $q+1$ ème cluster ($q \in N^*, i < n$). Ce système est ordonnançable si et seulement si (on admet que le nombre de processeurs est supérieur à 1) :
$\forall q \in \{1,..,n\}$ *tel que* $(|Cl_q| > |Cl_{pl}|)$,

$$\sum_{q}(|Cl_q| - (|Cl_{pl}| + \sum_{(t_i \in \{Cl_{pl}\} \setminus (((\{Cl_q\} \cap \{Cl_{pl}\}) \cup (\{Cl_{q-1}\} \cap \{Cl_{pl}\}),\ i<n)} C(t_i))) \leq L \qquad (3.8)$$

Preuve
En combinant les preuves des deux précédents théorèmes on aboutit à la preuve du théorème 3.5. En effet deux clusters Cl_i et Cl_{i+1}, qui se croisent, rallongent chacun de son côté le temps d'ordonnancement d'un chemin dont les tâches appartiennent au cluster principal. Comme ces deux chemins partagent des tâches alors la somme des durées d'exécution de ces tâches doivent être déduite du rallongement provoqué par les clusters Cl_i et Cl_{i+1}⊡

Pour résumer, Le but de la clusterisation est de répartir les tâches en clusters dont les longueurs peuvent être calculées tout en prenant en compte les coûts de communication entre les tâches de ces clusters. Les différents théorèmes énoncés permettent de tester l'ordonnançabilité suivant toutes les configurations possibles que peuvent avoir ces clusters.

Union de clusters

Les théorèmes énoncés dans la clusterisation suppose que chaque cluster est ordonnancé sur un processeur et qu'on dispose, à chaque fois, du nombre nécessaire de processeurs. Les clusters qui

jouent un rôle, dans la suite de l'étude, sont ceux dont la longueur est supérieure à celle du cluster principal puisqu'ils rallongent le temps global d'ordonnancement. Dès lors on en déduit que pour un cluster Cl_i :

- si $|Cl_i| > |Cl_{pl}|$ le fait d'ordonnancer ce cluster sur un autre processeur que celui de Cl_{pl} provoque le rallongement du temps global d'ordonnancement entraînant peut être le non respect de la contrainte de la latence. Alors que le fait de les ordonnancer sur le même processeur peut permettre le respect de la contrainte puisque les coûts de communication sont supprimés ;
- si $|Cl_i| < |Cl_{pl}|$ le fait d'ordonnancer ce cluster sur un autre processeur que celui de Cl_{pl} ne provoque pas le rallongement du temps global d'ordonnancement. Néanmoins, si il existe plusieurs clusters dans ce cas, on ne sait pas si le nombre de processeurs, dont dispose l'architecture, suffit pour ordonnancer tous ces clusters.

Ceci est aussi vrai pour deux clusters autres que Cl_{pl} qui sont liés par des dépendances. Car le fait de les ordonnancer sur deux processeurs différents peut provoquer le rallongement de l'un d'eux le rendant plus long que la contrainte de latence. Dire qu'un cluster est ordonnancé sur un autre processeur que celui de Cl_{pl} signifie que les tâches de l'ensemble $\{Cl_i\}$ qui ne sont pas dans l'ensemble $\{Cl_{pl}\}$ sont ordonnancées sur un autre processeur que celui ou les tâches de $\{Cl_{pl}\}$ sont ordonnancées.
L'algorithme dont on a besoin doit tenter, suivant les longueurs des différents clusters obtenus et le nombre de processeurs de l'architecture, d'unir ces clusters. L'union de deux clusters consiste à construire un nouveau cluster qui va contenir les tâches des deux clusters (sans répéter les tâches communes aux deux clusters). Cette union permet, d'un côté, d'éliminer les clusters plus longs que la contrainte de latence (car cet algorithme unit en priorité les clusters qui communiquent entre eux) et, d'un autre côté, d'aboutir à un nombre de clusters inférieur ou égal au nombre de processeurs. L'algorithme 8 détaille l'union des clusters (L est la contrainte de latence et m est le nombre de processeurs de l'architecture).

REMARQUE 3.3
L'union d'un cluster, quel qu'il soit, et du cluster principal produit un cluster principal.

3.3.3 Algorithme glouton d'ordonnancement

Maintenant que les algorithmes de clusterisation et d'union ont été présentés, on est en mesure d'introduire l'algorithme d'ordonnancement de tâches avec des contraintes de précédence et une contrainte de latence. Cet algorithme considère que toutes les tâches concernées par la contrainte de latence constituent une super-tâche. Comme dans le chapitre précédent, l'algorithme d'ordonnancement est une variante de l'algorithme d'ordonnancement de type liste - détaillé dans la section 6.1.4.2 dans la deuxième partie du manuscrit - dans lequel on a introduit des instructions supplémentaires qui traitent la super-tâche. Lorsque l'algorithme proposé procède à l'ordonnancement de la super-tâche il fait appel aux algorithmes de clusterisation et d'union. Suivant le résultat obtenu,

Algorithme 8 Algorithme d'union des clusters

si Le nombre de clusters est inférieur à m et toutes les longueurs des clusters sont inférieures à L **alors**
  Considérer que ce système est ordonnançable
sinon
  tant que L'ensemble ω, des clusters Cl_i tel que : $|Cl_i| > L$, n'est pas vide **faire**
    Unir Cl_i avec l'un des clusters avec qui il communique
    si Le cluster obtenu est le cluster principal et il est plus long que L **alors**
      Considérer que ce système n'est pas ordonnançable
    sinon
      Mettre ce cluster (si il est plus long que L) dans l'ensemble ω
    fin si
  fin tant que
  si Le nombre de clusters est inférieur à m **alors**
    Considérer que ce système est ordonnançable
  sinon
    tant que Le nombre de clusters est supérieur à m **faire**
      si Il existe une paire de clusters dont le résultat de l'union produit un cluster de longueur inférieure à L **alors**
        Unir ces deux clusters
      sinon
        Considérer que ce système n'est pas ordonnançable et sortir de la boucle tant que
      fin si
    fin tant que
    Considérer que ce système est ordonnançable
  fin si
fin si

soit l'algorithme s'interrompt et annonce que le système n'est pas ordonnançable, soit il ordonnance les tâches de chaque cluster sur un processeur différent tout en respectant les précédences entre les tâches. En plus cet algorithme minimise le makespan de toutes les tâches du graphe d'algorithme. L'algorithme 9 détaille l'algorithme d'ordonnancement avec une seule contrainte de latence.

EXEMPLE 3.3
Pour illustrer l'algorithme d'ordonnancement 9 on traite un système composé du graphe d'algorithme de la figure 3.10 et d'un graphe d'architecture, composé de deux processeurs reliés par un médium de communication.. Le graphe d'algorithme est composé de onze tâches $\{t_a,t_b,t_c,t_d,t_e,t_f,t_g,t_h,t_i,t_j,t_k\}$

Algorithme 9 Algorithme glouton d'ordonnancement avec des contraintes de précédence et une seule contrainte de latence

1: Repérer toutes les tâches concernées par la même latence et former une super-tâche
2: Initialisation de l'ensemble Δ des candidats avec les tâches sans prédécesseurs
3: **tant que** L'ensemble Δ n'est pas vide **faire**
4: **pour** *i* allant de 1 au nombre de candidats **faire**
5: Calcul de la fonction de coût pour chaque candidat sur tous les processeurs
6: Choisir la fonction de coût minimale entre toutes celles calculées sur chacun des processeurs (le processeur en question est considéré comme le meilleur processeur pour cette tâche). Cela forme l'ensemble des paires (tâche, meilleur processeur)
7: **fin pour**
8: **pour** *i* allant de 1 au nombre de paires (tâche, meilleur processeur) **faire**
9: Choisir la paire (tâche, meilleur processeur) qui a la fonction de coût maximale
10: **fin pour**
11: **si** La tâche choisie est la super tâche **alors**
12: Exécuter l'algorithme de clusterisation (Algorithme 7)
13: Exécuter l'algorithme d'union de clusters (Algorithme 8)
14: **si** Le système est ordonnançable **alors**
15: Ordonnancer les tâches du cluster principal sur le meilleur processeur de la super-tâche et chaque cluster sur un processeur différent en respectant les dépendances entre les tâches
16: **sinon**
17: Considérer que ce système n'est pas ordonnançable avec cet algorithme
18: **fin si**
19: **fin si**
20: Ordonnancer cette tâche sur son meilleur processeur
21: Ajouter à l'ensemble Δ, des successeurs ordonnançables de ce candidat
22: Suppression du candidat ordonnancé de l'ensemble Δ
23: **fin tant que**

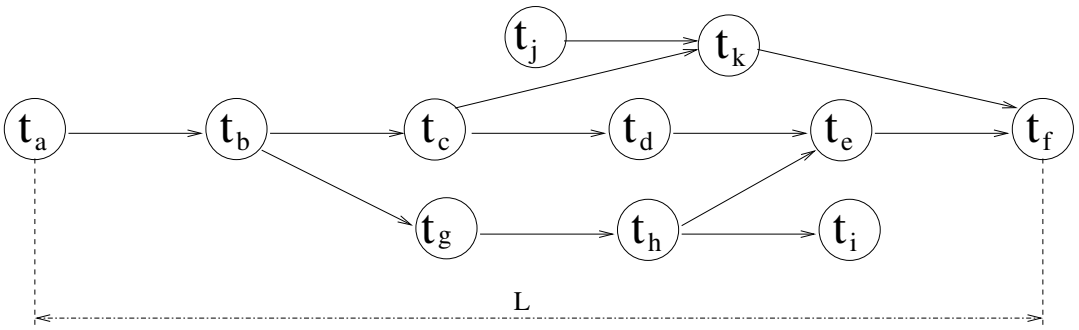

FIG. 3.10 – *(Exemple 3.3) Graphe d'algorithme*

de durée d'exécution égale à 1 et le coût d'une communication entre les processeurs a aussi une valeur de 1. La valeur de la contrainte de latence entre les tâches t_a et t_f est de $L = 8$.

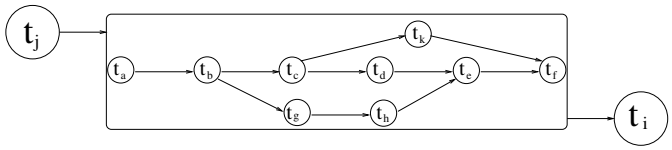

FIG. 3.11 – *(Exemple 3.3) Graphe résultant de la séparation de la super-tâche du graphe d'algorithme*

D'après le graphe de la figure 3.10 les tâches qui sont concernées par la contrainte de latence L sont $\{t_a,t_b,t_c,t_d,t_e,t_f,t_g,t_h,t_i\}$. C'est donc ces tâches qui forment la super-tâche tel qu'il est expliqué dans la section 3.3.2.2 (le graphe résultant est donné par la figure 3.11).

L'exécution de l'algorithme 9 se déroule comme suit :

- la tâche t_j est la première tâche candidate à l'ordonnancement et elle est ordonnancée sur le processeur P_1,
- la deuxième tâche candidate à l'ordonnancement est la super-tâche et le processeur qui minimise la fonction de coût dans ce cas est P_1 (comme t_j a été ordonnancée sur P_1 alors l'ordonnancement de la super-tâche sur P_2 engendre une communication qui rallonge le makespan) :
  - en exécutant l'algorithme de clusterisation on obtient les clusters suivants : $\{Cl_{pl}\} = \{t_a,t_b,t_c,t_d,t_e,t_f\}$, $\{Cl_{sd_1}\} = \{t_k\}$ et $\{Cl_{sd_2}\} = \{t_g,t_h\}$,
  - en exécutant l'algorithme d'union le cluster $\{Cl_{sd_1}\}$ est uni avec le cluster principal et on se retrouve avec deux clusters seulement : $\{Cl_{pl}\} = \{t_a,t_b,t_c,t_d,t_e,t_f,t_k\}$ et $\{Cl_{sd}\} = \{t_g,t_h\}$,
  - les tâches du cluster Cl_{pl} sont ordonnancées sur le processeur P_1 et celles de Cl_{sd} sont ordonnancées sur P_2 en respectant les coûts de communication entre les tâches t_b et t_g ainsi que t_h et t_e.

– la tâche t_i est la dernière tâche à être ordonnancée et c'est le processeur P_2 qui va l'exécuter.

La figure 3.12 montre le résultat de l'ordonnancement de ce système. On constate que la contrainte de latence L est satisfaite puisque le temps global d'ordonnancement (entre la tâche source t_a et la tâche destination t_f) est égal à 8.

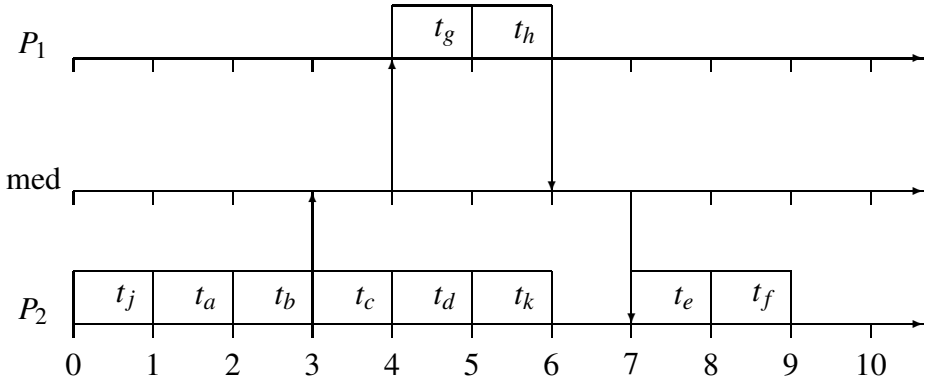

FIG. 3.12 – *(Exemple 3.3) Résultat de l'ordonnancement*

3.4 Étude du cas de plusieurs contraintes de latences

Quand plusieurs contraintes de latence sont imposées dans le même système de tâches, ces contraintes peuvent être ou non liées. Deux contraintes de latences sont liées signifie qu'un chemin entre une tâche concernée par la première latence et une autre tâche concernée par la deuxième latence existe. D'après [58] il existe quatre possibilités de liaisons entre deux ensembles de tâches concernées par deux contraintes de latences différentes :

– **Latence en ||** : aucun chemin entre les deux ensembles n'existe (L_1 et L_3 dans la figure 3.13),
– **Latence en Z** : il existe soit un ou plusieurs chemins qui relient une ou plusieurs tâches du première ensemble avec une ou plusieurs tâches du deuxième ensemble, soit l'inverse (L_1 et L_2 dans la figure 3.13),
– **Latence en X** : il existe un ou plusieurs chemins qui relient une ou plusieurs tâches du première ensemble avec une ou plusieurs tâches du deuxième ensemble et inversement. Contrairement aux deux premières possibilités, l'intersection entre les deux ensembles peut ne pas être l'ensemble vide (L_2 et L_3 dans la figure 3.13).

La condition d'ordonnançabilité dans le cas de plusieurs latences varie selon le type de liaisons qui relient ces latences. La prochaine section présente l'étude d'ordonnançabilité effectuée dans le cas monoprocesseur.

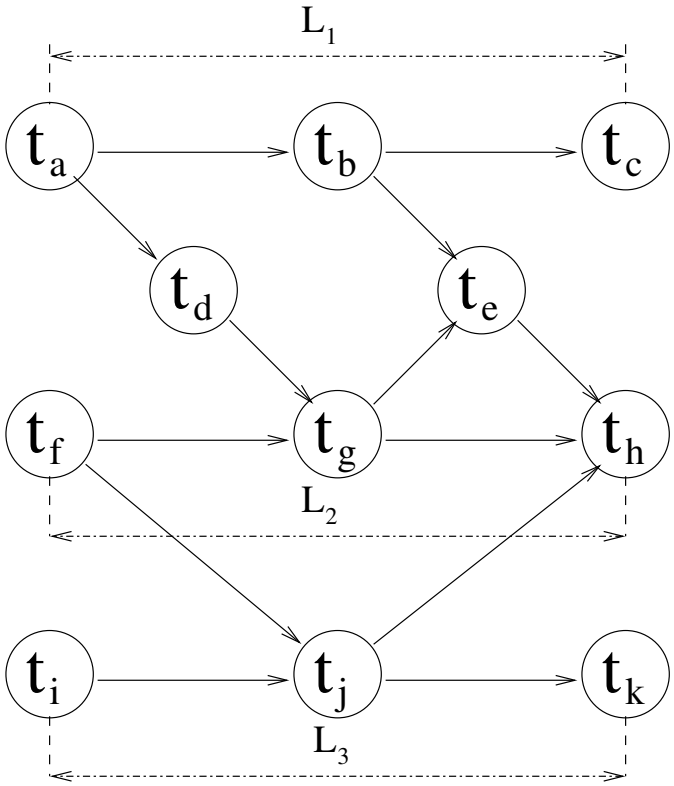

FIG. 3.13 – *Les différents types de liaisons entre les contraintes de latence*

3.4.1 Cas monoprocesseur

La première condition concerne les systèmes de tâches où les paires de latences sont soit en ||, soit en Z (ou une partie en || et l'autre en Z). Car, vis-à-vis de l'ordonnançabilité, ces deux cas se traitent de la même manière [58]. Même si le cas en Z impose un ordre partiel entre les deux ensembles de tâches concernées par les deux latences (alors que le cas || n'impose aucun ordre) cet ordre n'influe pas sur l'ordonnançabilité puisque chaque ensemble peut être ordonnancé séparément. Par conséquent la condition 3.2 reste valable. Le théorème suivant introduit une condition d'ordonnançabilité dans ce cas précis.

THÉORÈME 3.6
Soient $\{L_1,..,L_n\}$ *n contraintes de latences imposées respectivement entre les tâches* $(t_{a_1},t_{b_1}),..,(t_{a_n},t_{b_n})$. *On admet que toutes les latences sont soit en relation ||, soit en relation Z. Ce système de tâches est ordonnançable si et seulement si :*

$$\forall i \in \{1,..,n\} \quad \sum_{t_i \in \mathcal{V}(t_{a_i},t_{b_i})} C(t_i) \leq L(t_{a_i},t_{b_i}) \tag{3.9}$$

Preuve

Pour chaque paire de latences ($L(t_{a_i},t_{b_i})$,$L(t_{a_{i+1}},t_{b_{i+1}})$) on a $\mathcal{V}(t_{a_i},t_{b_i}) \cap \mathcal{V}(t_{a_{i+1}},t_{b_{i+1}}) = \emptyset$. Du coup il suffit d'appliquer la condition 3.2 pour chaque paire de latences, et ce qui revient à appliquer la condition 3.9□

On s'intéresse à présent à la condition d'ordonnançabilité des systèmes de tâches d'une paire de latences (L_1,L_2) en X. Ce cas a été déjà traité dans [58], mais la condition donnée est incomplète. Nous améliorons cette condition en divisant chacun des deux ensembles de tâches concernées par les latences en quatre sous-ensembles disjoints. Les quatre sous-ensembles issus de l'ensemble des tâches concernées par L_1 sont :

1. le sous-ensemble des tâches qui ont au moins un successeur dans l'ensemble des tâches concernées par L_2. On dénote cet ensemble par $\mathcal{V}_1(L_1)$. Par exemple dans la figure 3.14, $\mathcal{V}_1(t_a,t_e) = \{t_a,t_b\}$.
2. le sous-ensemble des tâches qui n'ont ni prédécesseur ni successeur dans l'ensemble des tâches concernées par L_2 (ce sous-ensemble peut être vide). On dénote cet ensemble par $\mathcal{V}_2(L_1)$. Par exemple dans la figure 3.14, $\mathcal{V}_2(t_a,t_e) = \{t_c\}$.
3. le sous-ensemble des tâches qui ont au moins un prédécesseur dans l'ensemble des tâches concernées par L_2. On dénote cet ensemble par $\mathcal{V}_3(L_1)$. Par exemple dans la figure 3.14, $\mathcal{V}_3(t_a,t_e) = \{t_d,t_e\}$.
4. le sous-ensemble de tâches communes aux deux ensembles concernés par L_1 et L_2 (ce sous-ensemble peut être vide). On dénote cet ensemble par $\mathcal{V}_4$. Par exemple dans la figure 3.14, $\mathcal{V}_4 = \{t_f,t_g\}$.

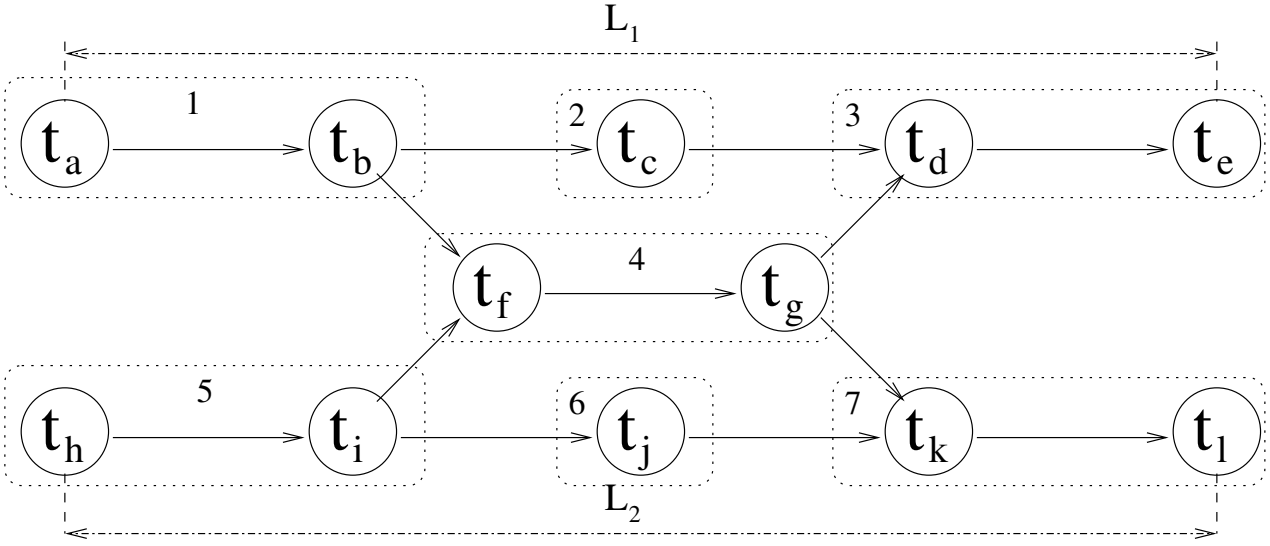

FIG. 3.14 – *(Preuve du théorème 3.7) Paire de latences en X*

De ce fait les tâches sous la contraintes de latences en X sont regroupées en sept sous-ensembles liés par des précédences. La condition d'ordonnançabilité va dépendre de la position dans l'ordonnancement de chaque ensemble par rapport aux autres, i.e., le fait d'ordonnancer les tâches d'un sous-ensemble avant ou après d'autres tâches d'un autre sous-ensemble influe sur le temps global d'ordonnancement des deux latences. Le théorème suivant reprend les conditions en fonction de toutes les positions possibles que peuvent occuper les sept différents sous-groupes.

THÉORÈME 3.7
Soient L_1 et L_2 deux latences en X entre, respectivement, les paires de tâches (t_{a1},t_{b1}) et (t_{a2},t_{b2}). En appliquant les définitions données plus tôt, $\mathcal{V}(t_{a1},t_{b1})$ et $\mathcal{V}(t_{a2},t_{b2})$ sont divisés, respectivement, en ($\mathcal{V}_1(t_{a1},t_{b1})$, $\mathcal{V}_2(t_{a1},t_{b1})$, $\mathcal{V}_3(t_{a1},t_{b1})$) et ($\mathcal{V}_1(t_{a2},t_{b2})$, $\mathcal{V}_2(t_{a2},t_{b2})$, $\mathcal{V}_3(t_{a2},t_{b2})$). Les tâches de ce système sont ordonnançables si et seulement si :

$$\begin{cases} \sum_{t\in\mathcal{V}(t_{ai},t_{bi})} C(t) \leq L_i \\ \sum_{t\in\mathcal{V}(t_{ai},t_{bi})\cup\mathcal{V}(t_{aj},t_{bj})} C(t) \leq L_j \end{cases}, (i=1,j=2)\vee(i=2,j=1) \tag{3.10}$$

ou

$$\begin{cases} \sum_{t\in\mathcal{V}(t_{ai},t_{bi})} C(t) + \sum_{t\in\mathcal{V}_3(t_{aj},t_{bj})} C(t) \leq L_i \\ \sum_{t\in\mathcal{V}(t_{ai},t_{bi})\cup\mathcal{V}(t_{aj},t_{bj})} C(t) - (\sum_{t\in\mathcal{V}_3(t_{ai},t_{bi})} C(t) + \sum_{t\in\mathcal{V}_2(t_{ai},t_{bi})} C(t)) \leq L_j \end{cases}, (i=1,j=2)\vee(i=2,j=1) \tag{3.11}$$

ou

$$\begin{cases} \sum_{t\in\mathcal{V}(t_{ai},t_{bi})} C(t) + \sum_{t\in\mathcal{V}_1(t_{aj},t_{bj})} C(t) \leq L_i \\ \sum_{t\in\mathcal{V}(t_{ai},t_{bi})\cup\mathcal{V}(t_{aj},t_{bj})} C(t) - (\sum_{t\in\mathcal{V}_1(t_{ai},t_{bi})} C(t) + \sum_{t\in\mathcal{V}_2(t_{ai},t_{bi})} C(t)) \leq L_j \end{cases}, (i=1,j=2)\vee(i=2,j=1) \tag{3.12}$$

Preuve
Chacune des équations 3.10, 3.11 et 3.12 s'intéresse à un ordre bien précis entre les sous-ensembles obtenus. Nous notons par $\mathcal{V}_4$ le sous-ensemble $\mathcal{V}(t_{ai},t_{bi})\cap\mathcal{V}(t_{aj},t_{bj})$.

L'équation 3.10 s'applique soit au cas où les sous-ensembles $\mathcal{V}_1(t_{ai},t_{bi})$, $\mathcal{V}_2(t_{ai},t_{bi})$, $\mathcal{V}_4$ et $\mathcal{V}_3(t_{ai},t_{bi})$ sont ordonnancés les uns après les autres ce qui contraint l'ordonnancement du sous-ensemble $\mathcal{V}_1(t_{aj},t_{bj})$ avant celui du sous-ensemble $\mathcal{V}_1(t_{ai},t_{bi})$ et le sous-ensemble $\mathcal{V}_3(t_{aj},t_{bj})$ après celui du sous-ensemble $\mathcal{V}_3(t_{ai},t_{bi})$ (le sous-ensemble $\mathcal{V}_2(t_{aj},t_{bj})$ peut être ordonnancé avant $\mathcal{V}_1(t_{ai},t_{bi})$ ou après $\mathcal{V}_3(t_{ai},t_{bi})$), soit au cas inverse où les sous-ensembles $\mathcal{V}_1(t_{aj},t_{bj})$, $\mathcal{V}_2(t_{aj},t_{bj})$, $\mathcal{V}_4$ et $\mathcal{V}_3(t_{aj},t_{bj})$ sont ordonnancés les uns après les autres ce qui contraint l'ordonnancement du sous-ensemble $\mathcal{V}_1(t_{ai},t_{bi})$ avant celui du sous-ensemble $\mathcal{V}_1(t_{aj},t_{bj})$ et le sous-ensemble $\mathcal{V}_3(t_{ai},t_{bi})$ après celui du sous-ensemble $\mathcal{V}_3(t_{aj},t_{bj})$ (le sous-ensemble $\mathcal{V}_2(t_{ai},t_{bi})$ peut être ordonnancé avant $\mathcal{V}_1(t_{aj},t_{bj})$ ou après $\mathcal{V}_3(t_{aj},t_{bj})$).

L'équation 3.11 s'applique soit au cas où l'ordre d'ordonnancement est $\mathcal{V}_1(t_{aj},t_{bj})$, $\mathcal{V}_2(t_{aj},t_{bj})$, $\mathcal{V}_1(t_{ai},t_{bi})$, $\mathcal{V}_4$, $\mathcal{V}_3(t_{aj},t_{bj})$, $\mathcal{V}_2(t_{ai},t_{bi})$, $\mathcal{V}_3(t_{ai},t_{bi})$, soit au cas où l'ordre est $\mathcal{V}_1(t_{ai},t_{bi})$, $\mathcal{V}_2(t_{ai},t_{bi})$, $\mathcal{V}_1(t_{aj},t_{bj})$, $\mathcal{V}_4$, $\mathcal{V}_3(t_{ai},t_{bi})$, $\mathcal{V}_2(t_{aj},t_{bj})$, $\mathcal{V}_3(t_{aj},t_{bj})$.

L'équation 3.12 s'applique soit au cas où l'ordre d'ordonnancement est $\mathcal{V}_1(t_{ai},t_{bi})$, $\mathcal{V}_2(t_{ai},t_{bi})$, $\mathcal{V}_1(t_{aj},t_{bj})$, $\mathcal{V}_2(t_{aj},t_{bj})$, $\mathcal{V}_4$, $\mathcal{V}_3(t_{ai},t_{bi})$, $\mathcal{V}_3(t_{aj},t_{bj})$, soit au cas où l'ordre est $\mathcal{V}_1(t_{aj},t_{bj})$, $\mathcal{V}_2(t_{aj},t_{bj})$, $\mathcal{V}_1(t_{ai},t_{bi})$, $\mathcal{V}_2(t_{ai},t_{bi})$, $\mathcal{V}_4$, $\mathcal{V}_3(t_{aj},t_{bj})$, $\mathcal{V}_3(t_{ai},t_{bi})$.

Ces trois équations couvrent tous les ordres possibles qu'on peut avoir et qui, en même temps, vérifient les précédences entre les sous-groupes. Comme l'emplacement du sous-ensemble $\mathcal{V}_2(t_{ai},t_{bi})$ (resp. $\mathcal{V}_2(t_{aj},t_{bj})$) n'influe pas sur la longueur d'ordonnancement des tâches sous L_1 (resp. L_2) alors on ne garde qu'une seule des différentes possibilités d'ordonnancement de ce sous-ensemble□

REMARQUE 3.4
On considère un système qui dispose de plusieurs latences en X et chaque latence est en relation X avec au plus une autre latence. Dans ce cas, les conditions du théorème 3.7 s'appliquent sur chaque paire de latences. Si les conditions sont satisfaites sur toutes les paires alors ce système est ordonnançable.

Désormais on dispose de conditions d'ordonnançabilité qui traitent tous les cas possibles dans le cas d'un seul processeur.

3.4.2 Cas multiprocesseur

D'après l'étude de complexité présentée dans la section 3.3.2.1 le problème d'ordonnancement multiprocesseur avec des contraintes de précédence et une **seule** contrainte de latence est NP-difficile. Le même problème avec **plusieurs** contraintes de latence est encore plus difficile que le précédent puisqu'il s'agit d'un cas plus général. On en déduit que le problème avec plusieurs latences est NP-difficile au sens fort.

De ce fait, on sait que des conditions multiprocesseur, vérifiables en un temps polynomial, sont impossibles à trouver. Afin de proposer un algorithme d'ordonnancement nous devons nous débrouiller, donc, avec les conditions monoprocesseur ainsi que les algorithmes introduits dans la section 3.3.2.2.

3.4.3 Algorithme glouton d'ordonnancement

Comme c'est le cas dans l'étude d'ordonnançabilité, l'algorithme d'ordonnancement prend en compte les différentes liaisons entre les paires de latences qui sont imposées au système de tâches. On a vu qu'en ce qui concerne les liaisons en || et en Z chaque latence peut être traitée séparément à la réserve de prendre en compte la précédence qui lie les deux latences dans le cas de liaison en Z. D'où l'obligation, dans ce cas, de traiter en premier lieu la latence de laquelle part la ou les précédences (voir la définition de la liaison en Z), qu'on peut désigner comme la latence prédécesseur, pour ensuite traiter la deuxième latence, qui peut être aussi désigne comme la latence successeur, et veiller à ce que les précédences entre les tâches concernées par ces deux latences soient respectées. En revanche, cette approche ne convient pas au cas de liaison en X puisque les deux latences ne peuvent pas être traitées séparément.

Effectivement dans le cas de paires de latence en X les algorithmes de construction de clusters et de l'union des clusters (7 et 8) ne sont plus valables, ce qui nous mène à proposer deux algorithmes spécifiques dans ce cas précis.

Le premier algorithme 10 est, comme dans les deux autres cas (en || et en Z), un algorithme de construction de clusters, sauf que la règle de la clusterisation change. Dans ce cas les tâches sont regroupées suivant les sous-groupes évoqués dans la section 3.4.1. Si la paire (L_1,L_2) est en X alors l'algorithme de construction de clusters est l'algorithme suivant :

Algorithme 10 Algorithme de construction de clusters dans le cas de latence en X

1: Regrouper dans un même cluster les tâches sous L_1 (resp. sous L_2) qui ont au moins un successeur dans l'ensemble des tâches concernées par L_2 (resp. L_1)
2: Regrouper dans un même cluster les tâches sous L_1 (resp. sous L_2) qui ont au moins un prédécesseur dans l'ensemble des tâches concernées par L_2 (resp. L_1)
3: Regrouper dans un même cluster les tâches sous L_1 (resp. sous L_2), qui n'ont ni prédécesseur ni successeur dans l'ensemble des tâches concernées par L_2 (resp. L_1) (si celles-ci existent)
4: Regrouper dans un même cluster les tâches communes aux deux ensembles concernés par L_1 et L_2 (si celles-ci existent)

Dans le cas de latence en X il peut y avoir au maximum sept clusters, et au minimum quatre, liés par des précédences. L'intérêt de la clusterisation, dans ce cas, est de s'assurer du respect de la contrainte de la précédence en ayant dans le même cluster des tâches qui partagent les mêmes prédécesseurs ainsi que les mêmes successeurs. Ainsi le deuxième algorithme 11, qu'on appelle algorithme d'ordonnancement dans le cas de contraintes de latence en X, commence par tester la condition monoprocesseur du théorème 3.7, et si celle-ci est satisfaite alors il ordonnance les tâches des clusters les uns après les autres sur le même processeur dans l'ordre établi par cette condition. Si, en revanche, la condition du théorème 3.7 n'est pas satisfaite alors l'algorithme ordonnance les tâches d'un cluster à la fois en veillant à ce que les tâches du premier cluster issu de la latence L_1 et celles de l'autre premier cluster issu de la latence L_2 ne soient pas ordonnancées sur le même processeur (ceci nous permet d'être sûr qu'on ordonnancera sur au moins deux processeurs). Le restant des tâches sont ordonnancées de façon à minimiser le temps d'ordonnancement entre la tâche source et la tâche destination de L_1 d'un côté et de L_2 de l'autre. Pour minimiser le temps d'ordonnancement, à chaque ordonnancement d'une tâche, on se sert d'une fonction de coût (comparable à celle définie à la section 6.1.4.2 dans la deuxième partie du manuscrit) qu'on calcule pour chaque processeur afin de déterminer l'impact, en termes de longueur sur le rallongement du temps d'ordonnancement, que provoque l'ordonnancement de cette tâche sur ce processeur. Si à la fin de l'ordonnancement de ces clusters les latences L_1 et L_2 ne sont pas satisfaites alors on dit que le système n'est pas ordonnançable avec cet algorithme. L'algorithme 11 détaille l'algorithme d'ordonnancement de tâches sous une contrainte de latence en X.

EXEMPLE 3.4
Pour illustrer l'algorithme d'ordonnancement 11 on propose de l'appliquer à un système avec deux contraintes de latence en X. Le graphe d'algorithme est le graphe donné par la figure 3.14 et l'architecture est composée de deux processeurs reliés par un médium de communication. La durée d'exécution, la même pour toutes les tâches, est égale à 1 et elle est égale aussi au coût d'une communication entre les processeurs. La valeur de la contrainte de latence entre les tâches t_a et t_e

Algorithme 11 Algorithme d'ordonnancement de tâches sous une contrainte de latence en X

1: Construire Ω l'ensemble des clusters en appliquant l'algorithme 10
2: **si** La condition du théorème 3.7 est satisfaite **alors**
3: Ordonnancer sur le même processeur les tâches des clusters les uns après les autres suivant l'ordre établi dans la condition du théorème 3.7
4: **sinon**
5: **pour** Chaque cluster Cl_j dans Ω pris dans l'ordre des précédences **faire**
6: **pour** Chaque tâche t_i dans Cl_j **faire**
7: En calculant la fonction de coût pour t_i sur tous les processeurs, choisir l'ordonnancement qui minimise le temps d'ordonnancement de la latence à laquelle t_i appartient
8: **fin pour**
9: **fin pour**
10: **si** L'un des deux temps d'ordonnancement relatifs aux deux latences est supérieur à la valeur de la contrainte de latence correspondante **alors**
11: Dire que ce système n'est pas ordonnançable avec cet algorithme
12: **fin si**
13: **fin si**

est de $L_1 = 8$ *et celle entre les tâches* t_h *et* t_l *est de* $L_2 = 8$.

- l'algorithme 11 commence par tester les conditions du théorème 3.7. On rappelle que dans la figure 3.14 les sous-ensembles obtenus après l'exécution de l'algorithme de culsterisation 10 ont été numérotés de s1 à s7. Par conséquent les sous-ensembles $\mathcal{V}_1(t_a,t_e)$, $\mathcal{V}_2(t_a,t_e)$, $\mathcal{V}_3(t_a,t_e)$, $\mathcal{V}_4$, $\mathcal{V}_1(t_h,t_l)$, $\mathcal{V}_2(t_h,t_l)$ et $\mathcal{V}_3(t_h,t_l)$ sont respectivement les sous-ensembles s1, s2, s3, s4, s5, s6 et s7 dans la figure 3.14.
  - la condition 3.10 est testée en premier :
    1. $\begin{cases} \sum_{t\in\mathcal{V}(t_a,t_e)} C(t) = 7 \leq (L_1 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_a,t_e)\cup\mathcal{V}(t_h,t_l)} C(t) = 12 \nleq (L_2 = 8) \end{cases}$
    2. $\begin{cases} \sum_{t\in\mathcal{V}(t_h,t_l)} C(t) = 7 \leq (L_2 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_h,t_l)\cup\mathcal{V}(t_a,t_e)} C(t) = 12 \nleq (L_1 = 8) \end{cases}$
  - la condition 3.11 est testée en second :
    1. $\begin{cases} \sum_{t\in\mathcal{V}(t_a,t_e)} C(t) + \sum_{t\in\mathcal{V}_3(t_h,t_l)} C(t) = 7+2 = 9 \nleq (L_1 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_a,t_e)\cup\mathcal{V}(t_h,t_l)} C(t) - (\sum_{t\in\mathcal{V}_3(t_a,t_e)} C(t) + \sum_{t\in\mathcal{V}_2(t_a,t_e)} C(t)) = 12-(2+1) = 9 \nleq (L_2 = 8) \end{cases}$
    2. $\begin{cases} \sum_{t\in\mathcal{V}(t_h,t_l)} C(t) + \sum_{t\in\mathcal{V}_3(t_a,t_e)} C(t) = 7+2 = 9 \nleq (L_2 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_h,t_l)\cup\mathcal{V}(t_a,t_e)} C(t) - (\sum_{t\in\mathcal{V}_3(t_h,t_l)} C(t) + \sum_{t\in\mathcal{V}_2(t_h,t_l)} C(t)) = 12-(2+1) = 9 \nleq (L_1 = 8) \end{cases}$

- la condition 3.12 est testée en dernier :

$$1.\begin{cases} \sum_{t\in\mathcal{V}(t_a,t_e)} C(t) + \sum_{t\in\mathcal{V}_1(t_h,t_l)} C(t) = 7+2 = 9 \not\leq (L_1 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_a,t_e)\cup\mathcal{V}(t_h,t_l)} C(t) - (\sum_{t\in\mathcal{V}_1(t_a,t_e)} C(t) + \sum_{t\in\mathcal{V}_2(t_a,t_e)} C(t)) = 12-(2+1) = 9 \not\leq (L_2 = 8) \end{cases}$$

$$2.\begin{cases} \sum_{t\in\mathcal{V}(t_h,t_l)} C(t) + \sum_{t\in\mathcal{V}_1(t_a,t_e)} C(t) = 7+2 = 9 \not\leq (L_2 = 8) \\ \\ \sum_{t\in\mathcal{V}(t_h,t_l)\cup\mathcal{V}(t_a,t_e)} C(t) - (\sum_{t\in\mathcal{V}_1(t_h,t_l)} C(t) + \sum_{t\in\mathcal{V}_2(t_h,t_l)} C(t)) = 12-(2+1) = 9 \not\leq (L_1 = 8) \end{cases}$$

– comme aucune des trois conditions n'a été satisfaite alors l'ordonnancement monoprocesseur qui satisfait en même temps les contraintes de latence L_1 et L_2 n'existe pas. Par conséquent, l'algorithme 11 tente de trouver un ordonnancement sur les deux processeurs qui respecte les contraintes de latence,
– les tâches du sous-ensemble s1 sont ordonnancées sur le processeur P_1 et les tâches du sous-ensemble s5 sont ordonnancées sur le processeur P_2,
– les tâches des sous-ensembles restants sont ordonnancés suivant l'ordre de précédence et chacune sur le processeur qui minimise le makspan (de la même manière que l'algorithme 16 dans la deuxième partie du manuscrit),
– l'ordonnacement résultant est exposé sur la figure 3.15. On constate que les contraintes de latence sont respectées puisque le temps d'ordonnancement entre les tâches t_a et t_e est de 8 et celui entre les tâches t_h et t_l est de 7. On en conclut que ce système est ordonnançable.

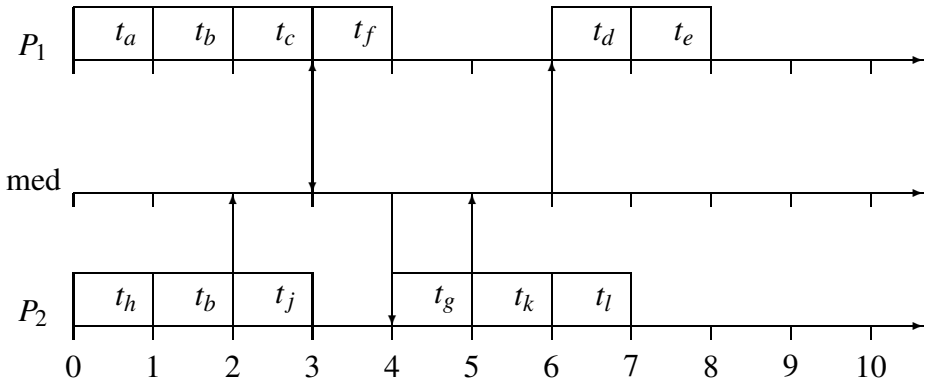

FIG. 3.15 – *(Exemple 3.4) Ordonnancement résultant*

Maintenant qu'on dispose d'algorithmes qui traitent tous les cas possibles, on est en mesure de présenter un algorithme d'ordonnancement de tâches avec des contraintes de précédence et de plusieurs contraintes de latence. De même que dans l'algorithme 9 (le cas d'une seule contrainte de latence), l'algorithme qu'on propose dans le cas de plusieurs latence considère que toutes les tâches concernées par la contrainte de latence constituent une super-tâche. Les tâches concernées par une paire de latence en || forment deux super-tâches indépendantes, et dans le cas Z deux super-tâches liées par une précédence. Par contre les paires de latence en X forment une seule super-tâches.

L'algorithme 12 détaille l'ordonnancement avec contraintes de précédence et plusieurs contraintes de latences.

3.5 Complexités des algorithmes

À l'instar du problème posé, les algorithmes proposés dépendent fortement de la forme du graphe de tâches en entrée. La complexité en temps de calcul (voir la section 1.5.1) de ces algorithme va dépendre du nombre de contraintes de latence qui sont imposées au graphe de tâches et surtout de la nature des liaisons entre les paires de latence. Cependant pour avoir une idée de la complexité des algorithmes proposés, on a calculé celle de l'algorithme traitant le cas d'une seule latence (Algorithme 9). Le résultat obtenu est le suivant :

- comme l'algorithme 9 est une extension de l'algorithme présenté à la section 6.1.4.2 dans la deuxième partie du manuscrit il a la même complexité que celui-ci, qui est de $O(n^2m)$, en plus de la complexité des instructions qu'on a rajoutées (n est le nombre de tâches du graphe d'algorithme en entrée et m est le nombre de processeurs de l'architecture),
- pour repérer toutes les tâches concernées par la latence L on effectue un parcours en profondeur. Si k est le nombre d'arcs dans le graphe des tâches alors la complexité de ce parcours est de $O(n+k)$;
- la complexité d'un algorithme de détection de tous les chemins dans graphe : nous prenons pour ce calcul la complexité la complexité d'une approche des équation linéaires qui est On^3 [153],
- l'algorithme de construction de clusters a une complexité de $O(n'+k'+p\ \log p)$ avec n' le nombre de tâches concernées par la contrainte de latence L, k' le nombre d'arcs reliant les n' tâches et p le nombre de chemins reliant la tâche source à la tâche destination,
- l'algorithme d'union a une complexité de $O(p)$.
- par conséquent la complexité globale de l'algorithme d'ordonnancement proposé est $O(n^3+n^2m+n+k+n'+k'+p(1+\log p))$.

En comparant cette complexité avec les complexités calculées dans le chapitre précédent, on s'aperçoit que dans ce cas le nombre de tâches du graphe et le nombre de processeurs de l'architecture ne suffisent plus à déterminer la complexité des algorithmes. Effectivement, ce résultat montre que la complexité d'un algorithme traitant la contrainte de latence est étroitement liée à des informations sur la forme du graphe comme le nombre d'arcs dans le graphe en entrée, le nombre d'arcs qui relient les tâches concernées par la contrainte de latence ainsi que le nombre de chemins qui relient la tâche source à la tâche destination.

Dans ce chapitre nous avons abordé le problème d'ordonnancement temps réel multiprocesseur avec contraintes de précédence et de latence. En utilisant les résultats de ce chapitre ainsi que ceux du chapitre précédent nous allons abordé, dans le prochain chapitre, le problème d'ordonnancement d'ordonnancement temps réel multiprocesseur avec contraintes de précédence, de périodicité stricte et de latence.

Algorithme 12 Algorithme glouton d'ordonnancement avec contraintes de précédence et plusieurs contraintes de latences

```
1: Repérer toutes les tâches concernées par la même latence et former une super-tâche
2: si La paire de latences est en X alors
3:     Regrouper les tâches concernées par les deux latences dans la même super-tâche
4: fin si
5: Initialisation de l'ensemble Δ des candidats avec les tâches sans prédécesseurs
6: tant que L'ensemble Δ n'est pas vide faire
7:     pour i allant de 1 au nombre de candidats faire
8:         Calcul de la fonction de coût pour chaque candidat sur tous les processeurs
9:         Choisir la fonction de coût minimale entre toutes celles calculées sur chacun des proces-
           seurs (le processeur en question est considéré comme le meilleur processeur pour cette
           tâche). Cela forme l'ensemble des paires (tâche, meilleur processeur)
10:    fin pour
11:    pour i allant de 1 au nombre de paires (tâche, meilleur processeur) faire
12:        Choisir la paire (tâche, meilleur processeur) qui a la fonction de coût maximale
13:    fin pour
14:    si La tâche choisie est la super tâche alors
15:        si Cette super tâche est composée des tâches sous deux latences en X alors
16:            Exécuter l'algorithme de construction de clusters dans le cas de latences en X (Algo-
               rithme 10)
17:            Exécuter l'algorithme d'ordonnancement de tâches sous latence en X (Algorithme 11)
18:        sinon
19:            Exécuter l'algorithme de construction de clusters (Algorithme 7)
20:            Exécuter l'algorithme d'union de clusters (Algorithme 8)
21:            si Le système est ordonnançable alors
22:                Ordonnancer les tâches du cluster principal sur le meilleur processeur de la super-
                   tâche et chaque cluster sur un processeur différent en respectant les dépendances
                   entre les tâches
23:            sinon
24:                Dire que ce système n'est pas ordonnançable avec cet algorithme
25:            fin si
26:        fin si
27:    sinon
28:        Ordonnancer cette tâche sur son meilleur processeur
29:    fin si
30:    Ajouter à l'ensemble Δ, des successeurs ordonnançables de ce candidat
31:    Suppression du candidat ordonnancé de l'ensemble Δ
32: fin tant que
```

Chapitre 4

Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence, de périodicité et de latence

Après un premier chapitre consacré à l'ordonnancement avec des contraintes de précédence et de périodicité puis un deuxième chapitre consacré à l'ordonnancement avec des contraintes de précédence et de latence, nous sommes en mesure de considérer l'ordonnancement avec des contraintes de précédence, de périodicité et de latence . Les résultats accumulés durant les deux précédents chapitres nous permettent de proposer des algorithmes d'ordonnancement avec les trois contraintes.

Ce chapitre est organisé de la manière suivante : la première section s'intéresse, comme on l'a fait pour chaque contrainte à part, aux effets de la présence des deux contraintes (latence et périodicité) sur l'exécution des tâches. Ensuite la deuxième section expose une étude d'ordonnançabilité monoprocesseur dont les résultats serons utilisés par l'algorithme proposé. Pour finir, la dernière section introduit l'algorithme glouton qu'on propose pour ordonnancer de tels systèmes. Notons que la contrainte de précédence, même si elle n'est pas mentionnée à chaque fois qu'on évoque les contraintes, est évidemment prise en compte. Nous réutilisons dans ce chapitre les modèles et les notations employés dans les deux chapitres précédents.

4.1 Nature de l'exécution des tâches avec contraintes de périodicité et de latence

On a vu dans les deux chapitres précédents que dans le modèle d'un graphe d'algorithme avec la contrainte de périodicité, chaque tâche est caractérisée par une période alors que dans le modèle d'un graphe d'algorithme avec une ou plusieurs contraintes de latence il se peut que certaines tâches ne soient pas concernées par ces contraintes. Par conséquent, dans un système soumis à ces

deux types de contraintes, on est amené à traiter deux types de tâches :

1. des tâches caractérisées par une période en plus d'une durée d'exécution,
2. des tâches caractérisées par une période, une durée d'exécution et concernées par une contrainte de latence.

Le premier type de tâches a été largement étudié dans le chapitre 2. Par contre l'étude du deuxième type n'a pas encore été abordée dans ce manuscrit.

On sait que les tâches concernées par une contrainte de latence appartiennent à des chemins qui relient la tâche source à la tâche destination. Dès lors une tâche concernée par cette latence a comme prédécesseur la tâche source de la latence, et d'après ce qui est expliqué dans la section 2.1.1.2, à propos des dépendances entre tâches périodiques, il est impératif que la période de toute tâche concernée par la latence soit égale ou multiple de la période de la tâche source. Par conséquent il n'y a que les tâches avec les mêmes ou des périodes multiples qui peuvent être concernées par la même contrainte de latence.

Pour étudier des tâches périodiques dépendantes sous une contrainte de latence on propose d'observer un cas simple. Prenons l'exemple d'un graphe de 3 tâches périodiques et une contrainte de latence qu'on impose sur deux d'entre elles (Figure 4.1). Si t_a, t_b et t_c sont, respectivement, des tâches de périodes 2, 4 et 8 alors on sait que pendant l'exécution, vu que l'hyper-période est égale à 8, la tâche t_a se répète 4 fois, la tâche t_b se répète 2 fois et la tâche t_c une seule fois. D'après la définition de la contrainte de latence (section 3.1), le fait d'imposer une latence $L(t_a,t_b)$ entre les tâches t_a et t_b signifie que le temps qui s'écoule entre la production des données par t_a et la consommation de celles-ci par t_b doit être inférieur à $L(t_a,t_b)$. En déroulant ce graphe on s'aperçoit que la contrainte de latence $L(t_a,t_b)$ se multiplie à son tour (puisque les tâches se multiplient) pour devenir :

– une première contrainte de latence $L(t_{a_1},t_{b_1})$ (égale à $L(t_a,t_b)$) qui impose que le temps qui s'écoule entre la production des données par t_{a_1} et la consommation de celles-ci par t_{b_1} doit être inférieur à $L(t_{a_1},t_{b_1})$,
– une deuxième contrainte de latence $L(t_{a_3},t_{b_2})$ (égale à $L(t_a,t_b)$) qui impose que le temps qui s'écoule entre la production des données par t_{a_3} et la consommation de celles-ci par t_{b_2} doit être inférieur à $L(t_{a_3},t_{b_2})$,

Cela s'explique par le fait que l'instance t_{b_1} dépend de t_{a_1}, et t_{a_2} et l'instance t_{b_2} dépend de t_{a_3} et t_{a_4}.

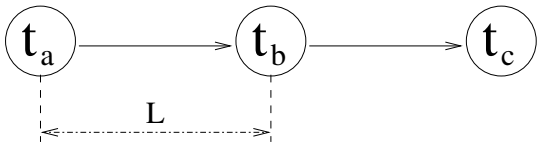

FIG. 4.1 – *Graphe d'algorithme avec les trois contraintes*

On en déduit que la latence se répète ou se propage entre les instances des tâches concernées par la latence. Dans la suite nous proposons un algorithme de déroulement qui prend en considération la répétition des contraintes de latence.

4.2 Étude d'ordonnançabilité

On n'a pas besoin de recourir à de lourdes démonstrations pour affirmer que ce problème d'ordonnancement est un problème NP-difficile au sens fort vu qu'il s'agit du cas plus général que des problèmes qui sont déjà NP-difficiles au sens fort. Par conséquent une condition nécessaire et suffisante vérifiable en un temps polynomial est impossible à trouver. Cependant, et comme on l'a fait dans les deux précédents chapitres, on étudie le cas monoprocesseur qui est considérablement plus simple à traiter. Les résultats obtenus nous servirons à proposer un algorithme d'ordonnancement multiprocesseur. Nous supposons, dans cette étude, que les conditions d'ordonnançabilité prenant en compte la période seulement (introduites dans le chapitre 2) sont satisfaites.

Tout d'abord, nous nous intéressons au cas basique de deux tâches dépendantes et périodiques entre lesquelles on impose une contrainte de latence. Le théorème suivant introduit une condition d'ordonnançabilité dans ce cas.

THÉORÈME 4.1
Soient t_a et t_b deux tâches dépendantes et périodiques telles que $T(t_b) = kT(t_a)$. Si une contrainte de latence $L(t_a,t_b)$ est définie entre les deux tâches t_a et t_b alors la condition nécessaire et suffisante d'ordonnançabilité est la suivante :

$$(k-1)T(t_a)+C(t_a)+C(t_b) \leq L(t_a,t_b) \tag{4.1}$$

Preuve
D'après la condition 3.2 on sait que le fait que le temps global d'ordonnancement soit inférieur ou égal à la valeur de la latence est une condition nécessaire et suffisante d'ordonnançabilité. Dès lors, pour prouver le théorème 4.1 il suffit de montrer que $(k-1)T(t_a)+C(t_a)+C(t_b)$ représente le temps d'ordonnancement entre les tâches t_a et t_b.
On sait que pour être exécutée, la tâche t_b doit disposer de toutes les données produites par les k instances de t_a. Comme les dates de début d'exécution de deux instances successives sont décalées de $T(t_a)$ unités de temps, on a besoin de $(k-1)T(t_a)$ unités de temps pour que la dernière instance de t_a soit exécutée et par la même occasion toutes les données nécessaires à l'exécution de t_b soient disponibles. La tâche t_b peut, donc, être exécutée tout de suite après. On en déduit que, pour que ce système soit ordonnançable, la valeur minimale que $L(t_a,t_b)$ peut avoir est $(k-1)T(t_a)+C(t_a)+C(t_b)\boxdot$

On passe maintenant au cas, plus général, de n tâches $\{t_1,t_2,...,t_n\}$ concernées par la contrainte de latence $L(t_1,t_n)$. Toutes les périodes des n tâches sont des multiples et on suppose que $T(t_1) \leq T(t_2) \leq ... \leq T(t_n)$. Dans ce cas, comme pour le cas précédent la condition nécessaire et suffisante d'ordonnançabilité est que le temps d'ordonnancement entre la date de début d'exécution de la première instance de t_1 et la date de fin d'exécution de la tâche t_n doit être inférieur ou égale à $L(t_1,t_n)$. Nous proposons dans la suite une méthode pour calculer ce temps d'ordonnancement qu'on désigne par Φ_{t_1,t_n}.

La méthode, qu'on propose, calcule Φ_{t_1,t_n} en partant de la valeur $[(\frac{T(t_n)}{T(t_1)}-1)T(t_1)+C(t_1)]$ qui correspond au temps d'ordonnancement des $(\frac{T(t_n)}{T(t_1)})$ premières instances de t_1. La valeur de Φ_{t_1,t_n}

est ensuite calculée récursivement en commençant par rajouter les durées d'exécution de toutes les tâches hormis celle de t_1 qui a déjà été prise en compte. Ensuite, en fonction de la valeur de Φ_{t_1,t_n}, on rajoute les durées d'exécution des instances des tâches t_1, ..., t_{n-1} qui s'exécutent avant la tâche t_n (comme l'hyper-période de ce système est forcément égale à la période de la tâche t_n donc il n'y a certainement qu'une seule instance de cette tâche). Même si t_n ne dépend pas de toutes les instances qui s'exécutent avant elle, ces instances doivent absolument s'exécuter pour que leurs périodes soient respectées. Ceci peut entraîner le retardement de la date de début d'exécution de t_n (plus de détails sont dans l'exemple 4.1). L'algorithme 13 détaille l'algorithme de calcul de Φ.

Algorithme 13 Algorithme de calcul de Φ

1: Numéroter les n tâches de 1 à n dans le sens des dépendances
2: $\Phi^0_{t_1,t_n} \leftarrow \sum_{i=2}^{i=n} C(t_i)$
3: **pour** i de 1 à $n-1$ **faire**
4: $\quad \delta^1_i = \left\lfloor \frac{\Phi^0_{t_1,t_n}}{T(t_i)} \right\rfloor$
5: **fin pour**
6: $\Phi^1_{t_1,t_n} \leftarrow \Phi^0_{t_1,t_n} + \sum_{i=1} \delta^1_i C(t_i)$
7: $j \leftarrow 0$
8: **tant que** $\Phi^{j+1}_{t_1,t_n} \neq \Phi^j_{t_1,t_n}$ **faire**
9: $\quad$ **pour** i de 1 à $n-1$ **faire**
10: $\quad\quad \delta^{j+1}_i = \left\lfloor \frac{\Phi^j_{t_1,t_n}}{T(t_i)} \right\rfloor$
11: $\quad$ **fin pour**
12: $\quad \Phi^{j+1}_{t_1,t_n} \leftarrow \Phi^j_{t_1,t_n} + \sum_{i=1} (\delta^{j+1}_i - \delta^j_i) C(t_i)$
13: $\quad j \leftarrow j+1$
14: **fin tant que**
15: $\Phi \leftarrow (\frac{T(t_n)}{T(t_1)} - 1) T(t_1) + C(t_1) + \Phi_{t_1,t_n}$

EXEMPLE 4.1
Pour expliquer l'algorithme 13, nous proposons de traiter les deux exemples suivants.

Dans le premier exemple on calcule ϕ pour cinq tâches, concernées par la contrainte de latence L, qu'on numérote de 1 à 5. Ces cinq tâches sont définies comme suit : $(t_1 : 1{,}4)$, $(t_2 : 1{,}8)$, $(t_3 : 1{,}8)$, $(t_4 : 1{,}16)$ et $(t_5 : 1{,}16)$.

- $\Phi^0_{t_1,t_n} = \sum_{i=2}^{i=5} C(t_i) = 1+1+1+1 = 4$,
- $\delta^1_1 = \lfloor \frac{4}{4} \rfloor = 1$, $\delta^1_2 = \delta^1_3 = \lfloor \frac{4}{8} \rfloor = 0$ et $\delta^1_4 = \lfloor \frac{4}{16} \rfloor = 0$,
- $\Phi^1_{t_1,t_n} = \Phi^0_{t_1,t_n} + (1-0)*1 + 0*1 + 0*1 + 0*1 + 0*1 = 4+1 = 5$
- $\delta^2_1 = \lfloor \frac{5}{4} \rfloor = 1$, $\delta^2_2 = \delta^2_3 = \lfloor \frac{5}{8} \rfloor = 0$ et $\delta^2_4 = \lfloor \frac{5}{16} \rfloor = 0$,

– comme les δ_i^2 sont égaux aux δ_i^1 alors $\Phi_{t_1,t_n}^2 = \Phi_{t_1,t_n}^1$
– pour finir $\Phi = (\frac{T(t_n)}{T(t_1)} - 1)T(t_1) + C(t_1) + \Phi_{t_1,t_n}^2 = (\frac{16}{4} - 1)4 + 1 + 5 = 18$
– on en déduit que la valeur de L doit être supérieure ou égale à 18 pour que ce système de tâches soit ordonnançable. Ce résultat est vérifiable sur l'ordonnancement de ces cinq tâches sur la figure 4.2.

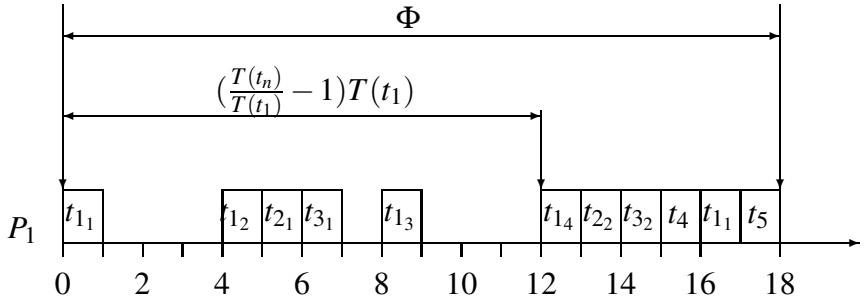

FIG. 4.2 – *(Exemple 4.1-1) Calcul de* Φ

Dans le deuxième exemple on dispose de cinq tâches concernées par la contrainte de latence L et qui sont définies comme suit : $(t_1 : 2,4)$, $(t_2 : 1,8)$, $(t_3 : 1,8)$, $(t_4 : 1,16)$ et $(t_5 : 1,16)$.

– $\Phi_{t_1,t_n}^0 = \sum_{i=2}^{i=5} C(t_i) = 1 + 1 + 1 + 1 = 4$,
– $\delta_1^1 = \lfloor \frac{4}{4} \rfloor = 1$, $\delta_2^1 = \delta_3^1 = \lfloor \frac{4}{8} \rfloor = 0$ et $\delta_4^1 = \lfloor \frac{4}{16} \rfloor = 0$,
– $\Phi_{t_1,t_n}^1 = \Phi_{t_1,t_n}^0 + (1-0) * 2 + 0 * 1 + 0 * 1 + 0 * 1 + 0 * 1 = 4 + 2 = 6$
– $\delta_1^2 = \lfloor \frac{6}{4} \rfloor = 1$, $\delta_2^2 = \delta_3^2 = \lfloor \frac{6}{8} \rfloor = 0$ et $\delta_4^2 = \lfloor \frac{6}{16} \rfloor = 0$,
– comme les δ_i^2 sont égaux aux δ_i^1 alors $\Phi_{t_1,t_n}^2 = \Phi_{t_1,t_n}^1$
– pour finir $\Phi = (\frac{T(t_n)}{T(t_1)} - 1)T(t_1) + C(t_1) + \Phi_{t_1,t_n}^2 = (\frac{16}{4} - 1)4 + 2 + 6 = 20$
– ce résultat est vérifiable sur l'ordonnancement de ces cinq tâches sur la figure 4.3
– on en déduit que la valeur de L doit être supérieure ou égale à 20 pour que ce système soit ordonnançable.

REMARQUE 4.1
En comparant la condition 3.2 et le calcul Φ *on se rend compte que l'étude d'ordonnançabilité dans ce cas est plus compliquée à faire que celle du chapitre 3 même pour un cas simple comme celui du monoprocesseur.*

Ce qu'on peut déduire de cette étude c'est que pour qu'un système de tâches avec des contraintes de précédence, de périodicité et de latence soit ordonnançable il est nécessaire que, d'un côté, ce système soit, déjà, ordonnançable relativement à la contrainte de la périodicité, et qu'il le soit aussi relativement à la contrainte de latence. On peut dire que ces deux conditions sont des conditions nécessaires dans le cas de ce chapitre. Pour en être sûr, il suffit de prendre un cas simple de deux

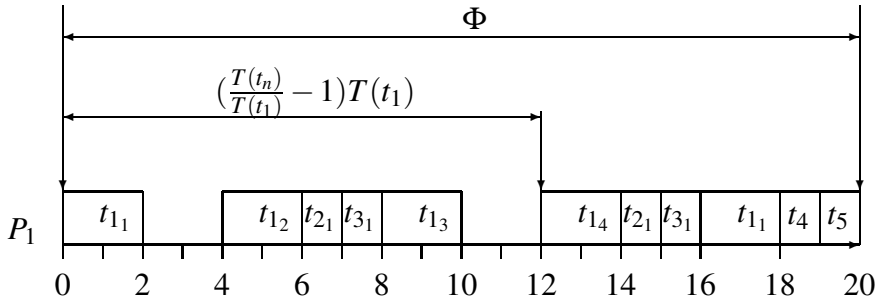

FIG. 4.3 – *(Exemple 4.1-2) Calcul de* Φ

tâches dépendantes et périodiques $(t_a : 1,2)$ et $(t_b : 1,4)$ et un processeur P. La contrainte de latence qui relie ces deux tâches est $L(t_a,t_b) = 2$. Il est facile de vérifier que ce système est ordonnançable en termes de latence d'un côté et en termes de périodicité de l'autre. Par contre si on teste la condition du théorème 4.1 on trouve que ce système n'est plus ordonnançable dès que les contraintes sont réunies.

Afin de réutiliser les résultats et les algorithmes proposés dans les deux chapitres précédents, on a cherché un moyen de se ramener du problème traité dans ce chapitre aux problèmes qu'on a déjà résolus. Les principes ainsi que les algorithmes qui nous le permettent sont présentés dans la prochaine section.

4.3 Heuristique d'ordonnancement multiprocesseur de systèmes de tâches avec des contraintes de précédence, de périodicité et de latence

Dans cette section nous reprenons les résultats obtenus dans les deux chapitres précédents afin de proposer un algorithme d'ordonnancement de systèmes de tâches avec des contraintes de précédence, de périodicité et de latence. Pour commencer, comme nous ne disposons pas de conditions d'ordonnançabilité prenant en compte toutes les contraintes, on commence par analyser l'ordonnançabilité des tâches conformément à la contrainte de périodicité seulement. Cette étape est donnée dans la section 2.2.1 où deux approches sont disponibles. Le résultat de cette étape nous permet, s'il est négatif, de savoir que c'est la périodicité qui cause la non-ordonnançabilité. Ensuite, et une fois que le système avec la contrainte de la périodicité est prouvé ordonnançable, on cherche un moyen permettant de se dégager de la contrainte de périodicité pour pouvoir traiter uniquement les contraintes de latence. En effet, dans la section 2.2.2, on a vu que le fait de dérouler le graphe d'algorithme permet de le transformer en un autre graphe dans lequel toutes les tâches sont à la même période qui est l'hyper-période. Cette manoeuvre se révèle être très utile puisqu'elle nous offre la possibilité de réutiliser les résultats du chapitre 3. Cependant, l'algorithme de déroulement en présence des contraintes de latence n'est pas le même que celui donnée par l'algorithme 3. Le nouvel algorithme de déroulement doit, en plus de ce que réalise son prédécesseur, répéter

les contraintes de latences entre les instances des tâches soumises à la même contrainte de latence dans le graphe de départ (comme l'explique le dernier paragraphe de la section 4.1). L'algorithme 14 présente la nouvelle méthode de déroulement.

Algorithme 14 Algorithme de déroulement dans le cas des contraintes de latence et de précédence

1: Répéter chaque tâche suivant la valeur de sa période et celle de l'hyper-période
2: Ajouter une précédence entre chaque paire d'instances successives de la même tâche
3: Ajouter les dépendances entre les instances des tâches productrices et celles des tâches consommatrices de la même manière que dans 2.2.2
4: **pour** Toutes les latences $L(t_i,t_j)$ **faire**
5: Créer $(\frac{HP}{T(t_j)} - 1)$ contraintes de latences
6: **pour** k = 1 à $(\frac{HP}{T(t_j)} - 1)$ **faire**
7: Imposer une contrainte de latence entre la $(k\frac{T(t_j)}{T(t_i)})$ ème instance de t_i et la k ème instance de t_j
8: **fin pour**
9: **fin pour**

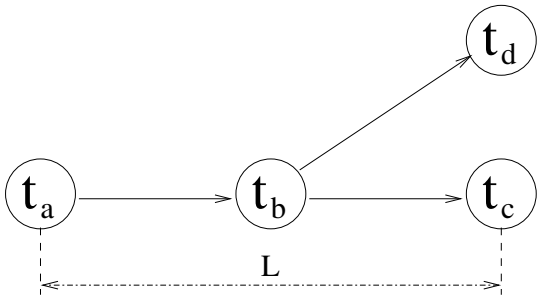

FIG. 4.4 – *(Exemple 4.2) Graphe d'algorithme*

EXEMPLE 4.2
Pour illustrer l'algorithme 14 nous proposons de l'appliquer au graphe de la figure 4.4. Le résultat est décrit par la figure 4.5.

Une fois que l'algorithme de déroulement est exécuté, on est en mesure de reprendre l'algorithme d'ordonnancement 12 introduit dans le chapitre 3. En plus, pour que les dates de début

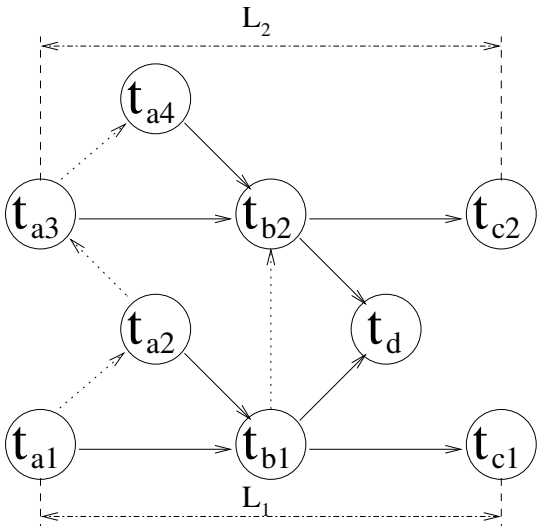

FIG. 4.5 – *(Exemple 4.2) Graphe d'algorithme après déroulement*

d'exécution des instances des tâches respectent leurs périodes on rajoute dans l'algorithme d'ordonnancement 12 une instruction qui vise à vérifier la condition du théorème 2.10 (voir la section 2.2.3).

L'algorithme 15 introduit l'heuristique d'ordonnancement multiprocesseur de systèmes de tâches avec des contraintes de précédence, de périodicité et de latence. Il reprend toutes les étapes citées avant et détaille les évolutions par rapport aux algorithmes présentés dans les deux précédents chapitres.

Algorithme 15 Heuristique d'ordonnancement multiprocesseur de systèmes de tâches avec des contraintes de précédence, de périodicité et de latence

Suivant le résultat que l'on veut obtenir on exécute soit l'algorithme 2, soit l'algorithme 1
Exécuter l'algorithme 14
Repérer toutes les tâches concernées par la même latence et former une super-tâche
si La paire de latences est en X **alors**
    Regrouper les tâches concernées par les deux latences dans la même super-tâche
fin si
Initialisation de l'ensemble Δ des candidats avec les tâches sans prédécesseurs
tant que L'ensemble Δ n'est pas vide **faire**
    pour *i* allant de 1 au nombre de candidats **faire**
        Calcul de la fonction de coût pour chaque candidat sur tous les processeurs où la condition du théorème 2.10 est satisfaite
        Remettre ce candidat dans Δ si aucun processeur ne vérifie la condition du théorème 2.10
        Choisir la fonction de coût minimale entre toutes celles calculées sur chacun des processeurs (le processeur en question est considéré comme le meilleur processeur pour cette tâche). Cela forme l'ensemble des paires (tâche, meilleur processeur)
    fin pour
    pour *i* allant de 1 au nombre de paires (tâche, meilleur processeur) **faire**
        Choisir la paire (tâche, meilleur processeur) qui a la fonction de coût maximale
    fin pour
    si La tâche choisie est la super tâche **alors**
        si Cette super tâche est composée des tâches sous deux latences en X **alors**
            Exécuter l'algorithme de construction de clusters dans le cas de latences en X (Algorithme 10)
            Exécuter l'algorithme d'ordonnancement de tâches sous latence en X (Algorithme 11) en respectant les dépendances entre les tâches et les dates de début d'exécution des instances (voir la section 2.2.3)
        sinon
            Exécuter l'algorithme de construction de clusters (Algorithme 7)
            Exécuter l'algorithme d'union de clusters (Algorithme 8)
            si Le système est ordonnançable **alors**
                Ordonnancer les tâches du cluster principal sur le meilleur processeur de la super-tâche et chaque cluster sur un processeur différent en respectant les dépendances entre les tâches et les dates de début d'exécution des instances (voir la section 2.2.3)
            sinon
                Dire que ce système n'est pas ordonnançable avec cet algorithme
            fin si
        fin si
    sinon
        Ordonnancer cette tâche sur son meilleur processeur en respectant sa date de début d'exécution si il s'agit d'une autre instance que la première (voir la section 2.2.3)
    fin si
    Ajouter à l'ensemble Δ, des successeurs ordonnançables de ce candidat
    Suppression du candidat ordonnancé de l'ensemble Δ
fin tant que

Conclusion

Dans la première partie du manuscrit nous avons présenté les résultats théoriques en analyse d'ordonnancement et en ordonnancement multiprocesseur hors ligne. Les contraintes prises en compte sont la précédence, la périodicité stricte et la latence.

L'état de l'art effectué a montré, d'une part, que le problème d'ordonnancement tel que nous le considérons est peu traité et, d'autre part, que les méthodes de résolution les plus appropriées sont les méthodes approchées. Pour des raisons de prototypage rapide nous avons choisi la méthode gloutonne qui est la plus rapide parmi toutes les méthodes approchées.

Le problème d'ordonnancement multiprocesseur avec contraintes de précédence, de périodicité et de latence a été traité en plusieurs étapes :

Tout d'abord nous avons pris en compte les contraintes de précédence et de périodicité stricte seulement. En se basant sur une étude d'ordonnançabilité que nous avons fait, nous avons proposé un algorithme glouton qui s'est révélé très rapide mais qui n'était pas efficace en terme de fiabilité dans l'ordonnançabilité. Pour y remédier nous avons proposé une heuristique du type "Recherche Locale" qui a des résultats en terme de fiabilité dans l'ordonnançabilité très intéressants tout en ayant un temps d'exécution rapide. En raison de l'absence de benchmarks sur ce problème on a développé un algorithme exact du type "Branch and Cut" afin d'évaluer les deux précédents algorithmes. On s'est également intéressé à la bonne gestion des ressources que sont la mémoire et la consommation qui est liée au temps d'exécution du système (Makespan). Nous avons proposé un algorithme glouton d'équilibrage de charge et de mémoire. Cet algorithme a été prouvé $(2-\frac{1}{M}$-approché en équilibrage de mémoire (M est le nombre de processeurs de l'architecture) ;

Ensuite nous avons pris en compte les contraintes de précédence et une contrainte de latence seulement. l'étude d'ordonnançabilité effectuée a montré que la contrainte de latence est très liée à la forme du sous-graphe d'algorithme contenant les tâches soumises à cette contrainte. Dès lors l'algorithme glouton d'ordonnancement proposé fait appel à un algorithme de clusterisation et un algorithme d'union. Nous abordons, par la suite, le cas où les contraintes de latence sont multiples en proposant une extension de l'algorithme proposé auparavant.

Finalement nous avons pris en compte les trois contraintes à la fois. Après avoir étudié les tâches soumises simultanément à la contrainte de périodicité et celle de la latence nous avons proposé un algorithme glouton d'ordonnancement. Cet algorithme est une synthèse des algorithmes proposés dans les deux précédentes étapes.

Deuxième partie

Développements logiciels

Introduction

La deuxième partie du manuscrit expose les travaux de développement logiciel effectués dans le logiciel SynDEx en utilisant les résultats théoriques obtenus dans la première partie afin que SynDEx puisse traiter des systèmes de tâches avec contraintes de précédence et de périodicité alors qu'auparavant il ne traitait que des systèmes avec précédence et une seule période égale au temps d'exécution de toutes les tâches. Afin de valider ses performances le nouveau logiciel a été testé sur une application multipériode de suivi pour train virtuel de CyCabs.

Cette partie est organisée de la manière suivante : le premier chapitre commence par définir ce qu'est un logiciel d'aide au développement d'applications temps réel embarqué, puis un état de l'art des logiciels existants est proposé. Le deuxième chapitre relate les modifications apportées à l'ancienne version du logiciel SynDEx ainsi que les caractéristiques de la nouvelle version. Pour finir le troisième et dernier chapitre décrit l'application de suivi pour train virtuel de CyCabs.

Chapitre 5

État de l'art

SynDEx est un logiciel graphique interactif d'aide au développement d'applications temps réel embarqué. Il assure que les spécifications de l'algorithme, de l'architecture (plate-forme) et des durées des opérations sur les différents "" conduisent à une implantation sur une machine multiprocesseur minimisant le temps d'exécution total. À partir d'une implantation satisfaisante, il génère automatiquement un exécutif correcte par construction, c'est-à-dire sans interblocage, libérant ainsi l'utilisateur des tâches lourdes de programmation bas niveau (système).

La principale contribution de cette thèse relativement au logiciel SynDEx, concerne la possibilité de traiter des applications temps réel contenant des tâches pouvant avoir différentes périodes. En se basant sur les résultats obtenus dans la première partie de la thèse une nouvelle version du logiciel a été développée.

Cette mutation permet à l'outil SynDEx de figurer sur deux domaines complémentaires mais qui sont dans la plupart du temps traitées séparément. Ces deux domaines sont : i) prototypage rapide d'applications temps réel sur des architectures distribuées hétérogènes (parallèles, multiprocesseur), ii) analyse et ordonnancement temps réel des systèmes de tâches non-préemptives avec des contraintes de dépendances et de périodicité.

L'état de l'art qui suit va s'intéresser aux deux aspects en citant les principaux travaux et outils existants avant de présenter en détails le logiciel SynDEx ainsi que les nouvelles caractéristiques.

5.1 Introduction

La réalisation d'un système temps réel nécessite une bonne maîtrise des outils fournis par la théorie de l'automatique lors de la phase de modélisation et de simulation, ainsi qu'une bonne maîtrise de l'informatique temps réel lors de la phase d'implantation.

La recherche sur la programmation des systèmes temps réel tente de fournir des méthodes de développement permettant de gérer au mieux cette grande complexité et d'aboutir à la réalisation de systèmes sûrs et efficaces. C'est dans ce but qu'ont été développées les méthodes formelles. Ces méthodes cherchent à définir et à regrouper des outils de spécification (description du modèle), des outils de vérification permettant de simuler et de vérifier les propriétés de la spécification ainsi que des outils de génération automatique des programmes temps réel implantant la spécification.

Réunir sous un même environnement de développement, un langage de spécifications, des outils de vérification et de génération automatique de programmes temps réel capables de prendre en compte différents types d'architectures (notamment multiprocesseur) et capables de garantir que le code généré est conforme à la spécification, constituerait l'outil de conception idéal capable d'aboutir rapidement à la réalisation d'un système sûr et optimisé. Réaliser un tel outil est l'un des challenges actuels de la recherche sur la programmation des systèmes temps réel.

5.1.1 Qu'est-ce que le prototypage ?

La complexité des applications visées, au niveau des algorithmes, de l'architecture matérielle, et des interactions avec l'environnement sous contraintes temps réel, nécessite des méthodes pour minimiser la durée du cycle de développement, depuis la conception jusqu'à la mise au point des prototypes ainsi que des "produits de série" obtenus à partir de ces prototypes, s'exécutant dans les deux cas en temps réel. Afin d'éviter toute rupture entre les différentes phases du cycle de développement et pour permettre des vérifications formelles et des optimisations, l'emploi du prototypage est essentiel. Il est présenté par Kordon et Luqui [154] comme une approche de développement, intégrant la spécification, la modélisation, l'évaluation, le raffinement et la génération automatique des prototypes.

Le prototypage est une procédure qui part de la spécification des traitements d'une application pour aboutir à la production d'une ou plusieurs architectures prototypes à partir desquelles le choix d'un produit final va être facilité. Cette démarche d'implémentation dépend de plusieurs éléments :

- la plate-forme d'implantation,
- le modèle choisi pour décrire l'application,
- le flot d'implémentation,
- les outils de compilation et de synthèse disponibles.

On dit qu'un prototypage est rapide quand il est réalisé dans un temps très court. Il vise à atteindre trois objectifs principaux :

1. réduction du temps de cycle de développement : par une approche systématique, voire un processus automatique, les différentes phases du développement devront être fortement accélérées;
2. sécurisation du développement : un processus automatique et sûr, au sens de la validité des résultats produits,
3. exploration de différentes solutions d'implantation : l'espace d'exploration des solutions sera d'autant plus large que la description d'entrée sera fournie à un haut niveau d'abstraction.

La méthode de prototypage est d'autant plus bénéfique qu'elle permet une description de l'implantation à un haut niveau d'abstraction, tout en aboutissant à une solution optimisée de la façon la plus automatique possible.

5.1.1.1 Plate-forme d'implantation

On appelle plate-forme le dispositif matériel que l'on peut configurer pour exécuter une application (architecture est le terme utilisé dans la première partie de la thèse). Ces plates-formes comprennent des éléments de calcul (processeurs programmables ou spécialisés, circuits configurables et/ou reconfigurables, etc.), des mémoires, des dispositifs d'entrée/sortie et des media de communication connectant ces divers composants.

Une plate-forme matérielle est une architecture qui peut être à la base de plusieurs produits dans un même domaine d'application. Alors qu'une plate-forme logicielle repose sur la séparation des fonctions d'un côté et sur des interfaces standard bien définies de l'autre. En outre il existe des plates-formes matérielle/logicielle qui combinent les deux propriétés. D'où l'approche de conception conjointe matériel/logiciel plus communément appelée "co-design". L'objectif de cette approche est de produire un système matériel/logiciel qui répond aux spécifications données, en respectant les différentes contraintes.

Quatre modèles de plates-formes sont utilisées pour le prototypage : les architectures multiprocesseur, l'assemblage d'IP (Intellectual Property) logiciels et/ou matériels [155], les architectures hétérogènes et les architectures reconfigurables [156, 157].

5.1.1.2 Modèle

La modélisation d'une application est la description des traitements qu'elle est sensée exécuter. Le choix d'un modèle dépend des objectifs à atteindre, comme par exemple :

- implémentation logicielle ou matérielle de l'application,
- critères à optimiser,
- vitesse, consommation, surface, etc.,
- exploitation du parallélisme de l'application.

Parmi les modèles qui existent on peut citer : le modèle flot de données (comme AAA par exemple qui sera détaillé dans la suite), les réseaux de processus de Kahn [158], les réseaux de Pétri [159], le modèle polyédrique [160], etc. Tous ces modèles ont un objectif commun : séparer l'expression des traitements de leur implémentation et exposer le parallélisme inhérents à ces traitements.

5.1.1.3 Flot d'implémentation

Un flot d'implémentation, encore appelé méthodologie d'implémentation, est une démarche de développement et de raffinement du modèle décrivant l'application en vue d'en déduire du code exécutable par la plate-forme. Il s'agit de sélectionner et d'exploiter des outils logiciels pour décrire, compiler et implanter les applications. Le flot permet de passer d'une représentation schématique d'un traitement à une représentation logicielle ou matérielle de ce traitement.

Le flot est fortement dépendant de la plate-forme cible de simulation et du type d'outils logiciels disponibles pour cette plate-forme. Il dépend aussi du type et de la structure de l'application

ainsi que de sa modélisation. Selon qu'on cible une plate-forme logicielle, matérielle ou logicielle/matérielle, le flot d'implémentation peut partir aussi bien d'une implémentation faite avec des langages impératifs tels que C ou Matlab que des langages matériels tels que VHDL ou Verilog. Dans le cas où les implémentations sont faites en langage matériel, de nombreux chercheurs soulignent que la tâche est plus complexe, fastidieuse et le taux d'erreurs est très élevé [161, 162]. À l'inverse, l'utilisation de langages de haut niveau rend difficile l'obtention de plate-formes très performantes en vitesse, surface ou consommation [163]. Il existent un flot d'implémentation pour chacun des modèles vus dans la section précédente.

5.2 Etat de l'art des outils pour la conception électronique au niveau système

Récemment apparu, le terme ESL (Electronic System Level pour la conception électronique au niveau système) regroupe les différentes initiatives cherchant à formaliser et automatiser le plus tôt possible la spécification et la conception de systèmes en vue de réaliser du prototypage rapide. Car en ce qui concerne la conception des systèmes électroniques, le niveau RTL (Register Transfert Level) est jusque-là, le niveau standard dans l'industrie. Dans le souci d'élever le niveau d'abstraction pour permettre une simulation conjointe (logiciel et matériel) plus rapide, un autre niveau d'abstraction ESL est apparu . Il s'agit d'un ensemble de niveaux d'abstractions appelé TLM (pour Transaction Level Modeling). À ce niveau, les communications sont représentées à l'aide d'opérations de lecture (read) et d'écriture (write) dans la mémoire au lieu de signaux propagés à travers des fils comme c'est le cas dans le niveau RTL. Cette abstraction permet de gagner d'un facteur de 100 à 1000 en vitesse de simulation. Ce gain conduit, systématiquement, à améliorer le prototypage rapide. Nous présentons dans la suite quelques uns des outils ESL les plus répandus.

5.2.1 Cofluent Studio

L'outil le plus proche de notre démarche est CoFluent Studio SDE, proposé par la société CoFluent Design [164]. Il est orienté vers la conception de systèmes électroniques numériques issus d'implémentations matérielles et logicielles. Les origines de ce logiciel se trouvent à l'École Polytechnique de Nantes où, sous la direction de Jean-PAUL CALVEZ, la méthodologie MCSE, pour méthodologie de conception de systèmes électroniques, a vu le jour [165]. CoFluent Studio supporte les étapes suivantes :

- la conception fonctionnelle qui est indépendante de la technologie matérielle,
- la conception architecturale,
- le prototypage et la réalisation.

Les algorithmes et les architectures sont conçus indépendamment dans les phases de conception fonctionnelle et architecturale [166]. Ces conceptions se font selon un modèle unifié du type graphe flot de contrôle. La conception architecturale permet d'étudier le portage de la solution fonctionnelle sur l'architecture.

CoFluent Studio permet aux concepteurs de capturer des informations via des graphiques, du code C et des spécifications d'attributs. Les utilisateurs commencent par créer et simuler un modèle fonctionnel intégralement temporisé et exécutable. Ensuite, ils définissent une architecture physique, en ayant recours à une analyse des performances et une co-simulation. Cette dernière sert à valider simultanément les étapes de la synthèse du logiciel et de la synthèse du matériel. La co-simulation consiste à simuler conjointement la partie matérielle et la partie logicielle du système. Enfin, CoFluent Studio permet de générer automatiquement un code C pour des noyaux temps réel et un code VHDL pour les portions matérielles de la conception (implémentation des systèmes sous la forme de circuits intégrés). Les concepteurs peuvent vérifier le comportement du modèle architectural en analysant et en co-simulant les performances. Trois niveaux d'abstraction ont été identifiés pour l'analyse des performances : i) le niveau Système, ii) le niveau composants, iii) le niveau des communications entre composants. CoFluent Studio traduit le modèle en un programme C++ ou SystemC. Le moteur de simulation exécute directement le code contenu dans les opérations au lieu d'émuler chaque instruction d'une unité centrale. Pendant la simulation, les concepteurs peuvent visionner des informations comme le tableau d'exécution d'un système, l'évolution des variables d'un système, des indices de performances comme l'utilisation mémoire et le débit de traitement d'images. Dans la dernière étape les utilisateurs peuvent générer un code C pour certains systèmes d'exploitation temps réel, tels que VxWorks de Wind River.

Conformément à la figure 5.1, l'approche préconisée par MCSE, ainsi que l'utilisation de l'outil CoFluent Studio, permettent, dans un premier temps, de modéliser et de valider une application à haut niveau d'abstraction. L'étape de validation conduit, dans un second temps, à une évaluation précise des performances d'implantation des différentes configurations architecturales.

Pour conclure, CoFluent Studio est un outil niveau système bien adapté pour des applications à fort contrôle pour réaliser des systèmes temps réel réactifs ou encore pour l'intégration de systèmes dans des circuits.

Il propose une approche générale pour la conception de haut niveau du domaine électronique, en considérant la modélisation du comportement des systèmes avec des modèles formels de calcul, de plus une issue vers la matérialisation est proposé. Par contre cet outil est moins adapté pour le prototypage d'applications orientées flot de données. De plus, comme l'outil ne permet pas de distribuer et ordonnancer automatiquement une application sur une architecture multiprocesseur, c'est à l'utilisateur de le faire à la main. L'utilisateur n'a, par conséquent, aucune idée de la qualité de l'exploration du parallélisme.

5.2.2 Gedae

Développé dans le cadre du projet RASSP [167] (Rapid Prototyping of Application Specific Signal Processor), GEDAE (Graphical Entry, Distributed Applications Environment [168]) est un logiciel commercial graphique, d'aide au développement d'applications de traitement du signal (systèmes temps réel) sur architectures multiprocesseur embarquées, vendu par la société [169]. Dans le domaine du Traitement de Signal (TS) GEDAE est considéré comme l'outil le plus complet. En effet GEDAE couvre tout ce domaine sans restriction apparente et permet de spécifier hiérarchiquement les applications à partir de bibliothèques de fonctions usuelles, de développer des fonctions utilisateur, de placer les traitements sur des architectures parallèles, de décrire les

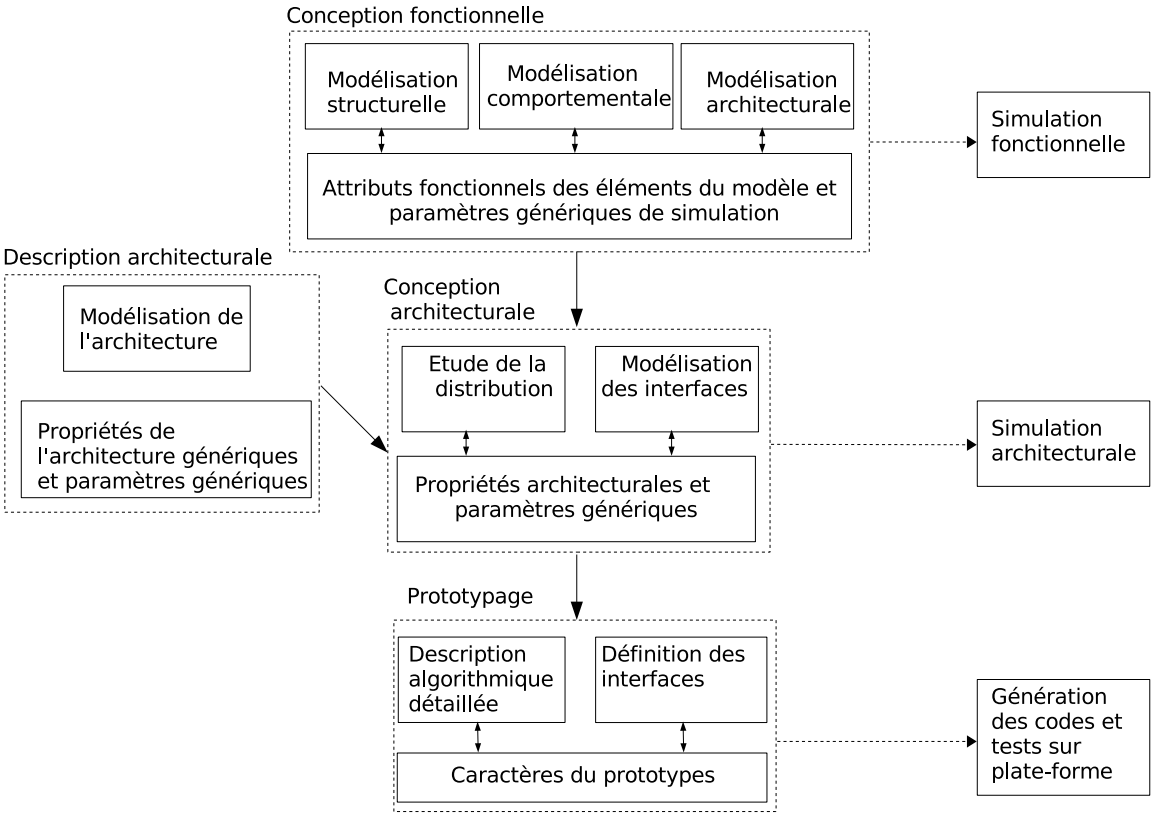

FIG. 5.1 – *La conception selon la méthodologie MCSE et utilisation de l'outil COFluent Studio*

transferts de données en tenant compte du parallélisme des traitements, de générer automatiquement du code C, de simuler l'exécution sur station de travail ainsi que le contrôle temps réel de l'exécution (visualiser la place mémoire occupée par chaque traitement et visualiser le temps passé à exécuter chaque traitement). Actuellement le logiciel GEDAE tourne sur les stations de travail Solaris, Windows 2000, et Redhat Linux, et génère du code pour le système hôte VxWorks, et les systèmes multiprocesseur CSPI, Ixthos, Mercury, et Sky.

GEDAE repose sur un formalisme de graphe flot de données, conditionnel, acyclique. GEDAE fournit un environnement hiérarchique pour spécifier les algorithmes. On décompose l'application en boîtes fonctionnelles réutilisables. Ces boîtes donnent une structure modulaire aux graphes flot de données. Elles peuvent comprendre d'autres boîtes hiérarchiques, des boîtes primitives, des entrées, des sorties, des paramètres et des équations mais l'imbrication ne peut être récursive. Les boîtes primitives sont atomiques du point de vue du placement, c'est-à-dire qu'on ne peut les distribuer que sur un unique processeur.

L'architecture est décrite par un unique fichier de configuration, elle contient 4 sections : i) la section processeur, ii) la section communication, iii) la section mémoire, ix) la section système pour insérer les paramètres d'initialisation.

GEDAE ne fournit pas d'outils automatiques de distribution et d'ordonnancement de l'algorithme sur l'architecture. Le partitionnement et la distribution sont donc donnés graphiquement par l'utilisateur. Les applications GEDAE peuvent être distribuées sur un réseau de stations de travail, sur un système de processeurs embarqués ou sur une station de travail multiprocesseur.

GEDAE génère un exécutif codé en C ANSI pour chacun des processeurs embarqués à partir du partitionnement et de la distribution utilisateur. Un noyau temps réel GEDAE porté sur la machine cible et résident sur chacun des processeurs embarqués exécute l'application auto-codée. Il génère du code pour différentes architectures COTS telles Mercury, CSPI, Ixthos, et Alex.

En développant GEDAE, ses concepteurs ont cherché à rendre ses étapes principales de fonctionnement aptes à être réitérées (ces étapes sont visibles sur la figure 5.2). Cette itérativité évite le développement en cascade (waterfall model) [170] et toutes les inefficacités qu'il engendre. Une des raisons qui permettent l'itération est la dissociation entre le développement et l'implémentation de l'application. Par ailleurs l'ensemble des architectures cibles pour cette implantation n'est pas extensible par l'utilisateur. De plus l'outil n'effectue aucune optimisation automatique, ni au niveau de la distribution et de l'ordonnancement, ni au niveau de l'optimisation de la mémoire.

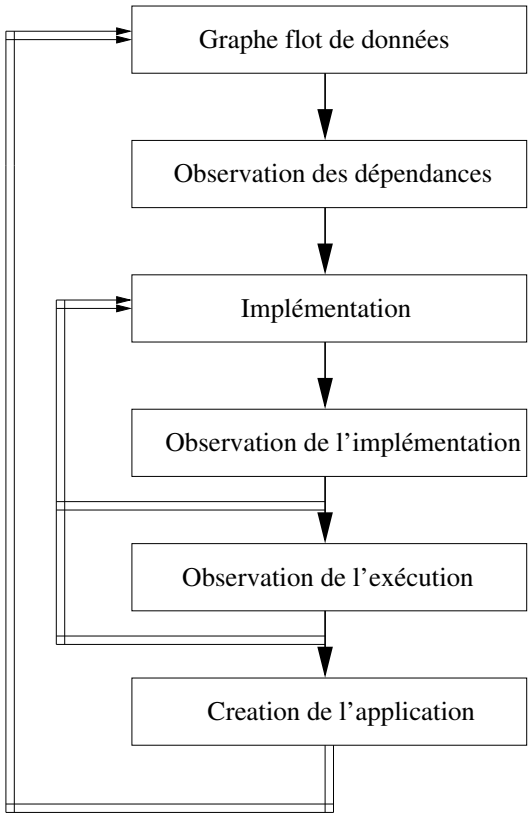

FIG. 5.2 – *Principales étapes de fonctionnement du logiciel GEDAE*

5.2.3 Ptolemy II

Ptolemy II [171, 172] est un logiciel développé à l'université de Berkeley. Il permet la modélisation, la conception et la simulation des systèmes embarqués hétérogènes. Ptolemy II est programmé en Java. Tous les composants dans Ptolemy II sont représentés par des classes Java. Parmi ces composants, on trouve les acteurs, les ports, les relations, etc. Ptolemy II vise le secteur des systèmes embarqués complexes, qui combinent plusieurs technologies, comme celles, par exemple, de l'électronique analogique et numérique, du matériel et du logiciel, des composants électroniques et mécaniques, et qui combinent également plusieurs types de traitements, comme le traitement du signal, les asservissements en boucle fermée, les automates de décision séquentielle, ou les interfaces utilisateur.

Pour manipuler plusieurs modèles en même temps, Ptolemy II se base sur la notion de domaine. Chaque domaine possède sa propre sémantique, quoique tous les domaines utilisent le modèle graphe flot de données comme modèle de base. Les entités de calculs sont les noeuds, et les arcs les relations entre ces entités. Pour la plupart des domaines, les entités sont des acteurs c'est-à-dire des fonctions ou traitements qui consomment des données et produisent d'autres (sauf pour le domaine FSM ou les entités sont des états), et les relations représentent les communications entre acteurs (pour le domaine FSM, des transitions). Chaque domaine est géré par un directeur.

La conception de Ptolemy II est basée sur la méthodologie orientée acteur [173] . Un acteur a une interface composée de ports d'entrée et de ports de sortie et de paramètres. Certains acteurs sont conçus pour être polymorphes de domaine. Ce type d'acteurs peut donc opérer dans divers domaines en accueillant les sémantiques de ces derniers avec les paramètres par défaut, tandis que d'autres sont spécifiques à un domaine, c'est-à-dire dédiés à un seul domaine. Comme cela est montré sur la figure 5.3, les modèles dans Ptolemy II sont des graphes où les noeuds représentent des entités décrivant des acteurs et les arcs sont des liaisons de communication entre ces acteurs. Plusieurs acteurs peuvent être rassemblés et connectés pour former un modèle hiérarchique.

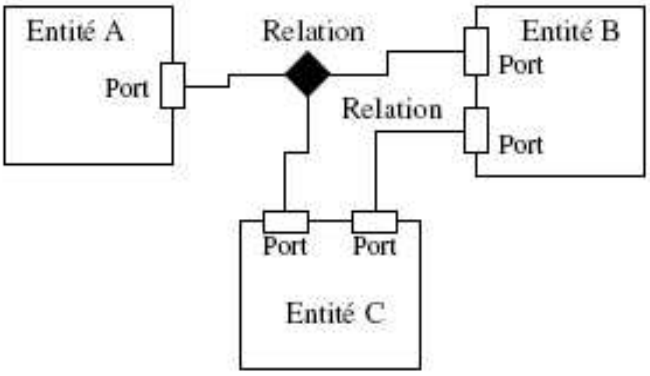

FIG. 5.3 – *Représentation graphique des modèles dans Ptolemy II*

Parmi les domaine que définit Ptolemy II on peut citer :

– CSP (Communicating Sequential Process): dans ce domaine, les processus communiquent

par des points de synchronisation appelés rendez-vous. Lorsque deux processus atteignent un point de synchronisation, ils communiquent par des actions atomiques,

– CT (Continuous Time) : dans ce domaine, c'est un signal continu qui représente le temps qui permet aux acteurs d'interagir. Il est bien adapté pour modéliser les circuits analogiques et les systèmes mécaniques. Les acteurs dans ce modèle peuvent être décrits par les équations différentielles suivantes :
$\frac{dx}{dt} = f(x;y;t)$
$y = g(x;u;t)$
Avec x : l'état du système, u son entrée, y sa sortie, et $\frac{dx}{dt}$ la dérivé de x par rapport au temps,
– DE (Discret Events) : dans ce domaine, qui est utilisé pour spécifier du matériel, et des systèmes de télécommunication numériques, les acteurs communiquent par des séquences d'évènements, un évènement est constitué d'une valeur et d'une étiquette de temps. Quand un acteur reçoit un évènement, il est activé pour être exécuté,
– FSM (Finite-State Machine) : ce domaine est différent des autres dans la mesure où les entités ne représentent pas des acteurs mais des états, et les arcs représentent des transitions entre les états. L'exécution est strictement une séquence ordonnée des transitions d'état,
– SDF (Synchronous Data Flow) : domaine des réseaux de processus synchrones où le nombre de jetons produits et consommés par chaque processus est fixé à la compilation, et seule une résolution des équations décrivant le système est nécessaire,
– PN (Process Networks) : domaine des réseaux de processus de Kahn, la sémantique définie par Kahn et MacQueen est respectée [174], et l'ordonnancement utilisé est celui proposé par Parks [175].

D'autres domaines existent (Synchronous/Reactive, Discret Time, Distributed Discret Events), mais sont encore au stade expérimental.

L'architecture est entièrement à la charge de l'utilisateur, qui doit la simuler soit implicitement soit explicitement dans ses modèles. La distribution est elle aussi à spécifier par l'utilisateur. L'ordonnancement des entités de calcul et les communications est effectué dynamiquement par l'ordonnanceur de chaque sous-domaine. Pour la génération de code Ptolemy II est doté d'un framework de composants spécialisés fondé sur Java et qui utilise XML pour la persistance des données.

Pour conclure, la modularité, la variété des modèles d'exécution disponibles et le concept de conception hiérarchique répond parfaitement aux problématiques de modélisation des systèmes embarqués et la simulation des applications associées, alors que l'optimisation et la génération du code sont négligées.

5.2.4 Logiciels dSPACE

La société dSPACE propose un ensemble de logiciels que l'on peut utiliser séparément ou combinés pour la simulation, le prototypage rapide, la génération automatique de code de production et la calibration [176].

À l'aide de ces logiciels, il est possible de traduire automatiquement en code assembleur, de compiler et d'exécuter les algorithmes de commandes décrits avec les blocs classiques de Simulink.

Ainsi on peut développer et tester l'algorithme de commande en simulation en utilisant Simulink et le traduire afin d'obtenir un code exécutable. Cela donne une grande flexibilité car le processus de développement et de test des algorithmes de commande est grandement raccourci. Un autre avantage est que grâce au logiciel d'interfaçage ControlDesk il est possible de visualiser en temps réel, de stocker les différentes grandeurs du système et de modifier les paramètres de la commande. Afin d'obtenir un outil ESL il faut combiner (comme le montre la figure 5.4) les logiciels de dSPACE suivants :

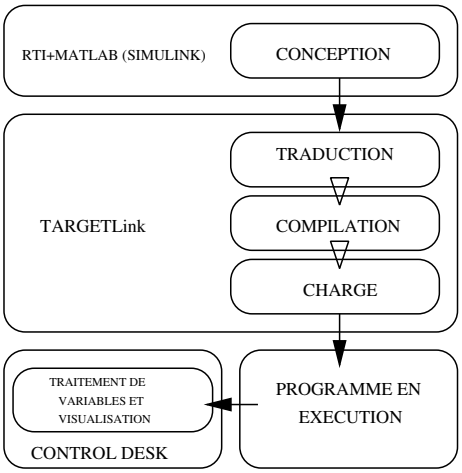

FIG. 5.4 – *Logiciels dSPACE*

– RTI (Real Time Interface) est une librairie de Simulink qui contient les blocs qui permettent d'adapter l'algorithme de la commande développée en Simulink pour qu'elle puisse être implémentée dans une architecture dSPACE particulière (processeur de signal DSP par exemple). Essentiellement ces blocs servent à relier les entrées/sorties de l'algorithme de la commande avec leurs équivalents physiques dans l'architecture. Ceci est réalisé en connectant des blocs spéciaux qui correspondent aux entrées/sorties physiques de l'architecture aux entrées/sorties de l'algorithme de commande. En fonction du contrôleur utilisé, la librairie est différente et les blocs aussi. La toute nouvelle interface RTI-MP supporte les systèmes multiprocesseurs basés uniquement sur les architectures modulaires dSPACE et aide à améliorer la performance des simulations temps réel. Elle permet aussi d'assister les concepteurs lors de l'installation de réseaux multiprocesseurs comprenant des canaux de communication ;
– TargetLink fournit un support d'évaluation des performances de l'algorithme de commande, des temps d'exécution et de la consommation en mémoire pour des applications données. Cette évaluation porte sur le choix des tailles de variables, l'échelle des variables, etc. TargetLink offre la possibilité de traiter des applications complexes en un temps court et ceci sans répercussions sur la qualité du résultat. Par contre il ne génère du code C qu'à partir du modèle graphique construit sous RTI-MATLAB/Simulink/Stateflow. En effet TargetLink génère du code pour des blocs de la librairie Simulink fournit par dSPACE. Par ailleurs la

relation entre le code généré et le comportement du modèle simulé n'est pas claire et souvent la rapidité du prototypage et l'optimisation mémoire prennent le pas sur la cohérence de la sémantique ;
– ControlDesk est un programme qui permet la construction d'une interface graphique de façon très simple. De plus il permet la visualisation temps réel des grandeurs physiques mesurées, il permet de les stocker pour ensuite pouvoir les traiter dans l'environnement Matlab [177].

En conclusion les logiciels dSPACE supportent essentiellement les architectures dSPACE et souffrent d'un manque de documentations relatives aux principes de base.

5.2.5 Giotto

Giotto est une plateforme proposant une abstraction de temps en le séparant des fonctionnalités. Cette abstraction est écrite à l'aide d'un langage spécifique à Giotto. Giotto a été développé pour des applications temps réel pouvant s'exécuter dans un environnement distribué. Un programme Giotto permet de spécifier les interactions possibles entre les composantes du système et leur environnement. Ces travaux ont été réalisés par Thomas A. Henzinger et son équipe à l'université de Berkeley [178].

Un programme Giotto est composé de quatre concepts principaux :

– les tâches : elles représentent des appels à des fonctions et peuvent avoir des paramètres définis à l'aide de ports,
– les ports : ils permettent d'effectuer les communications système/environnement. Trois types de ports sont définis selon leur rôle : d'entrée, de sortie et privés. Les sensors sont mis à jour par l'environnement, les autres le sont par Giotto. Les inputs et les outputs sont les paramètres des tâches. Les ports privés sont utilisés pour la communication entres des tâches concurrentes ;
– les drivers : ils permettent d'effectuer des opérations gardées. Ils font appel dans un premier temps à une fonction booléenne et selon la valeur retournée, appellent ou non la seconde fonction. Ils ont des paramètres pouvant être passés à l'une ou l'autre ou les deux fonctions le composant ;
– les modes : ils représentent les éléments majeurs d'un programme en Giotto. À un instant donné, un programme Giotto ne peut se trouver que dans un seul mode. Il est composé d'une période, d'un ensemble d'Outputs, d'invocations de tâches, de mise à jour de capteurs et d'un ensemble de changement de mode. Les invocations, les mises à jour et les changements ont une fréquence et seront appelés pendant la période selon cette fréquence. Un changement de mode décrit la transition d'un mode à un autre. Il est composé d'une fréquence, du mode cible et d'un driver. La garde du driver est évaluée selon la fréquence du changement.

Sur la figure 5.5, *In* (resp. *Out*) dénote l'ensemble des ports d'entrée (resp. de sortie). Une tâche est caractérisée par un état, représenté par *St*, qui est vu comme un ensemble de ports privés, dont les valeurs sont inaccessibles en dehors de la tâche. Par ailleurs, chaque tâche possède une

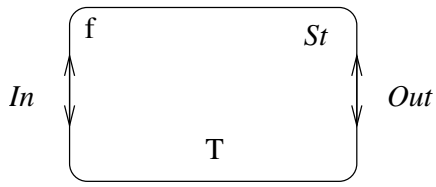

FIG. 5.5 – *Une tâche T*

fonction f qui calcule la valeur des ports de sortie et de l'état suivant de la tâche, à partir de la valeur des ports d'entrée et l'état courant de la tâche.

Pour résumer, la démarche Giotto consiste à séparer la conception en termes fonctionnels et d'exigences temporelles des questions relatives à son implantation. La description du système dans la notation formelle Giotto permet de décrire en Java ou en C les fonctions qui le composent et les contraintes temporelles relatives à leur exécution. Un programme peut également renfermer des contraintes de plate-forme, c'est ce que nous appelons contraintes de localité dans 1.3.4 dans l'état de l'art de la première partie du manuscrit. La compilation du texte produit vérifie ensuite qu'une structure d'ordonnancement existe sur l'architecture matérielle (éventuellement multiprocesseur) cible et produit, le cas échéant, le code permettant la mise en oeuvre des tâches conformément aux exigences de la spécification. Cette indépendance de la conception effectuée par rapport à la plate-forme de son exécution facilite le développement et permet la réutilisation du système sur différentes structures.

5.2.6 MLDesigner

MLDesigner est une approche pour les systèmes électroniques complexes utilisés dans des systèmes aérospatials et planétaires discutée dans [179]. Il s'agit d'une approche hiérarchique qui cherche à valider les composantes à concevoir dans une représentation comportementale de la mission où ils seront utilisés. Elle offre des solutions au niveau système qui intègrent la conception et l'analyse à plusieurs niveaux : opérationnel, architectural (performance), fonctionnel (algorithmique) et d'évaluation. L'outil correspondant automatise le processus de conception depuis la spécification du système jusqu'à son implémentation. Il permet aussi la validation et la co-simulation de systèmes mixtes hardware/software. Les systèmes et les performances attendues sont représentés à haut niveau par des machines à états finis. La partie matérielle est modélisée en termes de ressources utilisées par le logiciel comme le temps d'exécution des fonctions et des procédures. MLDesigner fournit à l'utilisateur des modèles écrits sous XML, les résultats des simulations et le code généré à travers une interface unifiée. Le principal avantage de MLDesigner est de proposer un environnement général de haut niveau pour des systèmes numériques mixtes dédiés à des fonctions critiques. En outre MLDesigner fournit des passerelles (importation de modèles FSM, SDF/DDF) avec SPW de Cadence, MatLab de Mathworks et COSSAP, il peut importer et exécuter des modèles de Ptolemy II.

Ce logiciel, étant une version améliorée de Ptolemy II destinée à l'industrie, garde cependant les mêmes manques que celui-ci.

5.3 État de l'art des outils pour l'analyse et l'ordonnancement temps réel

Il existe des outils qui partent d'un système spécifié avec un modèle donné, et qui effectuent un ordonnancement et une analyse d'ordonnançabilité hors-ligne. En général, le modèle est simple, et prend en compte des tâches temps réel périodiques ou sporadiques, avec contraintes de dépendance, avec possibilité de prise en compte de tâches apériodiques temps réel souple. Il existe énormément d'outils dans ce domaine et il est difficile d'apprécier leurs différentes capacités ou de faire la part entre la réalité et les possibilités annoncées. Pour comparer ce type d'outils il est impératif de considérer les objectifs qui diffèrent d'un outil à un autre, le modèle temps réel adopté et les algorithmes ou méthodes qu'ils pratiquent.

Les outils d'analyse d'ordonnançabilité sont essentiellement utilisés au début de la conception par d'autres outils, comme ceux qui sont cités dans la section 5.2, pour s'assurer que les contraintes temporelles d'une application sont respectées. Dans le domaine du temps réel embarqué, d'innombrable outils d'analyse d'ordonnançabilité ont été déjà développés ou sont en cours de développement que ce soit dans l'industrie ou dans les universités. Les deux outils les plus répandus sont :

- Tri-Pacific propose un produit composé d'un ensemble d'outils incluant Rapid Rma ex PERTS (Prototyping Environment for Real Time Systems) [180], Rapid Sim, et Rapid Build [181]. Cet ensemble est basé sur des recherches menées par l'université de l'Illinois dans Urbana-Champaign, et emploie l'approche RMA (Rate Monotonic Analysis). Il permet au concepteur de tester, simuler, et exécuter des modèles de tâches temps réel suivant les différents scénarios de la conception et évalue différents ordonnancements pour déterminer celui qui optmise les performance d'un système.
- TimeSys propose l'outil TimeWiz [182] qui permet à l'utilisateur, avant d'implémenter un système, d'analyser et de simuler le comportement temporel de ce système. Cet outil permet aussi d'exécuter ces applications temps réel sur un réseau de stations de travail. Comme pour l'ensemble Tri-Pacific, TimeWiz est basé sur l'approche RMA.

L'intérêt de ces deux outils est qu'ils fournissent des méchanismes appropriés pour intégrer et exploiter différentes caractéristiques, telles que l'extraction de la charge de travail ou l'analyse sous un seul environnement de développement. Cependant ils se concentrent sur une seule approche d'analyse qui est RMA. Par ailleurs il existe d'autres outils d'analyse d'ordonnançabilité parmi lesquelles :

- Cheddar [14] est un logiciel conçu dans un but pédagogique à l'université de Bretagne Occidentale, par l'équipe Lisyc. Il est développé en Ada95 et en GtkAda et devrait fonctionner sur toutes les plates-formes supportées par GNAT (un compilateur ada). Il comporte deux composants logiciels : i) une interface qui permet à l'utilisateur d'enregistrer le jeu de tâches qu'il souhaite faire analyser. Il peut ainsi, choisir un algorithme d'ordonnancement (RM, EDF, DM, LLF et POSIX 1003b), un protocole, lancer ou pas la représentation graphique de l'ordonnancement des tâches. Les résultats des simulations seront présentés sur la même

interface; ii) Une bibliotheque comportant les principaux résultats de la théorie de l'ordonnancement temps réel ainsi que quelques outils de files d'attente.

– ProtEx [183] étend le contexte d'étude afin de supporter les analyse de bout-en-bout [184]. Cet outil permet d'analyser des systèmes distribués temps réel.
– gRMA (a Graphical tool for Rate Monotonic Analysis) est un logiciel libre, dont les sources sont disponibles afin d'étendre les analyses [185].
– TkRTS [186] est un outil permettant d'analyser les systèmes monoprocesseurs avec les algorithmes classiques (cités dans la section 1.4.1), ainsi qu'avec les files multi-niveaux.
– OpenSTARS [187] est un outils d'analyse temps réel développé a l'université de Rhode Island en open-source.

Chapitre 6

SynDEx multipériode

6.1 SynDEx monopériode

L'appellation AAA regroupe un ensemble de recherches menées au niveau national [188], visant à développer des méthodes systématiques de meilleure mise en correspondance entre l'algorithme d'une part, et l'architecture d'autre part en vue de développer des méthodologies formelles permettant de réaliser une implantation optimisée d'un algorithme respectant des contraintes de temps réel, de précédence, etc.

Pour répondre spécifiquement à toutes les phases du prototypage rapide d'applications temps réel distribuées embarquées, de la spécification initiale de l'algorithme et de l'architecture jusqu'à l'exécution optimisée temps réel de l'algorithme par les processeurs de l'architecture, la méthodologie AAA a été implantée au début des années 90 dans un logiciel interactif appelé SynDEx. Cette version traite les sytèmes de tâches sous la contrainte de précédence et minimise le makspan. La figure 6.1 décrit les fonctionnalités offertes par SynDEx ainsi que les interactions possibles qu'il peut avoir avec l'utilisateur.

6.1.1 Présentation générale de la méthodologie AAA

SynDEx, logiciel de CAO au niveau système, est une concrétisation de la méthodologie AAA pour le prototypage rapide et l'implantation optimisée d'applications temps réel embarquées. Il permet en premier lieu de spécifier l'algorithme d'application et l'architecture multiprocesseur. Il réalise ensuite une adéquation qui est une implantation optimisée de l'algorithme sur l'architecture, dont le résultat est une prédiction temporelle de l'exécution de l'algorithme sur cette architecture. Il génère enfin automatiquement pour chaque processeur un exécutif temps réel dédié. Une présentation plus détaillée de cette méthodologie est donnée dans [142]. Dans la suite on va essayer de dresser un descriptif assez général tout en insistant sur les parties qui font l'objet de la nouvelle version de SynDEx.

Jusqu'à ce stade du manuscrit on a utilisé le terme 'tâche' pour désigner un ensemble d'instructions destinées à être exécutées et le terme 'processeur' pour désigner la machine d'exécution. Toutes les publications concernant AAA et SynDEx on utilise les termes 'opération' et 'opérateur'

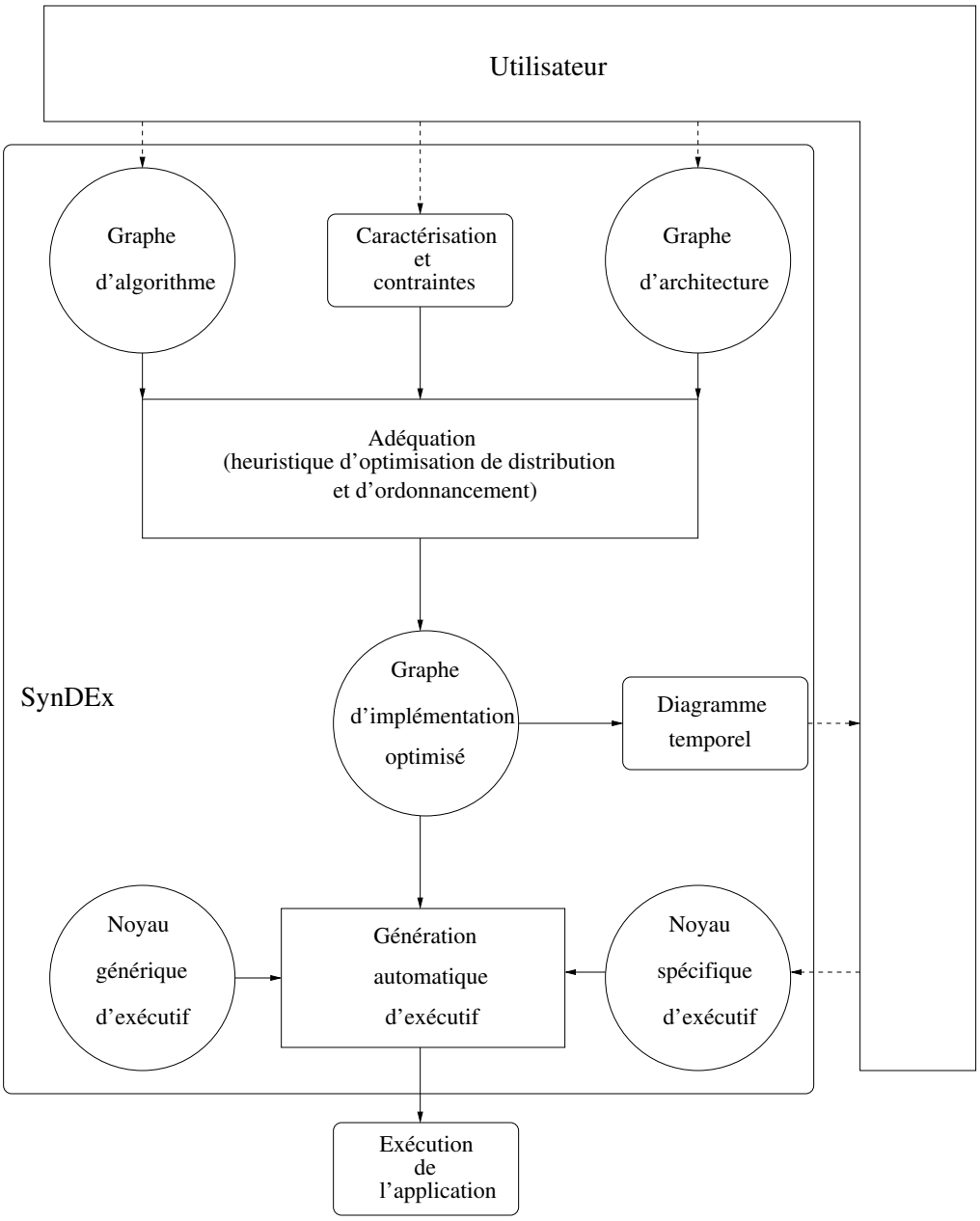

FIG. 6.1 – *Fonctionnalités de SynDEx*

[189, 142, 190]. Dans ce chapitre, pour des raisons de conformité on reprend les termes 'opération' et 'opérateur'. Toutefois, on insiste sur le fait que tous les résultats, obtenus dans la première partie du manuscrit, qui s'appliquaient aux tâches et aux processeurs s'appliquent aux opérations et aux opérateurs.

6.1.2 IHM

6.1.2.1 Modèle d'algorithme

Le modèle d'algorithme AAA est un graphe de dépendance de données hiérarchique, conditionné et factorisé [191]. Il s'agit d'un graphe orienté acyclique (DAG) [192], dont les sommets sont des opérations partiellement ordonnées [193] (parallélisme potentiel) par les dépendances de données inter-opérations (diffusion de données à travers des hyperarcs orientés pouvant avoir pour une seule origine plusieurs extrémités ou l'inverse).

Ce graphe de dépendances, étant infini, est modélisé par un motif infiniment répété. Il peut être décrit de manière hiérarchique : chaque opération du graphe peut contenir un sous-graphe permettant une spécification hiérarchique de l'algorithme jusqu'aux opérations atomiques. Une opération atomique est une opération élémentaire ne contenant que des ports d'entrée, de sortie, ou d'entrée-sortie. Les opérations atomiques que l'on peut définir sous SynDEx sont :

- Sensor : qui correspond aux entrées du graphe flot de données (capteur). Cette opération peut uniquement produire des données, elle ne contient que des ports de sortie;
- Actuator : qui correspond aux sorties du graphe flot de données (actionneur). Cette opération peut uniquement consommer des données, elle ne contient que des ports d'entrée;
- Constant : qui produit des données identiques lors de chaque exécution de l'algorithme. Il suffit donc de l'exécuter lors de la première itération du graphe d'algorithme;
- Delay : qui permet de spécifier les dépendances inter-itérations du graphe d'algorithme en les mémorisant d'une exécution à une autre. Il s'agit d'une dépendance de donnée entre chaque motif du graphe flot de données;
- Function : qui est une opération atomique, quand elle contient uniquement des ports d'entrée et de sortie. Cependant cette opération peut elle-même contenir des opérations de calcul, de conditionnement et de répétition, elles-mêmes hiérarchiques.

Le graphe d'algorithme traite aussi les dépendances de conditionnement et fournit la possibilité de spécifier les opérations répétées sous forme factorisée.

6.1.2.2 Modèle d'architecture

Le modèle d'architecture hétérogène multi-composants [142, 194, 189, 188] choisi dans AAA est un graphe orienté, dont chaque sommet est une machine à états finis (machine séquentielle) et chaque arc une connexion physique entre deux machines à états finis. Dans AAA, il existe cinq types de sommets :

- opérateur : pour exécuter des opérations de calcul,
- médium de communication : pour exécuter des opérations de communication (bus/mux/démux, mémoire).

La mémoire peut aussi être considérée comme une machine séquentielle. Dans AAA, il existe deux types de sommets mémoire :

- la mémoire RAM (à accès aléatoire) pour stocker les données ou programmes locaux à un opérateur,

– la SAM (à accès séquentiel).

L'hétérogénéité signifie non seulement que les sommets peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations et taille mémoire des données communiquées par exemple), mais aussi que certaines opérations peuvent n'être exécutées que par certains opérateurs, ce qui permet de décrire tout autant des composants programmables (processeurs-FPGA) que des composants non programmables (ASIC).

6.1.3 Mise à plat

Le graphe d'algorithme spécifié par l'utilisateur est mis à plat par SynDEx avant d'effectuer l'adéquation. La mise à plat du graphe a pour objectif de résoudre les références (les références sur des définitions hiérarchiques sont remplacées récursivement par le graphe d'algorithme de ces définitions) et de défactoriser les définitions répétées ou conditionnées, mettant ainsi à plat sa hiérarchie : expansion du graphe d'algorithme. Le graphe obtenu après mise à plat est un graphe de dépendance portant uniquement sur des opérations atomiques.

6.1.4 Adéquation

L'adéquation consiste à trouver la meilleure distribution et le meilleur ordonnancement d'un graphe flot de données sur une architecture cible.

6.1.4.1 Implantation

La distribution consiste à ordonnancer chaque opération de l'algorithme sur un opérateur capable de l'exécuter. Ceci conduit à une partition de l'ensemble des opérations de l'algorithme en autant de sous-graphes que d'opérateurs. Ensuite pour chacune de ces opérations, un sommet d'allocation mémoire est ajouté et est affecté à une RAM connectée à l'opérateur qui exécute l'opération. Enfin,

chaque dépendance de données inter-opérateurs (c'est-à-dire entre opérations ordonnancer sur des opérateurs différents) est affectée à un chemin reliant les deux opérateurs (chemin dans le graphe de l'architecture).

Entre ces deux opérations de l'algorithme, autant d'opérations de communication que de média de communication, autant de sommets identité [142] que de sommets bus/mux/démux/arbitre et autant de sommets d'allocation de mémoires données qui sont communiquées que de sommets mémoire SAM et RAM, sont crées et affectés au chemin qui relie ces deux opérations. Il ne reste plus qu'à affecter ces éléments aux sommets correspondants du graphe de l'architecture.

Une implantation est donc le résultat d'une transformation du graphe de l'algorithme (ajout de nouveaux sommets et de nouveaux arcs) en fonction du graphe de l'architecture. Seulement un algorithme de distribution et d'ordonnancement est nécessaire pour permettre cette implantation.

6.1.4.2 Heuristique d'ordonnancement

L'heuristique proposée est basée sur la méthode de liste (list scheduling). Elle définit des règles de construction de liste et une fonction de coût qui prend en compte la durée d'exécution des opérations du graphe d'algorithme et la durée des communications quand des opérations dépendantes sont distribuées sur des opérateurs différents. Rappelons que les opérations prises en compte ne sont pas périodiques.

Le principe de cette heuristique est qu'à chaque étape, une opération est sélectionnée dans une liste d'opérations dites candidates, puis cette opération est distribuée et ordonnancée sur un opérateur choisit parmi l'ensemble des opérateurs. La liste des opérations est alors remise à jour.

Avant d'exécuter l'heuristique de distribution et d'ordonnancement et pour gérer la construction des communications, il faut construire des routes entre chaque couple d'opérateur du graphe d'architecture. Pour cela nous construisons des tables de routage pour chacun des opérateurs du graphe de l'architecture.

Avant de détailler l'heuristique, il est indispensable de définir la fonction de coût utilisée dans celle-ci. On note par ρ la fonction de coût qui est appelée aussi pression d'ordonnancement et qui mesure le degré d'urgence à ordonnancer une opération. Pour cela elle tient compte de la flexibilité F d'une opération (plus une opération a de la flexibilité, moins il est urgent de l'ordonnancer), mais aussi de l'allongement du chemin critique qu'induit l'ordonnancement d'une opération qui est appelé la pénalité d'ordonnancement.

La pénalité d'ordonnancement engendrée par l'ordonnancement de l'opération o_i sur l'opérateur p_j est notée par $P^{(n)}(o_i,p_j)$. Si $R_{ij}^{(n)}$ est la longueur du chemin critique à l'étape (n) où on veut ordonnancer o_i sur p_j et $R^{(n-1)}$ représente la longueur du chemin critique à l'étape précédente, alors :

$$P^{(n)}(o_i,p_j) = R_{ij}^{(n)} - R^{(n-1)}$$

La flexibilité d'ordonnancement engendrée par l'opération o_i quand elle est ordonnancée sur p_j est donnée par :

$$F^{(n)}(o_i,p_j) = R^{(n)} - \bar{S}^{(n)}(o_i,p_j) - S^{(n)}(o_i,p_j)$$

$\bar{S}$ et S sont respectivement la date de début au plus tard depuis la fin et la date de début au plus tôt depuis le début.

Enfin la pression d'ordonnancement est donnée par :

$$\rho^{(n)}(o_i,p_j) = P^{(n)}(o_i,p_j) - F^{(n)}(o_i,p_j)$$

Cette fonction de coût mesure le degré d'urgence à ordonnancer une opération. Pour cela elle considère la flexibilité d'une opération qui signifie que plus une opération a de la flexibilité, moins il est urgent de l'ordonnancer. En même temps elle considère l'allongement du chemin critique provoqué par l'ordonnancement de cette opération. Plus de détails sur le calcul de la pression d'ordonnancement se trouvent dans [190, 189, 142].

L'heuristique d'ordonnancement peut être résumée par les points suivants :

– phase de routage : entre chaque paire d'opérateurs, l'heuristique repère le plus court chemin,

– parmi toutes les opérations du graphe d'algorithme l'heuristique met dans la liste des opérations candidates à l'ordonnancement celles qui n'ont pas de prédécesseurs,
– la sélection de l'opération à ordonnancer s'effectue en 2 étapes :
    1. en calculant la fonction de coût de chaque opération candidate sur chaque opérateur pour trouver celui qui minimise cette fonction,
    2. une fois que toutes les paires (opération candidate, meilleur opérateur) auront été trouvées, l'heuristique sélectionne la paire qui maximise la fonction de coût, c'est-à-dire, celle qui contient l'opération la plus urgente à ordonnancer),
– l'opération sélectionnée est ordonnancée sur son meilleur opérateur,
– cette opération est supprimée de la liste,
– toutes les opérations qui n'ont plus de prédécesseurs sont mises dans la liste.

L'algorithme 16 décrit les étapes de l'heuristique d'ordonnancement.

Algorithme 16 Heuristique d'ordonnancement

1: En utilisant un algorithme de plus court chemin repérer pour chaque d'opérateurs le plus court chemin entre eux
2: Initialisation de l'ensemble Δ des candidats avec les opérations sans prédécesseurs
3: **tant que** L'ensemble Δ n'est pas vide **faire**
4: **pour** *i* allant de 1 au nombre de candidats **faire**
5: Calcul de la fonction de coût pour chaque candidat sur tous les opérateurs
6: Choisir la fonction de coût minimale entre toutes celles calculées sur chacun des opérateurs (l'opérateur en question est considéré comme le meilleur opérateur pour cette opération). Cela forme l'ensemble des paires (opération, meilleur opérateur)
7: **fin pour**
8: **pour** *i* allant de 1 au nombre de paires (opération, meilleur opérateur) **faire**
9: Choisir la paire (opération, meilleur opérateur) qui a la fonction de coût maximale
10: **fin pour**
11: Ordonnancer cette opération sur son meilleur opérateur
12: Ajout à l'ensemble Δ, des successeurs ordonnançables de ce candidat
13: Suppression du candidat ordonnancé de l'ensemble Δ
14: **fin tant que**

6.1.5 Génération de code

Quand le calculateur est une machine multiprocesseur dont les séquenceurs d'instructions sont capables de fonctionner en parallèle, l'implantation parallèle de l'algorithme consiste à le traduire en un ensemble de programmes chargés puis exécutés simultanément sur chaque opérateur de l'architecture (un programme par opérateur). Ces programmes coopèrent (communiquent) pour

réaliser les fonctionnalités de l'algorithme. L'ensemble de ces programmes forme le logiciel de l'application.

La génération automatique d'exécutif distribué se fait suivant des règles décrivant la transformation d'un graphe d'implantation optimisé en un graphe d'exécution. Pour chaque opérateur (resp. médium de communication) SynDEx construit un programme séquentiel formé de la séquence des opérations de calcul (resp. communications) qu'il doit exécuter. Les opérations de communication sont :

- des "Send" et des "Receive" de données transmises entre communicateurs via une SAM (communication par passage de messages),
- des "Write" et des "Read" quand les données sont transmises via des RAM (communication par mémoire partagée).

Pour garantir les précédences d'exécution entre les opérations appartenant à des séquences de calcul et/ou de communication différentes, et pour garantir l'accès en exclusion mutuelle aux données partagées par les opérations de ces séquences, SynDEx ajoute des opérations de synchronisation avant et après chaque opération qui lit (resp. écrit) une donnée écrite (resp. lue) par une opération appartenant à une autre séquence. Ces opérations de synchronisation utilisent des sémaphores générés automatiquement. Les principes majeurs de la génération d'exécutif de SynDEx sont très bien détaillés dans le document [194].

La version 6 est la dernière mise à jour majeure de SynDEx. Le code a été réécrit en OCaml (langage de type fonctionnel développé à l'INRIA) et comporte environ 50,000 lignes. Les possibilités de modularité hiérarchique, de répétition et de conditionnement ont été intégrées. Le générateur de code a été nettement amélioré en rapidité. L'interface graphique a été codée en Tcl/Tk.

6.2 Avantages de SynDEx vis-à-vis des logiciels existants

L'avantage principal du logiciel SynDEx sur la plupart des logiciels existant est qu'il permet l'ordonnancement et la distribution automatique sur architecture distribuée d'un programme décrivant les fonctionnalités d'un système. En outre cette implantation prend en compte les architectures composées d'opérateurs de types différents ainsi que le routage et le coût des communications.

Il existe d'autres logiciels permettant de faire de la distribution de programmes (voir la section 5.2), néanmoins, c'est avec l'apport de multiple travaux à l'instar de celui exposé dans ce manuscrit ainsi que dans le cadre d'autres thèses effectuées précédemment que SynDEx est capable de proposer davantage d'aptitudes. En effet, on a constaté que dans les logiciels permettant de faire de la distribution de programmes, les différentes alternatives d'un programme sont considérées comme atomiques lors de la distribution. Les programmes perdent alors une partie de leur parallélisme potentiel. Enrichir le langage SynDEx avec le conditionnement ainsi que la mise à plat associée [62] lui permet d'exploiter complètement, si l'architecture le permet, le parallélisme potentiel des programmes y compris celui contenu dans leurs différentes alternatives. Par ailleurs ce logiciel traite désormais les systèmes périodiques ainsi que tout ce que le traitement de ce type de systèmes entraîne comme l'étude d'ordonnançabilité par exemple. Les principales modifications apportées à la version monopériode pour en faire un logiciel traitant les systèmes multipériodiques sont détaillées dans la prochaine section.

6.3 SynDEx multipériode

Dans cette section nous évoquons les principales modifications qu'on a apportées à l'ancienne version de SynDEx pour obtenir la version multipériode. Cette nouvelle version traite des systèmes d'opérations sous les contraintes de précédence et de périodicité et offre toutes les fonctionnalités de l'ancienne version. Elle exploite les résultats théoriques obtenus dans le chapitre 2.

Si, dans la suite, un détail vous semble ambigu n'hésitez pas à consulter le manuel ou le tutaurial disponibles sur la page web du logiciel [195].

6.3.1 IHM modifiée

Au départ il nous fallait trouver un moyen d'attribuer les périodes aux opérations qui ne pénalise pas les facilités qu'offre SynDEx vis-à-vis de la construction des graphes (d'algorithme dans ce cas). Par exemple, quand on crée une opération, celle-ci peut être appelée à plusieurs reprises en utilisant plusieurs noms de référence. Par contre, si on attribuait la période à l'opération elle-même alors on obligerait l'utilisateur, au cas où il a besoin de faire appel à la même opération à plusieurs endroits dans le graphe d'algorithme mais avec des périodes différentes, de créer autant d'opérations que de périodes différentes.

Afin d'éviter ce désagrément à l'utilisateur, on attribue la période aux références et non aux opérations elles-mêmes. Ainsi quand l'utilisateur fait appel à une opération (il la référence) il apparaît, en plus de ce qui se trouve déjà dans SynDEx monopériode, un champs où il est possible d'insérer la période si l'application visée est multipériode. Donc, on a commencé par modifier, dans le fichier "types.mli" le type "reference" auquel on rajoute l'information de la période. La nouvelle déclaration du type "reference" qui peut être lu dans le fichier "types.mli" est la suivante :

```
and reference_type    = {
  mutable  ref_name                : string;
  mutable  ref_period              : int;
  mutable  ref_algorithm           : algorithm_type;
  mutable  ref_arguments_values    : Symbolic.expression     list;
  mutable  ref_condition           : condition_type;
  mutable  ref_position            : Coord.coord2d;
  mutable  ref_repetition          : repetition_type;
  mutable  ref_description         : string
}
```

Conformément à ce qui est dit dans la section 2.1.1.2 les opérations dépendantes doivent être à la même période ou à des périodes multiples. Pour faire respecter cette restriction et afin d'aider l'utilisateur à construire des graphes d'algorithme correctes on a introduit un message d'erreur. Ce dernier s'affiche quand l'utilisateur tente de créer un arc de dépendance entre deux opérations qui ont des périodes non-multiples. Par ailleurs, comme SynDEx permet de créer des opérations hiérarchiques (des opérations qui contiennent plusieurs autres opérations hiérarchiques ou atomiques), cette vérification se propage à l'intérieur de et entre ces opérations hiérarchiques. La

figure 6.2 expose un cas où on a essayé de lier par une dépendance deux opérations de périodes incompatibles (10 et 15). Ainsi la vérification se propage à l'intérieur des deux opérations hiérarchiques (fct_hierarchique_1 et fct_hierarchique_2) pour pouvoir vérifier toutes les dépendances qui peuvent lier les opérations atomiques contenues dans ces deux opérations hiérarchiques.

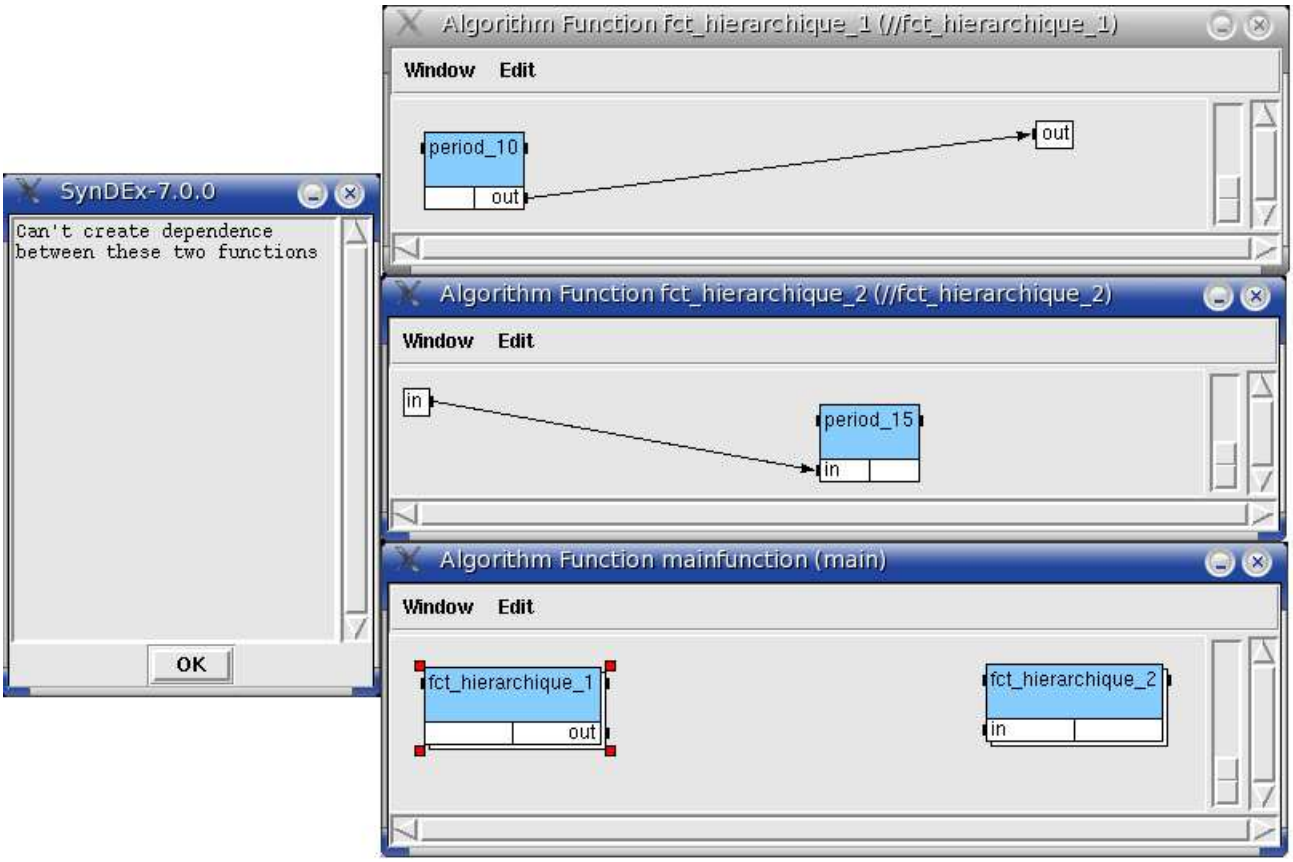

FIG. 6.2 – *Dépendance entre deux opérations hiérarchiques*

Il existe un deuxième type de vérification qui intervient sur deux niveaux :

- soit à l'attribution d'une période à une opération hiérarchique si celle-ci n'a pas encore de période (n'étant pas encore référencée),
- soit à l'attribution de la période à une opération alors que l'opération hiérarchique qui fait appel à cette opération possède déjà une période.

Sur le premier plan on vérifie si la période que l'utilisateur attribue à l'opération hiérarchique est compatible avec les périodes des opérations qu'elles contient. En effet, la période de l'opération hiérarchique doit être égale au (ou multiple du) *PPCM* des périodes des opérations qu'elle contient. Autrement cela induirait des erreurs pendant le déroulement du graphe (chapitre 2). Pour les mêmes

raisons, concernant le deuxième plan, la période d'une opération qu'on veut référencer doit être égale à (ou un diviseur de) la période de l'opération hiérarchique qui fait l'appel.

À chaque fois que l'utilisateur crée une dépendance entre deux opérations de périodes différentes (multiples) un message apparaît pour signaler que la taille du port d'entrée de l'opération consommatrice va changer de taille (plus de détails dans la prochaine section). Une fois que l'utilisateur aura fini de construire le graphe d'algorithme chaque opération (atomique ou hiérarchique) possède sa propre période si il s'agit d'une application multipériode ou bien toutes les opérations ont la même période qui est égale à 0 si l'application est monopériode. En passant le curseur de la souris sur une opération on peut lire la valeur de sa période sur la fenêtre principale du logiciel.

Donc, au niveau de l'IHM, on a essayé de modifier que là où ça été nécessaire afin de ne pas compliquer l'utilisation du logiciel. Par exemple on a évité de mettre un bouton différent pour chaque type d'adéquation (monopériode et multipériode). C'est le même bouton pour les deux adéquations et c'est au logiciel de distinguer la nature de l'application.

Les principales modifications concernant l'IHM ont été apportées aux fichiers "algorithm.ml" et "algorithm_ctk.ml".

6.3.2 Mise à plat modifiée

Après que l'utilisateur ait fini de construire l'application multipériode (graphe d'algorithme + graphe d'architecture), il peut lancer l'adéquation qui commence par la mise à plat. Les modifications apportées à la mise à plat de l'ancienne version sont :

6.3.2.1 Assignation

Au début de la transformation on a introduit la vérification de l'ordonnançabilité de l'application en exécutant l'algorithme de la recherche locale présenté dans le chapitre 2. Les fonctions de cet algorithme se trouvent dans le fichier "assignment.ml". Suivant le résultat de cette exécution, soit il apparaît un message qui signale qu'aucune assignation n'a pu être trouvée et du coup l'application est non-ordonnançable, soit chaque opération est assignée à un ou plusieurs opérateurs (voir la section 2.2.1).

6.3.2.2 Déroulement

Ensuite le graphe d'algorithme initial est déroulé suivant l'algorithme 3. les fonctions de cet algorithme se trouve dans le fichier "Unroll.ml". En plus de répéter les opérations suivant leurs périodes et l'hyper-période ainsi que de rajouter les dépendances (voir la section 2.2.2), des opérations de type "Implode" sont introduite dans le graphe d'algorithme. Ces opérations "Implode" permettent de regrouper les données envoyées par plusieurs instances de la même opération productrice à une instance de l'opération consommatrice (voir la section 2.2.2.1). Explicitement si l'opération "Implode" reçoit n tableaux (puisque les données circulent sous forme de tableaux) de taille m alors elle réunit ces données dans un tableau de taille $n * m$ et l'envoie à l'opération consommatrice. Par conséquent le port d'entrée de l'opération consommatrice, dont la taille au départ est égale à la taille du port de sortie de l'opération productrice (c'est-à-dire de taille m),

est remplacé par un port de taille $n * m$. Automatiser cette manoeuvre évite à l'utilisateur de le faire manuellement ce qui serait une lourde tâche en particulier si la taille de l'application était considérable. Parmi les cinq types d'opérations qui existent sous SynDEx : "Function", "Sensor", "Actuator", "Delay" et "Constant", il n'y a que les opérations de type "Constant" qui ne sont pas répétées puisque la valeur émise par une constante ne change pas. L'opération "Constant" prend une période égale à l'hyper-période pour être exécutée qu'une seule fois. L'assignation et le déroulement s'exécutent de manière invisible à l'utilisateur, par contre l'utilisateur peut consulter le graphe déroulé sous forme d'une image "PostScript" ou "Jpeg". Une fois que l'assignation et le déroulement soient exécutée la transformation poursuit son exécution de la même manière que dans l'ancienne version, à savoir : la mise à plat, la vérification de la cohérence du graphe et la détection de cycles.

6.3.3 Adéquation modifiée

Après la mise à plat, vient le tour de l'étape d'ordonnancement. Comme les opérations sont périodiques, après que la première instance d'une opération t_x soit ordonnancée, c'est-à-dire que sa date de départ soit connue, toutes les dates de départ des autres instances de cette opération sont calculées à l'avance en utilisant l'équation suivante : $S(t_{x_i}) = S(t_{x_1}) + T(t_x)(i-1)$. Quand une des instances devient une candidate à l'ordonnancement elle est ordonnancée directement à la date précalculée sur l'opérateur où elle a été assignée. On a introduit également dans la version monopériode de l'algorithme d'ordonnancement le test de la condition du théorème 2.10 qui précède l'ordonnancement de chaque première instance d'une opération (voir la section 2.2.3).

On a rajouté aussi, au cours de l'ordonnancement, un test qui permet de savoir si l'ordonnancement s'effectue en une ou deux phases (voir le dernier paragraphe de la section 2.2.3). Il faut savoir que suivant les applications multipériodiques, leurs ordonnancements peuvent s'étaler sur deux phases, une phase transitoire qui ne s'exécute qu'une seule fois suivie d'une phase permanente qui va s'exécuter indéfiniment. La longueur de chaque phase est égale à l'hyper-période. Dans le cas où il n'y a qu'une seule phase c'est cette même phase (de longueur égale à l'hyper-période) qui s'exécute indéfiniment.

Pour que la périodicité soit respectée on introduit, à chaque fois qu'il y a un temps creux (voir la section 1.2.3), une opération "wait" qui permet d'obliger l'opérateur, comme il n'a pas d'autres opérations à exécuter hormis celle qui suit le temps creux, à respecter la date de départ définie par l'algorithme d'ordonnancement. Une opération "wait" peut être remplacée par la fonction "sleep()" dans C ou OCAML par exemple. Elle est utilisée aussi pour ajuster la longueur de la ou les phases d'ordonnancement à l'hyper-période.

EXEMPLE 6.1

Pour avoir une idée de l'ordonnancement en deux phases nous proposons d'appliquer l'adéquation à un système composé d'un graphe d'algorithme de 3 tâches et une architecture monoprocesseur. les graphes d'algorithme et celui de l'architecture sont donnés par la figure 6.3. Le diagramme temporel résultant de l'adéquation du système précédent est donné par la figure 6.4. L'axe des temps est divisé en deux parties : une partie de couleur rouge pour la phase transitoire et une partie

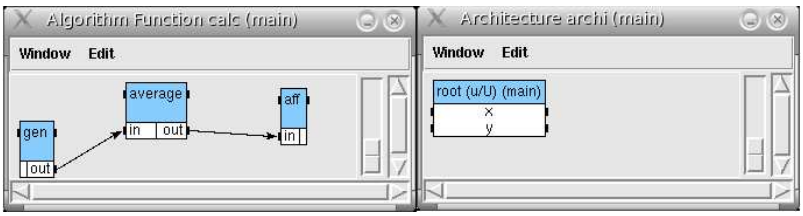

FIG. 6.3 – *Graphe d'algorithme et graphe d'architecture*

de couleur verte pour la phase permanente. Les opérations de la phase transitoire se distinguent des opérations de la phase permanente par un contour pointillé. Sur cet exemple on peut se rendre compte du rôle des opérations "wait" qu'on a rajoutées. Tout d'abord "wait_gen 1" qui sert à obliger l'opération "gen_1" a démarrer son exécution à la date 4 et du coup permettre à l'opération "gen" de respecter sa période. Ensuite "wait_1" qui sert à mettre la phase transitoire à la bonne longueur, c'est-à-dire 12 qui est la valeur de l'hyper-période.

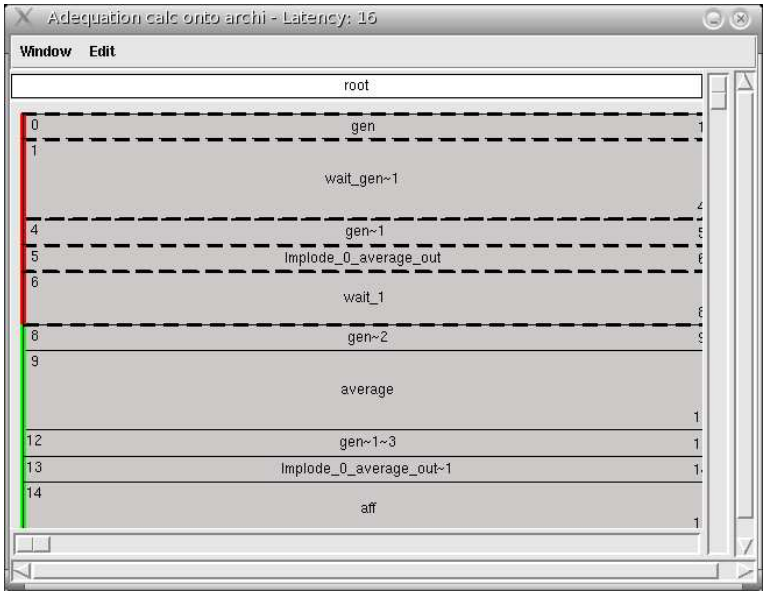

FIG. 6.4 – *Ordonnancement sur deux phases*

6.3.4 Génération de code modifiée

les modifications qu'on a faites au niveau de la génération de code visent essentiellement la distinction entre les phases transitoire et permanent. Dès lors la phase transitoire s'exécute avec la partie initialisation puis c'est la phase permanente qui s'exécute en boucle. La figure 6.5 illustre le code généré correspondant au système de la figure 6.3.

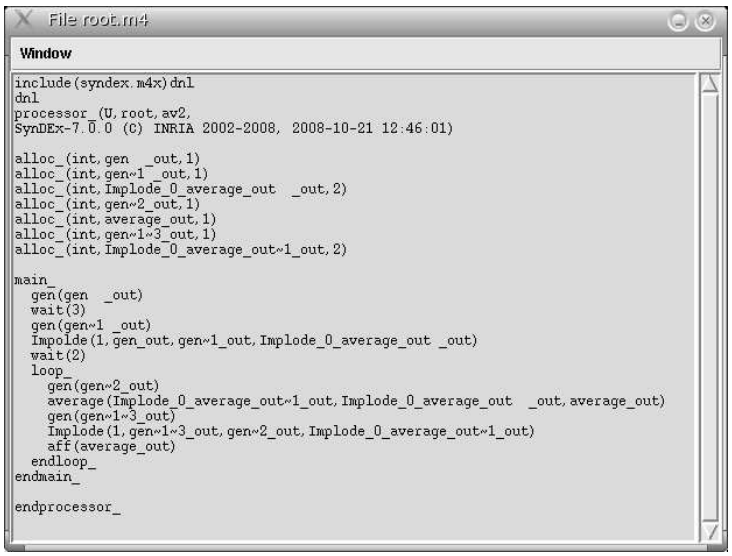

FIG. 6.5 – *Génération de code*

On a aussi intégré la définition de la fonction wait(x) (qui va contenir le code associé à l'opération "wait") dans la génération de code. Elle représente une fonction avec une seule variable x qui est la durée de temps pendant laquelle l'opérateur doit rester inactif.

Chapitre 7

Application de suivi pour train virtuel de CyCabs

Depuis quelques années, au sein de l'équipe AOSTE, des travaux sur le suivi d'un CyCab sont menés. Ceci consiste à implémenter un algorithme d'estimation de la distance entre deux véhicules basé sur la détection de contours dans des images, d'effectuer l'asservissement longitudinal puis latéral du CyCab suiveur et enfin de générer, grâce à l'outil SynDEx, un code temps réel distribué sur chacun des processeurs embarqués incluant les différentes communications et synchronisations.

L'apport de cette thèse par rapport à ces travaux a été de passer d'une application de suivi CyCab monopériode à une autre multipériode grâce à la nouvelle version de SynDEx. Cette nouvelle application, comme nous allons le voir, se rapproche d'avantage de la réalité puisqu'elle supporte des opérations ayant des périodes différentes. Alors qu'avec l'ancienne application toutes les tâches ont la même période même si ce n'est pas le cas dans la réalité.

7.1 Conduite automatique

Dans le cadre de la route automatisée, l'INRIA a imaginé un système de transport original de véhicules en libre-service pour la ville de demain. Ce système de transport public est basé sur une flotte de petits véhicules électriques spécifiquement conçus pour les zones où la circulation automobile doit être fortement restreinte. Pour tester et illustrer ce système, un prototype, nommé CyCab (contraction pour Cyber Cab), a été réalisé (Figure 7.1).

Les chercheurs de l'INRIA et de l'Inrets (Institut National de Recherche sur les Transports et leur Sécurité) travaillent depuis 1991 sur de nouveaux moyens de transport intelligent pour la ville. Ils étudient en particulier le concept du libre-service et celui de la voiture automatique. Les premiers résultats de recherche ont débouché sur le projet Praxitèle (1993-1999), qui était en exploitation à Saint-Quentin-en-Yvelines. Les partenaires industriels du projet étaient CGFTE (la filiale transports publics de Vivendi), Dassault Électronique, EDF et Renault. Dans le cadre du projet Praxitèle l'INRIA a démontré la faisabilité de la conduite automatique sous certaines conditions : créneau et train de véhicule expérimenté sur une Ligier électrique instrumentée à cet effet. Pour des raisons de législation et de responsabilité ces automatismes de conduite n'ont pas pu être

FIG. 7.1 – *Un CyCab*

implémentés sur les Clio électriques de Saint-Quentin-en-Yvelines. Le CyCab a ensuite été développé par l'INRIA avec l'aide de l'Inrets, de EDF, de la RATP et de la société Andruet S.A. pour montrer le potentiel de l'informatique dans la conduite de véhicules. Le CyCab est un véhicule électrique à quatre roues motrices et directrices avec une motorisation indépendante pour chacune des roues et pour la direction. Pour contrôler et commander les 9 moteurs du CyCab (4 de traction, 1 de direction et 4 de frein), une architecture matérielle a été choisie. Elle est constituée de noeuds intelligents pouvant gérer les différents moteurs du CyCab et répartie autour d'un bus de terrain CAN (Controller Area Network), très répandu dans le monde de l'automobile. Le rôle des noeuds est d'asservir les moteurs en fonction des consignes de vitesse et de braquage qui transitent sur le bus CAN soit en provenance de la position du joystick, soit par un programme de planification de trajectoires. Le noeud doit donc non seulement être capable de fournir la puissance nécessaire aux moteurs, mais aussi exécuter les boucles d'asservissement de vitesse ou de position. Pour ce faire il doit prendre en compte un certain nombre d'informations en provenance des capteurs proprioceptifs : état, odomètrie, fins de course, mesures de température, de courant, ... Un train de véhicules est constitué d'un véhicule de tête conduit par un chauffeur et d'autres véhicules automatisés, chacun suivant celui qui le précède. Ainsi le premier véhicule est suivi par le deuxième qui à son tour est suivi par le troisième ... C'est donc une procession de véhicules. Ce type d'automatisation a été pensé pour les conduites sur autoroute ou dans les périphériques. Ce procédé a l'avantage de maximiser la vitesse des véhicules ainsi que leur nombre tout en minimisant les accidents.

7.2 Architecture matérielle

7.2.1 Caractéristique générale d'un CyCab

Il existe différents types de CyCab. Celui qu'on utilise a les caractéristiques suivantes :

- longueur : 1,90 m,
- largeur : 1,20 m,
- poids total avec batteries : 300 kg,
- 4 roues motrices et directrices,
- vitesse théorique maximale : 20 km/h,
- autonomie : 2 heures d'utilisation continue,
- capacité d'accueil : 2 personnes,
- conduite manuelle ou automatique.

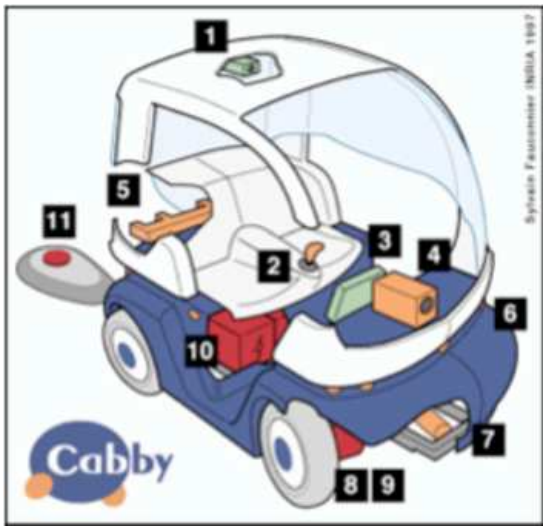

FIG. 7.2 – *Architecture d'un CyCab*

La figure 7.2 montre une vue en coupe de l'architecture du CyCab qui est constituée de :

- 1 ensemble de batteries avec un gestionnaire automatique de charge (10) et un bouton arrêt d'urgence qui est soit de type poussoir (2) soit de type radio-commandé (1),
- 2 cartes électroniques (5) et (6) d'acquisition de données comprenant chacune un microprocesseur 32 bit PowerPC (appelés MPC555). Chaque carte permet de contrôler 2 roues du CyCab. Nous reviendrons plus tard sur l'architecture de ces cartes que l'on appellera par la suite noeuds ;
- 1 PC embarqué au format rack (taille 2U), placé sous le siège (2), possédant un processeur Intel cadencé à 3 GHz, avec un Linux temps réel, RTAI. L'ensemble est alimenté par une tension d'entrée de -48V (350W). L'écran est situé en (3) ;

– 2 bus CAN indépendants : le bus CAN 0 permet la communication entre les 2 MPC555 et le PC embarqué, alors que le bus CAN 1 permet d'ajouter d'éventuels futurs composants électroniques,
– 4 moteurs et leurs freins électriques (8) et (9) commandés par 4 contrôleurs de moteur appelés Curtis PMC 1227 (9) servant d'amplificateurs de puissance pour contrôler la vitesse des roues. La consigne de vitesse est donnée par une tension de 0 à 5V aux Curtis qui fourniront des signaux PWM adéquats aux moteurs. Les Curtis protègent les noeuds des contre-courants des moteurs, quand par exemple, on les arrête brusquement ;
– 4 codeurs incrémentaux donnant la vitesse des roues (8),
– 1 vérin de direction électrique alimenté par signal PWM (7) faisant pivoter les 4 roues,
– 1 encodeur absolu avec sortie SPI et donnant l'angle des roues,
– 1 joystick (2) fournissant deux courants indiquant : la consigne de vitesse des roues et la consigne de direction des 4 roues,
– depuis l'année dernière, le CyCab possède une caméra type webcam (4) se branchant sur un port FireWire du PC embarqué.

7.2.2 Architecture

La figure 7.3, montre en formalisme UML, l'architecture complète du CyCab avec ses moyens de communication, à savoir les deux noeuds, le PC embarqué et les deux bus CAN.

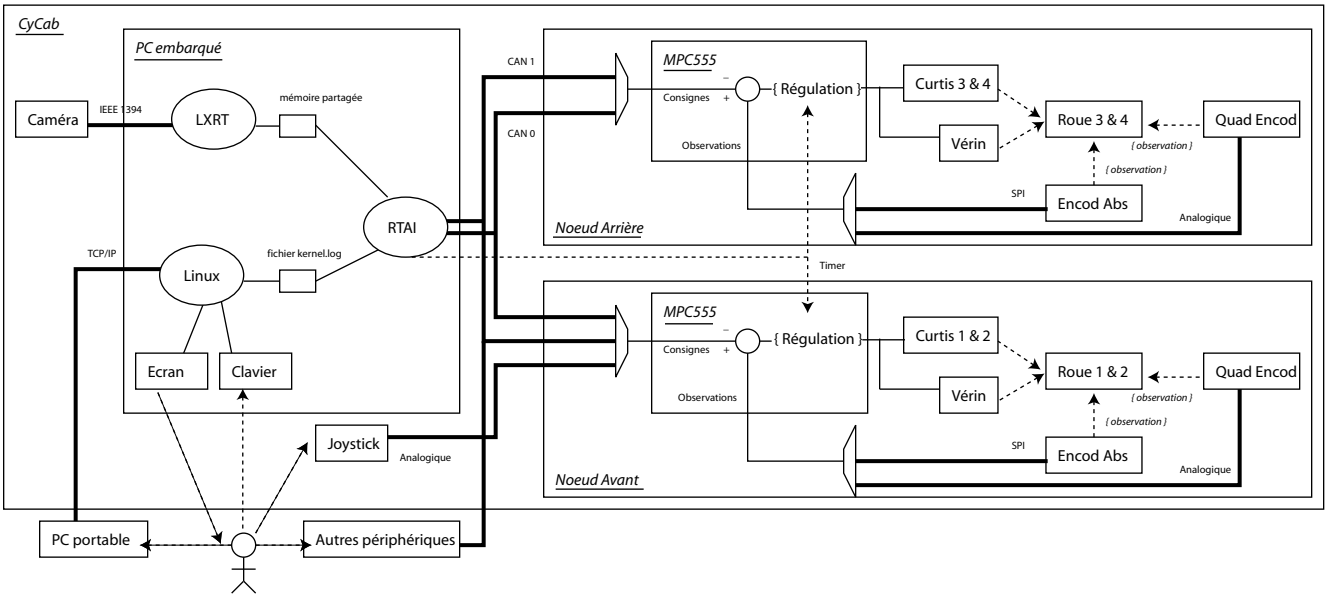

FIG. 7.3 – *Architecture d'un CyCab en formalisme UML*

SynDEx va distribuer le programme de conduite sur les 2 noeuds et sur le PC embarqué qui communiqueront grâce au bus CAN 0. Le passager (ou acteur en formalisme UML) du CyCab peut

communiquer avec le PC embarqué (clavier/souris/écran, USB, FireWire, Ethernet, CAN 1) mais n'a accès aux noeuds que par l'intermédiaire du bus CAN 1. Une partie du programme de conduite (manuel ou automatique) va tourner sur un Linux temps réel (RTAI) servant essentiellement de timer 10 ms aux deux noeuds MPC. La partie LXRT va gérer les images issues de la caméra, appliquer le traitement d'image de détection de CyCabs et communiquer avec RTAI via une mémoire partagée. Un programme Linux peut observer le flot de données pour, par exemple, le rejouer en simulation. Le programme de conduite s'exécute sur les deux noeuds. Il lit les données fournies par les capteurs (direction des roues, vitesse des roues, ...). Il calcule l'asservissement et envoie le résultat aux différents actionneurs gérant les quatre roues du CyCab (traction et direction).

7.2.3 PC embarqué

Pour faire du traitement d'image, et donc par conséquent, de faire du suivi automatique basé sur une caméra, le PC embarqué du CyCab est constitué de :

- le même châssis alimenté par une tension -48 V.
- une carte mère contenant un Pentium 4 cadencé à 3 GHz et 512 Mo de mémoire vive.
- une carte de fond de panier avec 4 slots PCI.
- une carte SPI pour la communication CAN.
- une carte SPI pour la communication avec la caméra FireWire.
- une carte graphique PCI NVidia GX 5200, la puce graphique incluse dans la carte mère est suffisante pour un affichage confortable.
- un disque dur IDE de 20 Go.

7.2.4 Caméra FireWire

La caméra FireWire que nous utilisons est une Fire-I fabriquée par UniBrain, sa taille est 36×65 mm. Voici les caractéristiques techniques 1 :

- interface : FireWire 400 Mbps, 2 ports (6 broches),
- capteur : Sony Wfine 1/4 CCD Color,
- résolution : VGA 640 x 480,
- vidéo : YUV, RGB, Monochrome,
- fréquence : 30, 15, 7.5 et 3.5 images par seconde,
- l'angle de vue horizontale : 42ř,
- l'angle de vue verticale : 32ř.

Nous utilisons les librairies libraw1394 et libdc1394 pour obtenir les images. Les configurations que nous avons choisies sont : format YUV422, taille 320×240, 7.5 images par seconde. Ces options peuvent facilement être modifiées. Les termes FireWire, IEEE 1394 ou i.Link, sont synonymes. Le bus FireWire utilisé est un bus série plug and play, pouvant supporter jusqu'à 63 périphériques et permettant différentes topologies alors que l'USB ne permet que des configurations en étoile. Le débit maximum de la norme IEEE 1394 le plus répandu est de 400 Mbit/s, ce qui permet de fournir des débits plus élevés que celui de la norme USB 2.0. Il existe 2 modes distincts

pour le bus IEEE 1394 : le mode asynchrone et le mode isochrone. Le mode isochrone est le plus rapide car il permet l'envoi de paquets de tailles fixes à intervalles de temps réguliers et sans avoir besoin d'accusé de réception. Même si le débit théorique est de 400 Mbit/s, le cadencement des paquets d'une transmission isochrone fait que le débit utile est de 256 Mbit/s. On fera donc attention à ne pas dépasser ce débit maximum si on veut brancher, deux caméras sur une même carte pour faire, par exemple, du traitement d'image stéréo. Pour cela on jouera sur les paramètres suivants : la taille de l'image, son format, son taux de compression, le nombre d'images par seconde. Par exemple, une image YUV422 de taille 640×480 à 30 images par seconde aura un débit de 147 Mbit/s.

7.3 Algorithme de suivi pour train de CyCab

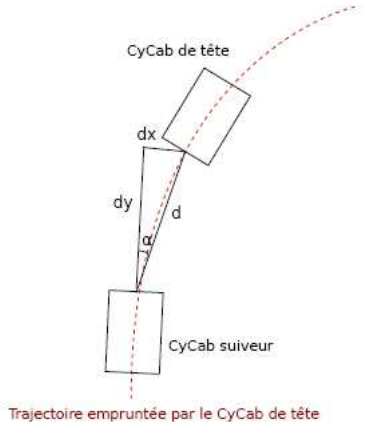

FIG. 7.4 – *Les informations nécessaires à déterminer*

On se met dans le cas d'une chaîne de 2 CyCabs formé de la voiture de tête (qu'on appelle aussi leader) et la voiture suiveuse. Le principe sera le même pour des chaînes de plus de 2 CyCabs. Comme il existe plusieurs types de CyCabs nous supposons que les CyCabs ont tous les mêmes caractéristiques. La conduite du CyCab leader ne nous intéresse pas car elle est pilotée par une personne. Nous supposons aussi que le seul capteur utilisé est la caméra décrite plutôt. Le graphe d'algorithme de l'application de suivi pour train de CyCab est composé de trois blocs (opérations) principaux :

– la partie traitement d'image : elle consiste à extraire les informations utiles sur le CyCab de tête à partir de chaque image de la séquence venant de la webcam. Cette procédure est précédée par le repérage du CyCab leader en détectant ses contours. Une fois que le CyCab leader est détecté nous calculons l'inter-distance des deux CyCabs *d* ainsi que l'angle de déportation α (voir figure 7.4) ;

– la partie contrôle ou asservissement longitudinal et latéral qui, suivant les données envoyées par la partie traitement d'image, permet au CyCab de suivre un autre CyCab en ligne droite et en virage,
– la partie affichage qui sert à afficher sur l'écran du CyCab différentes informations comme : les images transmises par la caméra et les outils pour la détection.

7.3.1 Communication entre les modules LXRT et RTAI

Avant d'aborder la communication entre ces deux modules nous allons tout d'abord donner leurs définitions.

D'un côté RTAI est un ajout à Linux qui permet de faire de la programmation temps réel, c'est-à-dire, en garantissant les échéances des opérations. Concrètement, RTAI est fondé sur un ordonnanceur temps réel qui considère Linux comme l'opération à exécutée la moins importante. En conséquence, toutes les opérations qui ne sont pas explicitement temps réel ne seront exécutées que si les opérations temps réel leur en laissent le temps. Ainsi le système d'exploitation Linux perturbe le moins possible le déroulement des opérations temps réel. Le développement sous RTAI se fait en général dans l'espace noyau. L'espace noyau dans Linux est la couche logicielle du système d'exploitation, où sont regroupés, entre autres, tous les drivers. Le plus gros inconvénient de travailler dans l'espace noyau est que des librairies comme celles en C, nécessaires au traitement d'image et au fonctionnement de la caméra, ne sont pas utilisables.

D'un autre côté LXRT est un module RTAI spécialement développé afin d'écrire des programmes temps réel pour l'espace utilisateur, ce qui permet de bénéficier des atouts de RTAI et d'une réserve plus conséquente de librairies.

Par conséquent une fois que nous savons comment obtenir les consignes de vitesse et d'angle, il faut pouvoir, depuis le module RTAI, les envoyer aux MPC555 afin qu'ils puissent les exploiter dans l'asservissement. Il n'est pas possible, donc, de faire le traitement d'image dans une tâche temps réel RTAI à cause de certaines librairies manquantes (driver de la webcam, du FireWire, etc.) ce qui nous conduit à l'introduire dans une tâche qui sera exécutée par le module LXRT. Il partage, donc, le processeur du PC embarqué avec les opérations du module RTAI, à travers lequel il communique avec les autres processeurs via le bus CAN.

Dans la figure 7.5, le partie gauche du rectangle en pointillé représente la tâche LXRT qui permet de gérer la caméra FireWire et la détection de CyCab. Les résultats du traitement d'image, c'est-à-dire les consignes de vitesse et d'angle, sont stockées dans une mémoire partagée. La partie droite du rectangle en pointillé représente la tâche RTAI. Les blocs qu'elle contient sont les blocs que l'on retrouve dans l'application SynDEx de conduite automatique (Figure 7.9). Le sémaphore joue le rôle de mutex pour gérer les accès concurrents de lecture et d'écriture dans la mémoire partagée.

La mémoire partagée est une zone mémoire qui est désignée par un même identifiant dans chacune des parties concernées. Elle n'est soumise à aucun contrôle, donc n'importe quel programme qui y a accès peut écrire dans ou lire le contenu de cette mémoire à n'importe quel moment, et n'impose donc aucune contrainte de synchronisation, et c'est justement la raison pour laquelle un tel mode de communication a été utilisé. Il faut savoir que la partie traitement d'image est très longue

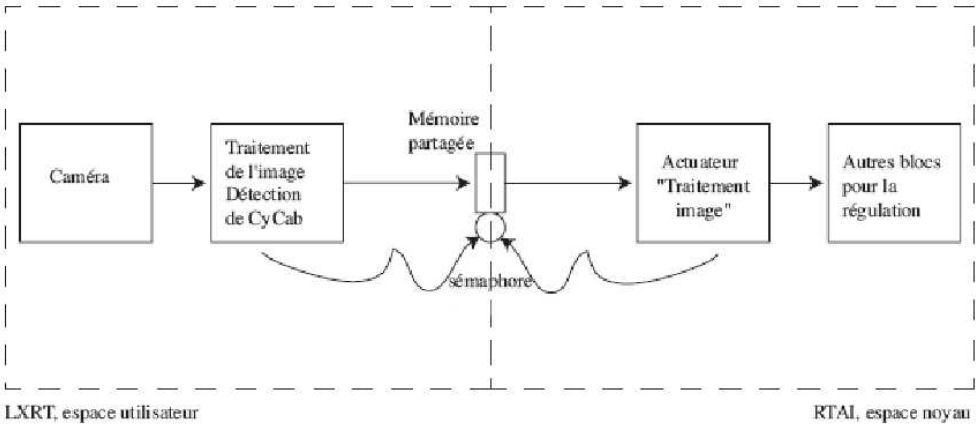

FIG. 7.5 – *Communication entre les modules LXRT et RTAI*

par rapport au reste de l'application de suivi - 0,5 seconde pour la période du traitement d'image contre 10 millisecondes pour celle de la partie commande - et comme SynDEx monopériode ne gère pas ce genre de situation toutes les tentatives de synchronisation seraient inutiles.

7.3.2 Application CycabVitessAutoMultiPeriods

De toutes les applications liées au CyCab celle-ci est la première version multipériode qui existe. Dans les précédentes versions toutes les opérations de chacun des trois blocs composant le graphe d'algorithme (voir la figure 7.6) ont la même période.

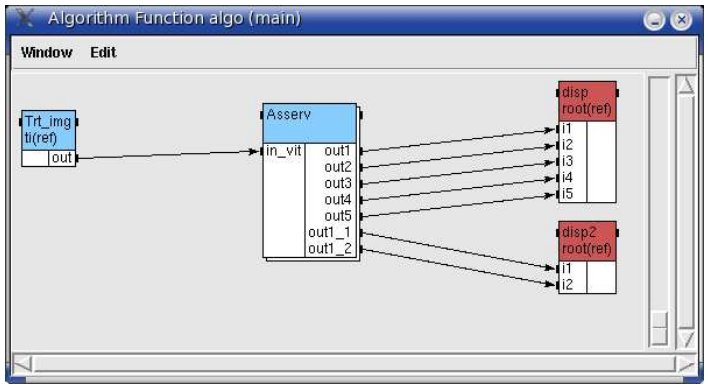

FIG. 7.6 – *Graphe d'algorithme de l'application CycabVitessAutoMultiPeriods*

Désormais, dans le graphe d'algorithme (voir la figure 7.6) "Trt_img" s'exécute à une période égale à 5O fois la période de "Asserv". Dès lors les données produites par l'exécution d'une instance de "Trt_img" servent à exécuter 50 instances de "Asserv". Cette dépendance permet de synchroniser la communication entre ces deux blocs puisque on sait que c'est toujours la première

instance de “Asserv” qui reçoit les nouvelles consignes de vitesse et d’angle puis ces mêmes valeurs vont être diffusées au restant des instances de “Asserv”. Par conséquent on maîtrise l’instant à laquelle les résultats du traitement d’image présents dans la partie LXRT peuvent être récupérés par les opérations présentes dans la partie RTAI.

Par ailleurs on a choisi de mettre le troisième bloc (la partie affichage qui se compose des opérations “disp” et “disp2”) à une période égale à celle du premier bloc (“Trt_img”). À savoir que comme :

– la première partie de l’affichage sert essentiellement avant le suivi à faire l’acquisition des contours du CyCab leader pour la détection, ensuite elle ne sert qu’a retransmettre les images captées par la caméra,
– la deuxième partie sert à communiquer au passager du CyCab des informations telles que la vitesse du CyCab, l’angle de rotation des roues, etc.

On a préféré les exécuter une fois toutes les 0.5 seconde pour éviter de surcharger le PC embarqué (c’est le processeur sur lequel elles s’exécutent) avec des opérations de moindre priorité. Sous la version monopériode, cette opération s’exécutait obligatoirement à la même période que l’asservissement, c’est-à-dire 10 milliseconds, simplement parce qu’on ne pouvait pas faire autrement. Ceci dit nous devons préciser qu’aucune donnée produite par les 50 instances de “Asserv” n’est perdue (voir la section 2.1.1.2) et que c’est à l’utilisateur de choisir soit de les afficher toutes ou quelques unes, soit de procéder à des calculs comme ceux de la moyenne ou de l’écart-type entre autres.

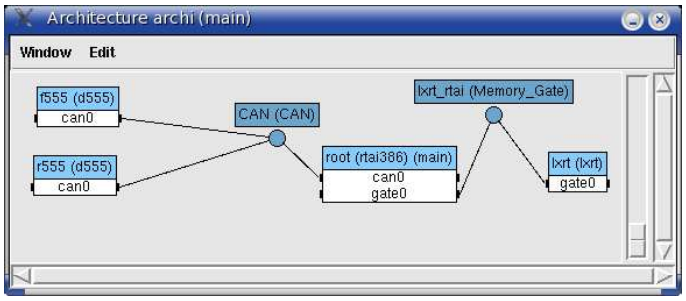

FIG. 7.7 – *Graphe d’architecture de l’application CycabVitessAutoMultiPeriods*

La figure 7.8 présente le résultat de la distribution et l’ordonnancement des opérations du graphe d’algorithme de la figure 7.6 sur les processeurs de l’architecture de la figure 7.7. Pour des raisons de visibilité on a choisi de montrer un exemple où la période du bloc “Trt_img” est égale à seulement deux fois la période du bloc “Asserv” (au lieu de 50 fois). Le but est, uniquement, de faire apparaître les dépendances entre l’instance de “Trt_img” et les instances des opérations concernées appartenant au bloc “Asserv” sachant que dans le cas réel de l’application ces dépendances ne sont pas visibles sur le diagramme à cause de la différence entre la durée d’une communication entre LXRT et RTAI et le temps nécessaire à l’exécution de toutes les opérations du bloc “Asserv” ainsi que leurs instances. Cette dépendance est visible à travers sont coût de communication (en orange) qui se trouve sur le médium de communication “lxrt_rtai”. La date de

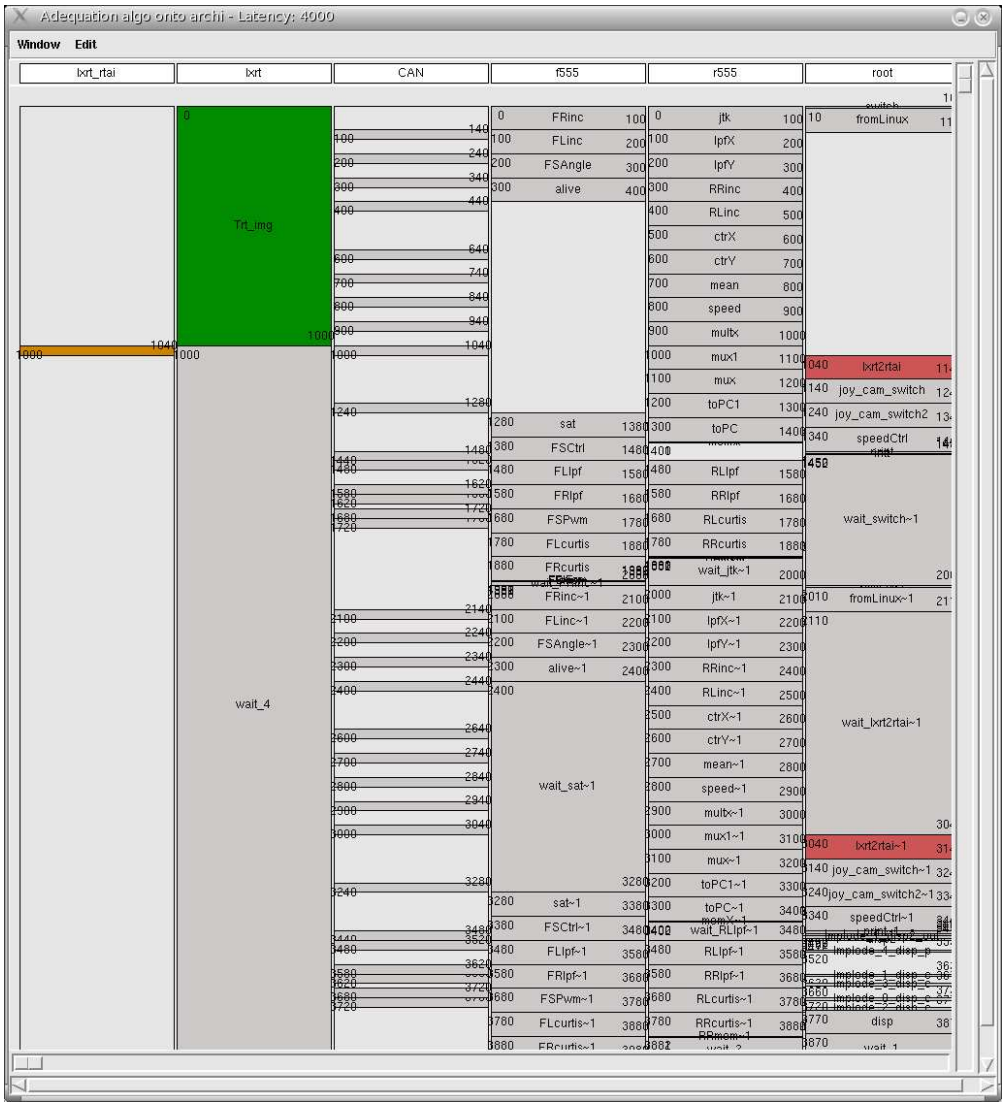

FIG. 7.8 – *Diagramme temporel résultant de l'adéquation*

départ de cette communication correspond à la date de fin d'exécution de l'opération "Trt_img" (en vert) sur le processeur "lxrt" tandis que la date de fin de la communication correspond à la date de début d'exécution de la première instance de l'opération "lxrt2rtai" (en rouge, de même pour la deuxième instance) qui est exécutée sur le processeur "rtai".

La figure 7.9 expose les opérations contenues dans le bloc "Asserv" étant donné que c'est une opération hiérarchique.

Voici les rôles des principales opérations qui participent à l'asservissement longitudinal et latéral permettant de faire du suivi en ligne droite et en virage :

– jtk : convertisseur Analogique-Numérique du joystick,

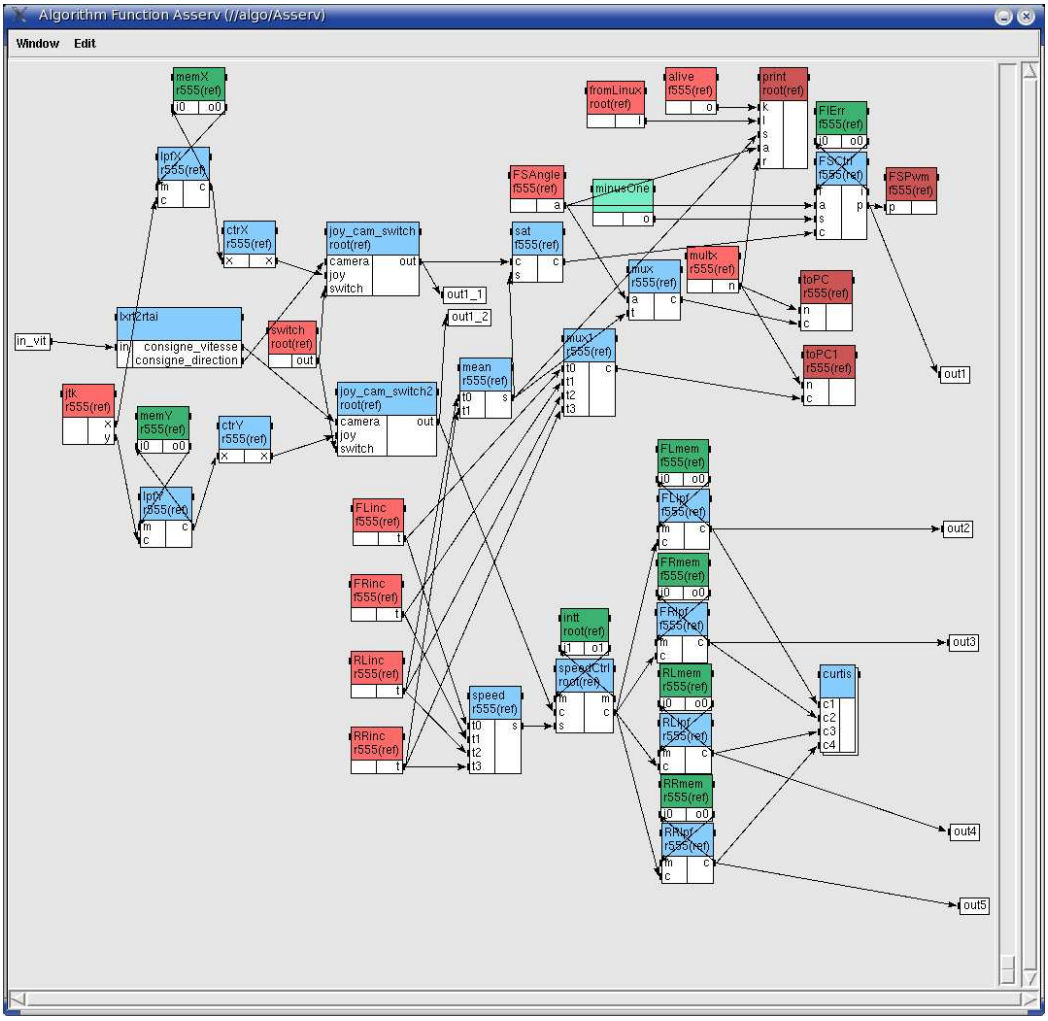

FIG. 7.9 – *L'opération Asserv*

- lpfX + memX : filtre passe bas avec une fréquence de coupure de 10Hz. C'est pour éviter que l'asservissement ne réagit au bruit de conversion. Le grand X signifie qu'il s'agit du mouvement horizontal du joystick (la direction),
- ctrX : Au repos la valeur du joystick (sur l'axe de direction) vaut 440. Cette opération sert tout simplement à mettre cette valeur à 0. On aura donc en sortie une consigne comprise entre [-293, +293],
- ctrY : comme sur l'axe de direction, au repos la valeur du joystick (sur l'axe de traction) vaut 440. Cette opération sert tout simplement à mettre cette valeur à 0. On aura donc en sortie une consigne comprise entre [-300, +300] ;
- lxrt2rtai : lecture des valeurs résultant du traitement d'image et qui consistent en la consigne de direction et la consigne de vitesse,
- switch : un drapeau pour les différents modes de fonctionnement : 0=conduite manuelle et

1=conduite automatique,

- joy_cam_switch : suivant la valeur de “switch” (mode manuel ou auto), cette opération recopie l’entrée “camera” ou “joy” vers la sortie “out”,
- joy_cam_switch2 : la même opération que joy_cam_switch mais pour la consigne de direction.
- FLinc : encodeur incrémental de la roue avant gauche. Il sert à calculer la vitesse angulaire de la roue,
- FRinc : idem pour la roue avant droite,
- RLinc : idem pour la roue arrière gauche,
- RRinc : idem la roue arrière droite,
- speed : elle calcule la vitesse angulaire moyenne des quatre roues pour obtenir la vitesse du CyCab par la suite,
- mean : cette opération calcule la vitesse moyenne des roues arrières fournies à l’opération sat,
- sat : cette opération calcule l’angle de braquage max suivant la vitesse du véhicule et ensuite fait la saturation de la consigne de direction,
- FSAngle : encodeur absolu pour les roues avant. Elle sert à déterminer l’angle de braquage des roues avant,
- FSCtrl + FIErr : régulateur de type PI pour la direction,
- FSPwm : elle sert à générer le signal PWM envoyé à l’actionneur,
- speedCtrl + initt : régulateur de type PI pour la vitesse.
- FLlpf + FRlpf + RLlpf + RRlpf : filtre passe bas avec une fréquence de coupure de 10Hz. Cette opération sert à supprimer des variations en hautes fréquences ;
- curtis : actionneurs pour les 4 roues.

La série de tests effectuée a été satisfaisante dans des conditions favorables sans trop de perturbations comme les poteaux, les arbres, les bordures de bâtiments, etc. Ce problème est du à la non-robustesse de l’algorithme de suivi de CyCab qui reste à améliorer.

Conclusion

La partie développement logiciel constitue la concrétisation dans le logiciel SynDEx des résultats proposés dans la partie théorique concernant le problème d'ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence et de périodicité stricte.

Dans l'état de l'art est présentée une première série d'outils pour la conception électronique au niveau système. En explorant les principales caractéristiques de ces outils on s'est rendu compte que la quasi totalité ne proposent pas, comme le fait SynDEx, la distribution et l'ordonnancement automatique optimisé. Pour faire face à ce manque l'utilisateur doit soit le faire à la main, soit faire appel à d'autres types d'outils. Dans la suite de l'état de l'art est présentée une deuxième série d'outils pour étudier l'ordonnançabilité en utilisant en général des algorithmes d'ordonnancement temps réel préemptifs classiques comme RM, DM, EDF, etc.

Le fait que l'ordonnancement multiprocesseur de tâches non-préemptives avec contraintes de précédence et de périodicité stricte soit peu étudié dans la littérature, explique le manque d'outils sur ce sujet. On donc a été amené à modifier l'ancienne version de SynDEx en y intégrant la condition d'ordonnançabilité et l'heuristique d'ordonnancement proposées dans le chapitre 2. En plus des fonctionnalités de l'ancienne version le logiciel résultant réalise, sous les contraintes citées auparavant, une étude d'ordonnançabilité ainsi que l'ordonnancement, tout en minimisant le temps d'exécution de toutes les tâches du système. Ainsi cette nouvelle version s'inscrit dans les deux catégories d'outil évoquées dans l'état de l'art, et constitue un pas de plus vers un outil idéal. La nouvelle version de SynDEx permet aux utilisateurs de traiter des applications multipériode en effectuant de l'analyse d'ordonnançabilité comme cela est fait dans d'autres outils.

Le premier véritable test auquel SynDEx multipériode a été confronté est l'application de suivi pour train virtuel de CyCabs qui consiste à ce qu'un CyCab puisse suivre automatiquement (sans conducteur). Cette application a été testée dans les conditions réelles et a montré des résultats satisfaisants.

Conclusion générale et perspectives

Les travaux de cette thèse s'inscrivent dans le domaine de l'ordonnancement multiprocesseur de systèmes temps réel ainsi que celui de la conception des logiciels temps réel embarqué. Dès lors les résultats présentés dans ce manuscrit lui permettent d'exposer cette double compétence.

Les tâches temps réel considérées sont des tâches périoques strictement avec échéance sur requète, non-préemptives, non-concrètes, avec contraintes de précédence et de latence entre elles. L'ordonnancement considéré est un ordonnancement multiprocesseur hors-ligne.

Le problème d'ordonnancement de tâches temps réel avec les contraintes de précédence, de périodicité stricte et de latence étant complexes, nous avons choisi, pour avoir des résultats d'ordonnançabilité et d'optimisation dans un temps raisonnable pour des applications réalistes, un algorithme glouton. Contrairement aux algorithmes exacts qui misent énormément sur la puissance des machines pour espérer résoudre des problèmes et aux méta-heuristiques qui sont des méthodes prédéfinies et prêtent à l'emploi, les algorithmes gloutons s'appuient sur une bonne maîtrise du problème pour le résoudre.

Les différentes études d'ordonnançabilité et l'algorithme d'ordonnancement proposés constituent l'apport principal de cette thèse. Désormais nous somme capable de traiter des systèmes de tâches temps réel avec les contraintes suivantes :

- précédence due à la dépendance : les tâches dépendantes, qu'elles soient à la même ou à des périodes différentes, échangent les données sans qu'il y ait de pertes,
- la périodicité stricte : les tâches sont ordonnancées de manière à pouvoir se répéter à des intervalles stricts suivant leurs périodes,
- la latence : les tâches sont ordonnancées de manière à ce que toutes les paires de tâches soumises à des contraintes de latence les respectent, c'est-à-dire que le délai entre le début d'une tâche et la fin de l'autre tâche d'une paire soit inférieur au temps de cette latence.

La figure 7.3.2 illustre les deux domaines dans lesquels le logiciel SynDEx peut figurer. Les frontières entre ces domaines sont difficiles à cerner en raison de leurs complémentarités.

Enfin le travail qui a été effectué s'ouvre à un certain nombre de perspectives que nous citons ci-après. L'extension du modèle nous semble être le travail futur le plus immédiat. Dans le modèle adopté dans ce manuscrit la contrainte de périodicité prise en compte est stricte pour les raisons qui sont rapportées dans le deuxième chapitre. Néanmoins le fait que la périodicité classique demeure une contrainte très répandue incite à réfléchir à un modèle où une tâche peut, soit être strictement périodique, soit être périodique dans le sens classique du terme.

On parle aussi dans ce manuscrit de dépendance sans pertes de données. Il est dit aussi que ce mode de dépendance entraîne des restrictions au niveau du graphe d'algorithme comme celle

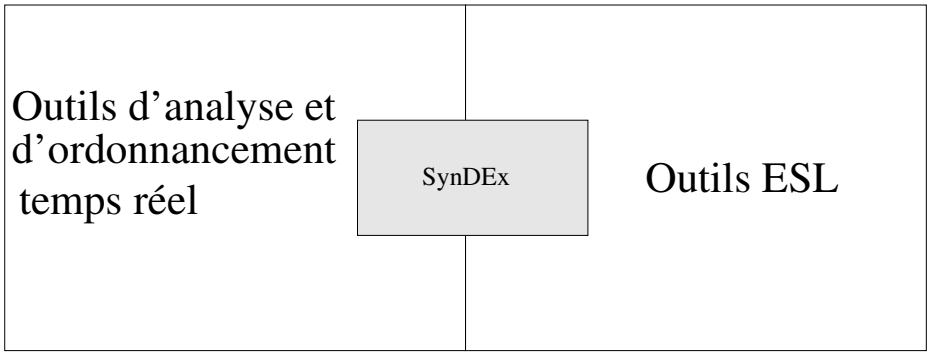

FIG. 7.10 – *Les différents domaines d'application de SynDEx*

qui n'autorise la dépendance qu'entre les tâches avec les mêmes périodes ou avec des périodes multiples. L'idée serait de réfléchir à un modèle avec un mode de dépendance qui supporte les pertes de données tout en garantissant le bon fonctionnement du système.

Une autre extension très importante concerne la prise en compte de la préemption peut permettre de trouver plus de systèmes ordonnançables. Cependant cette extension n'aura sa raison d'être qu'une fois que le coût de la préemption sera mieux maîtrisé afin de fournir le même niveau de sûreté de l'exécution des systèmes que dans le contexte non-préemptif.

Étant donné que la plupart des algorithmes proposés sont des algorithmes gloutons, ceux-ci peuvent être réutilisés, en effectuant les transformations nécessaires, pour faire de l'ordonnancement en-ligne ou mixte hors-ligne et en-ligne. En effet les algorithmes d'ordonnancement en-ligne doivent avoir une complexité très faible, ce qui est le cas des algorithmes proposés.

Au niveau du logiciel SynDEx, nous avons intégré l'heuristique d'ordonnancement et avons implémenté l'algorithme du Branch&Cut présentés dans le chapitre 2 mais cela n'a pas été le cas pour l'algorithme d'équilibrage de charges et de mémoire qui a seulement été évalué théoriquement.

D'un autre côté comme il existe déjà une version monopériode tolérante aux fautes, il serait intéressant d'en faire une version multipériode.

Bibliographie

[1] Jay M. Hyman, Aurel A. Lazar, and Giovanni Pacifici. Real-time scheduling with quality of service constraints. *IEEE Journal of Selected Areas in Communications*, 9(7):1052–1063, 1991.

[2] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[3] J. Y-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3):115–118, 1980.

[4] J. Orozco, R. Cayssials, J. Santos, and E. Ferro. Precedence constraints in hard real-time distributed systems. *iceccs*, 00:33, 1997.

[5] R. Kocik. *Sur l'optimisation des systèmes distribués temps réel embarqués : application au prototypage rapide d'un véhicule électrique semi-autonome*. PhD thesis, Université de Rouen, Spécialité informatique industrielle, 22/03/2000.

[6] C-H. Lin and C.-J. Liao. Makespan minimization for multiple uniform machines. *Comput. Ind. Eng.*, 54(4):983–992, 2008.

[7] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[8] J. Zhu, T. G. Lewis, W. Jackson, and R. L. Wilson. Scheduling in hard real-time applications. *IEEE Softw.*, 12(3):54–63, 1995.

[9] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[10] P. Meumeu Yomsi and Y. Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Proceedings of 19th Euromicro Conference on Real-Time Systems, ECRTS'07*, Pisa, Italy, July 2007.

[11] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Tech. Rep. UNC-CS TR01-016*, Departement of Computer Science, University of North Carolina at Chapel Hill, May 2001.

[12] S. Pallier. *Analyse Hors Ligne d'Ordonnançabilité d'Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques*. Docteur en informatique, École Nationale Supérieure de Mécanique et d'Aérotechnique, 2006.

[13] K. Tindell. Deadline monotonic analysis. *Embedded Systems Programming*, 13(6):20–38, 2000.

[14] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGADA Conference, Atlanta*, November 2004.

[15] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *SEUS*, pages 263–272, 2007.

[16] O.O Roux and A. M. Deplanche. A t-time petri net extension for real-time task scheduling modelling. *JESA Modelling of Reactive Systems*, 36(7):973–986, 2002.

[17] Manuel Serrano and Hans-J. Boehm. Understanding memory allocation of scheme programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 245–256, New York, NY, USA, 2000. ACM.

[18] Andrew W. Appel. Book review: Garbage collection: Algorithms for automatic dynamic memory management by richard jones and rafael lins, john wiley & sons, 1996. *J. Funct. Program.*, 7(2):227–229, 1997.

[19] T. Higuera, V. Issarny, M. Banâtre, and F. Parain. Memory management for real-time java: An efficient solution using hardware support. *Real-Time Syst.*, 26(1):63–87, 2004.

[20] N. Ventroux. *Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture*. PhD thesis, 2006. Université de Rennes.

[21] F. Parain. *Ordonnancement sous contraintes énergétiques d'applications multimédia sur une plate-forme multiprocesseur hétérogène*. PhD thesis, 2002. Université de Rennes.

[22] A. B. Abril Garcia. *Estimation et optimisation de la consommation dans les description architecturales des systèmes integrés complexes*. PhD thesis, 2005. Université de Paris 6.

[23] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[24] D. Decotigny. *Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel*. PhD thesis, Université de Rennes 1, 2003.

[25] C. L.Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

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

[27] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(237-250), 1982.

[28] A.K. Mok. *Fundamental Design Problems for the Hard Real-Time Environments*. PhD thesis, MIT, 1983.

[29] A. K. Mok and M. L. Detouzos. Multiprocessor scheduling in a hard real-time environment. In *7th IEEE Texas Conf. on Computing Systems*, USA, 1978. IEEE.

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

[31] F. Bimbard. *Dimensionnement temporel de systèmes embarqués : application à OSEK*. PhD thesis, Conservatoire National des Arts et Métiers, 2007.

[32] R. R. Howell. and M. K. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle times. *Inf. Comput.*, 117(1):50–62, 1995.

[33] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12 th IEEE Symposium on Real-Time Systems*, pages 129–139, December 1991.

[34] Y. Cai and M. C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[35] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. *Rapport de Recherche RR-2966, INRIA*, 1996.

[36] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, Department of Computer Science, University of York, 1991.

[37] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. In *JPL Space Programs Summary 37-60*, volume 2, pages 28–37, November 1969.

[38] J. Goossens. Introduction à l'ordonnancement temps réel multiprocesseur. In *école d'été temps réel*, 2007.

[39] K. Danne and M. Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. *SIGPLAN Not.*, 41(7):93–102, 2006.

[40] J. M. Lopez, J. L. Diaz, and D. F. Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 67, Washington, DC, USA, 2001. IEEE Computer Society.

[41] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. *ecrts*, 00:25, 2000.

[42] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[43] J. H. Anderson. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[44] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[45] K. Jung and C. Park. A technique to reduce preemption overhead in real-time multiprocessor task scheduling. In *Asia-Pacific Computer Systems Architecture Conference*, pages 566–579, 2005.

[46] J. Sun and J. W. S. Liu. Bounding the end-to-end response time in multiprocessor real-time systems. In *WPDRTS '95: Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, page 91, Washington, DC, USA, 1995. IEEE Computer Society.

[47] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[48] K. Tindell. Adding time-osets to schedulability analysis. Technical report, Department of Computer Science, University of York, 1994.

[49] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.

[50] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, 2006.

[51] N. Fisher and S. Baruah. The feasibility of general task systems with precedence constraints on multiprocessor platforms. *Real-Time Systems*.

[52] Y-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[53] M. A. Bender and C A. Phillips. Scheduling dags on asynchronous processors. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 35–45, New York, NY, USA, 2007. ACM.

[54] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An integrated approach for processor allocation and scheduling of mixed-parallel applications. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 443–450, Washington, DC, USA, 2006. IEEE Computer Society.

[55] Azeddien M. Sllame and Vladimir Drabek. An efficient list-based scheduling algorithm for high-level synthesis. In *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 316, Washington, DC, USA, 2002. IEEE Computer Society.

[56] M. Singer. Decomposition methods for large job shops. *Comput. Oper. Res.*, 28(3):193–207, 2001.

[57] D. Dor and M. Tarsi. Graph decomposition is npc - a complete proof of holyer's conjecture. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 252–263, New York, NY, USA, 1992. ACM.

[58] L. Cucu. *Ordonnancement non préemptif et condition d'ordonnançabilité pour système embarqués à contraintes temps réel*. PhD thesis, Université de Paris Sud, Spécialité Électronique, 28/05/2004.

[59] P. Meumeu Yomsi and Y. Sorel. Non-schedulability conditions for off-line scheduling of real-time systems subject to precedence and strict periodicity constraints. In *Proceedings of 11th IEEE International Conference on Emerging technologies and Factory Automation, ETFA'06, WIP*, Prague, Czech Republic, September 2006.

[60] L. Cucu and Y. Sorel. Condition d'ordonnançabilité pour systèmes temps réel non-préemptif à contraintes de précédences, de périodicités et de latences. In *Actes du 6ème congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision, ROADEF'05*, Tours, France, February 2005.

[61] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of 10th Real-Time Systems Conference, RTS'02*, Paris, France, March 2002.

[62] N. Pernet. *Implantation distribuée temps réel de programmes conditionnés à l'aide d'ordonnancements mixtes hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques*. PhD thesis, Université de Paris 6, Spécialité Informatique, 07/07/2006.

[63] L. Cucu and Y. Sorel. Non-preemptive scheduling algorithms and schedulability conditions for real-time systems with precedence and latency constraints. Research Report RR-5403, INRIA, Rocquencourt, France, 2004.

[64] L. Cucu and Y. Sorel. Periodic real-time scheduling: from latency-based model to deadline-based model. In *Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications, MISTA'05*, New-York, USA, July 2005.

[65] L. George, P. Muhlethaler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. *Rapport de Recherche RR-2516, INRIA*, 1995.

[66] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

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

[68] 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(12):745–758, 1997.

[69] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions, on Parallel Distibuted Systems*, 19(2), February 1993.

[70] F. Benhamou. Principles and practice of constraint programming. In *CP*, volume 4204 of *Lecture Notes in Computer Science*. Springer, 2006.

[71] http://www.mozart oz.org/.

[72] K. Schild and J. Wurtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, 2000.

[73] H. Thane and M. Larsson. Scheduling using constraint programming. Technical Report, Malardalen University, June 1997.

[74] C. Ekelin and J. Jonsson. Solving embedded systeme scheduling problems using constraint programming. Departement of Computer Engineering, Chalmers University of Technology SE-412 96 Göbeborg, Sweden.

[75] P-E. Hladik, H. Cambazard, A-M. Deplanche, and N. Jussien. Solving a real-time allocation problem with constraint programming. *J. Syst. Softw.*, 81(1):132–149, 2008.

[76] K. Subramani. Periodic linear programming with applications to real-time scheduling. *Mathematical. Structures in Comp. Sci.*, 15(2):383–406, 2005.

[77] N. Maculan T. Davidovic, L. Liberti and N. Mladenovic. Mathematical programming-based approach to scheduling of communicating tasks. Technical Report G-2004-99, Cahiers du GERAD, 2004.

[78] R. J. Vanderbei. *Linear Programming: Foudations and Extensions*. Second edition edition, 2001.

[79] G. B. Dantzig and M. N. Thapa. *Linear Programming 2: Theory and Extensions*. Springer-Verlag, New York, 2003.

[80] http://www.ilog.com/products/cplex/.

[81] P. Hansen and N. Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467(19), 1 May 2001.

[82] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[83] T. F. Abdelzaher and K. G. Shin. Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers*, 49-1:81–87, January 2000.

[84] G. J. Woeginger. Exact algorithms for np-hard problems: a survey. pages 185–207, 2003.

[85] G. J. Woeginger. Open problems around exact algorithms. *Discrete Appl. Math.*, 156(3):397–405, 2008.

[86] A. Jakoby, M. Liskiewicz, and R. Reischuk. Dynamic process graphs and the complexity of scheduling. *Electronic Colloquium on Computational Complexity (ECCC)*, (090), 2001.

[87] M. Widmer, A. Hertz, and D. Costa. *Les Métaheuristiques, chap. 3 of Ordonnancement de la Production*. Hermes Science Publications, 2000.

[88] Q. Rong, E.K. Burke, B. McCollum, L.T.G. Merlot, and S.Y. Lee. A survey of search methodologies and automated approaches for examination timetabling. Computer Science Technical Report NOTTCS-TR-2006-4, School of Computer Science and Information Technology, University of Nottingham, 2006.

[89] C. Duhamel. *Un Cadre Formel pour les Méthodes par Amélioration Itérative , Application à deux problèmes d'Optimisation dans les Réseaux*. Docteur en informatique, Université Blaise Pascal Clermont-Ferrand II, 2001.

[90] P. Calegari, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5(2):145–158, 1999.

[91] E. D. Taillard. Principes d'implémentation des métaheuristiques. *Chapter 2 of Optimisation approchée en recherche opérationnelle*, pages 57–79, 2002.

[92] E. G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

[93] A. Roli C. Blum, M. Blesa and M. Sampels, editors. *Hybrid Metaheuristics: An Emerging Approach to Optimization*. Studies in Computational Intelligence. Springer-Verlag, 2008.

[94] M. Mezmaz, N. Melab, and El-G. Talbi. A parallel exact hybrid approach for solving multi-objective problems on the computational grid. In *IPDPS*, 2006.

[95] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1):21–45, 2002.

[96] L. Di Gaspero. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools*. PhD thesis, Mathematics and Computer Science Departement, degli Studi di Udine University, Udine, Italy, 2003.

[97] K. Tindell, A. Burns, and A. Wellings. A.: Allocating hard real time tasks (an nphard problem made easy. *Journal of Real-Time Systems*, 4:145–165, 1992.

[98] S-T. Cheng and A. Agrawala. Allocation and scheduling or real-time periodic tasks with relative timing constraints. In *Technical Report CS-TR-3402*, Computer science, University of Maryland, College Park, January 1995.

[99] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[100] A. Thesen. Design and evaluation of tabu search algorithms formultiprocessor scheduling. *Journal of Heuristics*, 4(2):141–160, 1998.

[101] 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.

[102] S. Safra and O. Schwartz. On the complexity of approximating tsp with neighborhoods and related problems. *Comput. Complex.*, 14(4):281–307, 2006.

[103] S-T. Lo, R-M. Chen, Y-M. Huang, and C-L. Wu. Multiprocessor system scheduling with precedence and resource constraints using an enhanced ant colony system. *Expert Syst. Appl.*, 34(3):2071–2081, 2008.

[104] C. Reuter, M. Schwiegershausen, and P. Pirsch. Heterogeneous multiprocessor scheduling and allocation using evolutionary algorithms. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 294, Washington, DC, USA, 1997. IEEE Computer Society.

[105] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[106] I. Ahmad and M. K. Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Comput.*, 22(3):395–406, 1996.

[107] R. Nossal. An evolutionary approach to multiprocessor scheduling of dependent tasks. In *1st International Workshop on Biologically Scheduling of Dependent Tasks*, Orlando, Florida, USA, March 1998.

[108] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, dec 1989.

[109] X. Kong, J. Sun, and W. Xu. Permutation-based particle swarm algorithm for tasks scheduling in heterogeneous systems with communication delays. *International Journal of Computational Intelligence Research*, 4(1):61–70, 2008.

[110] J. Kennedy and R. C. Eberhart. *Swarm intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[111] J. Kennedy. The behavior of particles. In *EP '98: Proceedings of the 7th International Conference on Evolutionary Programming VII*, pages 581–589, London, UK, 1998. Springer-Verlag.

[112] P.Y. Yin, S.S. Yu, P.P Wang, and Y.T. Wang. A hybrid particle swarm optimization algorithm for optimal task assignment in distributed systems. *Standards and Interfaces*, 28:441–450, 1966.

[113] D. W. Boeringer and D. H. Werner. Particle swarm optimization versus genetic algorithms for phased array synthesis. *Antennas and Propagation, IEEE Transactions on*, 52(3):771–779, 2004.

[114] E. G. Talbi and T. Muntean. Hill-climbing, simulated annealing and genetic algorithms : a comparative study. In *International Conference on System Sciences HICSS: Task schedu-*

ling in parallel and distributed systems, pages 565–573, Hawaii, Jan 1993. IEEE Computer Society Press.

[115] A. W. Johnson. *Generalized hill-climbing algorithms for discrete optimization problems*. PhD thesis, 1996. Chair-S. H. Jacobson.

[116] I. H. Osman and J. P. Kelly. *Meta-heuristics : theory applications*. Kluwer Academic, 1996.

[117] S. Fidanova. Simulated annealing for grid scheduling problem. In *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 41–45, Washington, DC, USA, 2006. IEEE Computer Society.

[118] D.-T. Peng and K. G. Shin. Optimal scheduling of cooperative tasks in a distributed system using an enumerative method. *IEEE Trans. Softw. Eng.*, 19(3):253–267, 1993.

[119] D. S. Johnson, A. J. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[120] D. S. Johnson. Approximation algorithms for combinatorial problems. In *STOC '73: Proceedings of the fifth annual ACM symposium on theory of computing*, pages 38–49, New York, NY, USA, 1973. ACM.

[121] J. Korst, E. H. L. Aarts, J. K. Lenstra, and J. Wessels. Periodic multiprocessor scheduling. In *PARLE '91: Proceedings on Parallel architectures and languages Europe : volume I: parallel architectures and algorithms*, pages 166–178, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[122] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. *rtss*, 00:279, 2002.

[123] ARTIST FP5 Consortium, editor. *Embedded Systems Design - chapter Real-Tuime Scheduling*. Springer Berlin / Heidelberg ISSN, Los Alamitos, CA, USA, Monday, February 07, 2005.

[124] K. Kim, J. Luis Diaz, and J. Maria Lopez. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *IEEE Trans. Comput.*, 54(11):1460–1466, 2005.

[125] A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[126] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[127] F. Hermery. *Etude de la Repartition Dynamique d'Activités sur Architectures Décentralisées*. PhD thesis, 1994. PhD thesis.

[128] Y. E. Chen and D. L. Epley. Memory requirements in a multiprocessing environment. *J. ACM*, 19(1):57–69, 1972.

[129] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–265, New York, NY, USA, 2003. ACM.

[130] J. L. Lanet. A load balancing task allocation scheme in a hard real time system. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 640–643, London, UK, 1996. Springer-Verlag.

[131] Vilfredo Pareto. *Cours d'économie politique*. Rouge, Lausanne, 1896-1897.

[132] E. G. Talbi. Métaheuristiques pour l'optimisation combinatoire multi-objectif: Etat de l'art. Technical report, CNET, 2000.

[133] J. Knowles and D. Corne. *Metaheuristics for Multiobjective Optimisation*, volume 535 of *Lecture Notes in Economics and Mathematical Systems*, chapter Bounded Pareto Archiving: Theory and Practice, pages 39–64. Springer, 2004.

[134] K. Li, X. Yue, L. Kang, and Z. Chen. A new multi-objective evolutionary algorithm for solving high complex multi-objective problems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 745–746, New York, NY, USA, 2006. ACM.

[135] R. Kumar. Evolutionary multiobjective combinatorial optimization. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 3366–3390, New York, NY, USA, 2007. ACM.

[136] E. Saule, P-F. Dutot, and G. Mounie. Scheduling With Storage Constraints. In *Electronic proceedings of IPDPS 2008*, Miami, Florida USA, APR 2008.

[137] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 280–288, New York, NY, USA, 2007. ACM.

[138] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[139] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[140] K. Danne and M. Platzner. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 568–573, August 2005.

[141] J. Pino and E. Lee. Hierarchical static scheduling of dataflow graphs onto multiple processors. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Detroit, Michigan*, 1995.

[142] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés*. PhD thesis, Université de Paris Sud, Spécialité électronique, 30/11/2000.

[143] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1:103–111, 1991.

[144] T. Grandpierre and Y. Sorel. Un nouveau modèle générique d'architecture hététogène pour la méthodologie aaa. In *Actes des Journées Francophones sur l'Adéquation Algorithme Architecture, JFAAA'02*, Monastir, Tunisia, December 2002.

[145] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28:65–72, 1990.

[146] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. L. van Meerbergen. Predictable embedded multiprocessor system design. In *SCOPES*, pages 77–91, 2004.

[147] D. Kirby. On bezout's theorem. *The Quarterly Journal of Mathematics*, Dec 1988.

[148] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4), April 1995.

[149] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, January 2002.

[150] W. F.J. Verhaegh, P. E.R. Lippens, E. H. L. Aarts, J. L. van Meerbergen, and A. van der Werf. The complexity of multidimensional periodic scheduling. *Discrete Appl. Math.*, 89(1-3):213–242, 1998.

[151] A. Munier. The complexity of a cyclic scheduling problem with identical machines, 1990.

[152] K. M. Borgwardt. *Graph Kernels*. PhD thesis, Faculté de Mathématique, d'Informatique et de Statistique de l'université de Ludwig Maximilians, Munich, 2007.

[153] F. Tutzauer. Entropy as a measure of centrality in networks characterized by path-transfer flow. *Social networks*, 29:245–265, 2007.

[154] F. Kordon and Luqi. An introduction to rapid system prototyping. *IEEE Trans. Softw. Eng.*, 28(9):817–821, 2002.

[155] G. Sevaton. *Méthodologie de conception de composants virtuels comportementaux pour une chaîne de traitement du signal embarquée*. PhD thesis, Université de Bretagne Sud, 2002.

[156] R. David. *Architectures reconfigurables dynamiquement pour les applications mobiles*. PhD thesis, Université de Rennes 1, 2003.

[157] L. Bossuet. *Exploration de l'espace de conception des architectures reconfigurables*. PhD thesis, Université de Bretagne Sud, 2004.

[158] G. Kahn. The semantics of a simple language for parallel programming, Aug 1974.

[159] J. L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[160] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[161] M. Glesner and A. Kirschbaum. State-of-the-art in rapid prototyping. In *SBCCI '98: Proceedings of the 11th Brazilian Symposium on Integrated circuit design*, page 60, Washington, DC, USA, 1998. IEEE Computer Society.

[162] L. Kessal. *Méthodes et architectures pour le TSI en temps réel, Traité Information, Commande et Communication*, chapter 7, Conception pour DSP SHARC, pages 143–168. Hermès, 2001. D. Demigny coordonateur.

[163] M. Vasilko, L. Machácek, M. Matej, P. Stepien, and S. Holloway. A rapid prototyping methodology and platform for seamless communication systems. In *RSP '01: Proceedings of the 12th International Workshop on Rapid System Prototyping*, page 70, Washington, DC, USA, 2001. IEEE Computer Society.

[164] http://www.cofluentdesign.com/.

[165] J-P. CALVEZ. *Spécification et conception des systèmes. Une méthodologie*. Masson, Paris, France, 1990.

[166] V. Perreir. System architecting complex designs. In *Embedded Systems Europ*, pages 24–26, 2004.

[167] James E. Saultz. Rapid prototyping of application-specificsignal processors (rassp) in-progress report. *J. VLSI Signal Process. Syst.*, 15(1-2):29–47, 1997.

[168] Lockheed Martin Advanced Technology Laboratories. Gedae technical paper. Technical report, Malardalen University, 1998.

[169] http://www.gedae.com/index.php.

[170] Larry Morell and David Middleton. The software engineering learning facility. *J. Comput. Small Coll.*, 16(3):299–307, 2001.

[171] J. Tsay. A code generation framework for ptolemy ii. Technical Report UCB/ERL M00/25, EECS Department, University of California, Berkeley, 2000.

[172] J. L. and E. A. Lee. A component-based approach to modeling and simulating mixed-signal and hybrid systems. *ACM Trans. Model. Comput. Simul.*, 12(4):343–368, 2002.

[173] M. Mbobi and F. Boulanger. Le paradigme acteur dans la modelisation des systemes embarques. In *CCECE*, pages 418–421, 2006.

[174] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[175] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, Berkeley, CA, USA, 1995.

[176] http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm RTI-MP.

[177] http://www.mathworks.fr/.

[178] Tom Henzinger, Ben Horowitz, and Christoph Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, January 2003.

[179] H. Salzwedel. Mission level design of avionics. In *The 23rd Digital Avionics Systems Conference, AIAA-IEEE DASC 04*, pages 24–28, Salt Lake City, Utah, USA, 2004.

[180] J. W.S. Liu, J. Redondo, Z. Deng, T. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. Shih. Perts: A prototyping environment for real-time systems. Technical report, Champaign, IL, USA, 1993.

[181] Tri-Pacific Software Inc 2005. Rapid RMA Data Sheet http://www.tripac.com.

[182] Time Sys Corporation 2005. TimeWiz Data Sheet http://www.timesys.com/.

[183] Y. Meylan, A. Bajpai, and R. Bettati. Protex: a toolkit for the analysis of distributed real-time systems. *Real-Time Computing Systems and Applications, International Workshop on*, 0:183, 2000.

[184] R. Bettati. End-to-end scheduling to meet deadlines in distributed systems. Technical report, Champaign, IL, USA, 1994.

[185] S. Tregar. A graphical tool for rate monotonic analysis, http://www.tregar.com/grma/.

[186] J. Migge. Tkrts: A tool for computing response time bounds, http://www.migge.net/jorn/rts/.

[187] K Bryan, T. Ren, J. Zhang, L. DiPippo, and V. Fay-Wolfe. The design of the openstars adaptive analyzer for real-time distributed systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 2*, page 129.2, Washington, DC, USA, 2005. IEEE Computer Society.

[188] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[189] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design*, Rome, Italy, May 1999.

[190] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. PhD thesis, Université de Paris Nord, Spécialité informatique, 5/07/1999.

[191] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6), 1997.

[192] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Springer, New York, 2000.

[193] V. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1), 1986.

[194] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de Recherche 3476, INRIA, August 1998.

[195] http://www-rocq.inria.fr/syndex/.