Real-Time Systems, RTS'2001, March 6-8 2001, Paris

Ordonnancement de tâches temps réel en environnement multiprocesseur à l'aide de réseaux de Petri

Emmanuel Grolleau, Annie Choquet-Geniet
Laboratoire d'Informatique Scientifique et Industrielle
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2
1, avenue Clément Ader
BP 40109
86961 FUTUROSCOPE CHASSENEUIL Cedex
grolleau,ageniet@ensma.fr

Résumé : Nous présentons une technique d'ordonnancement hors-ligne optimal de systèmes de tâches temps réel en environnement multiprocesseur. Elle se base sur la modélisation d'un système de tâches à l'aide d'un réseau de Petri avec la règle de tir maximal. Le centre du langage terminal du réseau de Petri construit est l'ensemble des séquences d'ordonnancement valides du système de tâches modélisé. De plus, nous soulevons un problème de cyclicité des ordonnancements multiprocesseur lorsque certaines tâches sont différées.
Mots clés : Ordonnancement multiprocesseur, réseaux de Petri, cyclicité

1. Introduction

Nous nous intéressons à des applications temps réel de grande taille, contrôlant des procédés complexes. Lorsqu'elles deviennent imposantes, ces applications ne peuvent plus être validées sur des architectures monoprocesseur. La mise en parallèle des différentes tâches constituant l'application devient donc nécessaire si l'on veut augmenter les performances et garantir le respect des contraintes temporelles induites par le procédé contrôlé. Le concepteur de l'application doit donc se tourner vers des architectures multiprocesseur. Ces architectures peuvent être parallèles : il existe une mémoire commune par laquelle les messages peuvent transiter, ou réparties : pas d'existence de mémoire commune et les messages transitent à travers un réseau de communication.

Nous considérons ici le modèle PRAM [11] : chaque processeur est identique et a une mémoire privée, mais une mémoire commune est accessible en temps constant par tous les processeurs. Sous cette hypothèse, le coût de communication inter processeurs est généralement négligé, contrairement au cas réparti.

Afin de tirer parti du parallélisme inhérent aux architectures multiprocesseurs, il convient de résoudre conjointement les problèmes de placement et d'ordonnancement. Concernant le placement, différentes hypothèses portant sur la migration de tâches peuvent être adoptées :

- migration totale : à chaque instant, une tâche peut migrer vers un autre processeur,
- migration faible : à chaque réveil, une tâche choisit un processeur qu'elle conserve pour l'instance courante,
- absence de migration : une tâche est affectée une fois pour toute sur un processeur.

Malgré la simplicité relative du cas multiprocesseur par rapport au cas réparti, la plupart des problèmes d'ordonnancement multiprocesseur sont NP-difficiles, et seuls quelques cas particuliers peuvent être résolus de façon optimale en un temps polynomial [1, 18].

En effet, même pour des tâches indépendantes non-périodiques ordonnancées sur deux processeurs, sous l'hypothèse de migration totale, le problème de l'existence d'un ordonnancement optimal sans que soient connues a priori les dates de réveil des tâches n'est pas décidable [16, 6]. Il en résulte qu'aucun algorithme en-ligne ne peut être optimal sous cette hypothèse.

Par ailleurs, l'ordonnancement de tâches, même indépendantes, sur au moins deux processeurs se heurte au problème de non régularité [15] : ceci correspond au fait que le pire cas en terme d'ordonnancement ne correspond pas nécessairement au pire cas en terme de durée d'exécution des tâches. Ainsi, une diminution des durées effectives des tâches, toujours envisageable puisque l'on vérifie avec des pires durées, peut remettre en cause l'ordonnançabilité de l'application établie pour les pires durées d'exécution.

Ces difficultés rendent délicate la mise en œuvre d'algorithmes en-ligne pour ordonnancer les systèmes de tâches en environnement multiprocesseur, et plaident en faveur de l'utilisation d'approches hors-ligne, de complexité exponentielle, mais pouvant fournir des solutions optimales.

De nombreuses études d'ordonnancement hors-ligne de systèmes temps réel ont donc été menées. La plupart adoptent une hypothèse de non-préemptibilité des tâches,

couplée à l'hypothèse de migration faible. [10, 19] proposent sous ces hypothèses des méthodes d'ordonnancement optimal avec contraintes de précédence et, pour [19] des contraintes d'exclusion. Dans ce même contexte, [13] propose une solution basée sur des algorithmes non-déterministes. Cependant, [12] montre que la puissance d'ordonnançabilité d'un algorithme hors-ligne optimal avec migration totale est strictement plus grande que celle d'un algorithme sans migration ou avec migration faible. Cela signifie qu'il existe des systèmes de tâches non ordonnançables avec des techniques optimales sous l'hypothèse de migration faible, alors qu'elles le sont sous l'hypothèse de migration totale.

Nous proposons ici une technique d'ordonnancement optimale dans le cas où la migration est totale, et les tâches préemptibles. La méthode se base sur une modélisation par réseau de Petri d'un système de tâches temps réel périodiques pouvant communiquer et partager des ressources critiques. Une approche similaire avait été présentée dans [9] pour le cas monoprocesseur. L'objectif de cet article est de montrer comment utiliser cette méthode dans le cas multiprocesseur, et de mettre en évidence les problèmes théoriques que cela soulève, notamment en matière de cyclicité.

Dans la seconde partie, nous présentons nos hypothèses de travail d'un point de vue logiciel et matériel. La troisième partie introduit le modèle réseau de Petri utilisé pour modéliser les systèmes de tâches, et la partie suivante explique comment un tel modèle est utilisé pour l'ordonnancement. Cette partie introduit un problème inhérent à la cyclicité des ordonnancements de tâches différées en environnement multiprocesseur.

2. Hypothèses

2.1. Modèle de tâches

Nous supposons que chaque tâche τ_i est caractérisée temporellement par quatre paramètres temporels [14] :

- r_i sa date d'activation, ou première date de réveil ;
- C_i sa charge, c'est à dire la durée que le processeur doit lui consacrer pour exécuter chacune de ses occurrences ;
- D_i son délai critique, représente le temps qui lui est accordé après chacune de ses activations pour être exécutée. Nous supposons que $D_i \leq P_i$. La date à laquelle une instance doit être terminée se nomme l'échéance ;
- P_i sa période d'activation ;

Le modèle utilisé est déterministe, et toutes les dates et les durées prennent des valeurs entières : en effet, le temps est discrétisé en quanta de temps correspondant à l'unité de préemption de l'architecture cible. Chaque tâche est (ré-)activée périodiquement aux dates r_i+kP_i pour $k \in \mathbb{N}$, chacune des occurrences doit respecter son échéance donnée par $r_i+kP_i+D_i$. La figure 1 représente graphiquement les paramètres temporels d'une tâche τ_i. Nous dénotons une tâche $\tau_i<r_i,C_i,D_i,P_i>$ et un système de tâches $S=\{\tau_i<r_i,C_i,D_i,P_i>\}_{i=1..n}$. On note $P=PPCM_{i=1..n}(P_i)$ la méta période d'un système et $r=max_{i=1..n}\{r_i\}$ sa date de réveil la plus tardive.

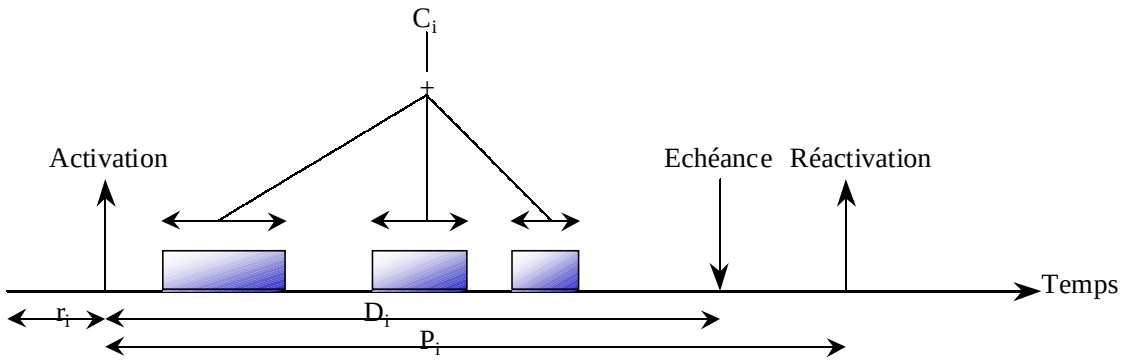

figure 1 : modèle temporel d'une tâche périodique

2.2. Architecture matérielle

L'architecture matérielle est composée de *p* processeurs identiques, ayant chacun leurs registres et leur mémoire privée, mais partageant une mémoire commune. Cela signifie que la durée des tâches est identique, quel que soit le processeur utilisé.

Les communications entre les tâches sont réalisées à l'aide de zones boîtes aux lettres situées dans la mémoire partagée. Pour tout couple de tâches communicantes, une boîte aux lettres $bal_{i,j}$ est mise en place. L'accès à une boîte aux lettres est protégée de façon standard pour une zone de mémoire partagée (sémaphore, moniteur,…), ce qui évite les accès simultanés de la tâche émettrice et de la tâche réceptrice. Cependant, ce mécanisme a un coût en temps, qui est pris en compte dans la durée des primitives d'envoi et de réception de messages.

Les ressources critiques sont prises en compte au niveau global (une ressource peut être accédée à partir de n'importe quel processeur). De même que pour les communications, l'accès aux ressources critiques a un coût qui est pris en compte dans les primitives de prise et de libération de ressource.

Enfin, tous les processeurs sont synchronisés au moins à chaque quantum de temps.

2.3. Architecture logicielle

La table 1 présente les hypothèses de placement et de préemptibilité pouvant être étudiées :

	Migration totale (M_2)	Migration faible (M_1)	Absence de migration ($\overline{M}$)
Préemptif (P)	PM_2	PM_1	$P\overline{M}$
Non-préemptif ($\overline{P}$)	$\overline{P}M_2$ (impossible)	$\overline{P}M_1$	$\overline{P}\ \overline{M}$

table 1: Hypothèses conjointes de placement et de préemptibilité

Sous les hypothèses $\overline{M}$, un processeur est alloué une fois pour toutes à chaque tâche périodique. Sous les hypothèses M_1, un processeur différent peut être alloué à chacune des instances des tâches périodiques. Enfin, sous l'hypothèse PM_2, une tâche peut migrer à n'importe quel quantum de temps d'un processeur à un autre. Sous cette hypothèse, les surcoûts dus aux durées d'écriture et de lecture des données locales d'un

processeur vers la mémoire commune doivent être prises en compte dans la durée des tâches pour chaque quantum de temps. Afin que cette hypothèse soit réaliste, il convient de choisir un quantum de temps relativement important pour l'hypothèse PM_2.

Nous définissons l'ordre partiel $\leq_{ordo}$: hypothèse placement et préemptibilité $\times$ hypothèse placement et préemptibilité $\rightarrow$ booléen, avec $H_1 \leq_{ordo} H_2$ si et seulement si tout système de tâches ordonnançable sous l'hypothèse H_1 l'est aussi sous l'hypothèse H_2. Et nous notons $<_{ordo}$ l'ordre strict associé.

Alors, $\overline{P}\ \overline{M} \leq_{ordo} \overline{P}\, M_1 \leq_{ordo} PM_1 <_{ordo} PM_2$ et $P\,\overline{M} \leq_{ordo} PM_1$. Dans ces relations, les $\leq_{ordo}$ découlent du fait que l'opérande de gauche est un cas particulier de l'opérande de droite. La relation d'ordre stricte $PM_1 <_{ordo} PM_2$ vient de [12].

A notre connaissance, toutes les approches hors-ligne étudiées dans la littérature se basent sur des tâches non périodiques et non préemptibles, ce qui place ces approches sous l'hypothèse $\overline{P}\, M_1$.

Notre approche se base sur l'hypothèse d'ordonnancement la plus large : PM_2. Alors que sous les hypothèses $\overline{M}$, seul l'ordonnancement des tâches est à étudier pour un placement donné, sous les hypothèses de migration M_1 et M_2, le placement et l'ordonnancement doivent être étudiés conjointement.

3. Modélisation de systèmes de tâches par réseaux de Petri

3.1. Présentation de la méthode

La méthode présentée s'inspire de la méthode d'ordonnancement par réseaux de Petri en environnement monoprocesseur dont différents aspects ont été présentés dans [8, 9].

Elle se base sur une modélisation fine du système de tâches à ordonnancer par un réseau de Petri autonome, fonctionnant avec la règle de tir maximal. Cette règle de tir donne au réseau de Petri la puissance d'expression d'une machine de Turing, et permet entre autres de modéliser le temps.

Le réseau de Petri construit est tel qu'un sous ensemble de son langage, nommé centre de son langage terminal, est exactement l'ensemble des séquences d'ordonnancement valides pour le système de tâches modélisé.

Le langage est obtenu par construction du graphe des marquages du réseau de Petri. A partir de ce graphe des marquages, il est possible d'extraire en un temps linéaire de la taille du graphe des séquences d'ordonnancement optimales en fonction de différents critères qualitatifs (temps de réponse minimal, répartition des temps creux,…).

Cette méthode est donc une méthode d'énumération exhaustive (à l'exception de séquences peu intéressantes éliminées par des méthodes heuristiques) des séquences d'ordonnancement valides.

3.2. Présentation des réseaux de Petri avec la règle de tir maximal

3.2.1 Définition

Les réseaux de Petri (RdP) temporisés [17, 4] ont déjà été utilisés dans le cadre de l'ordonnancement cyclique [2]. [5] a montré l'équivalence entre les RdP temporisés à vitesse maximale[1] et les réseaux de Petri autonomes avec la règle de tir maximale[2].

Cependant, l'implémentation des RdP avec la règle de tir maximal est simplifiée par le fait qu'il n'est pas nécessaire de prendre en compte le vecteur des durées résiduelles de tir en plus du marquage courant pour connaître l'état du RdP. Ces RdP ont la puissance d'expression d'une machine de Turing, ce qui permet de modéliser toute interaction possible entre les tâches. En contrepartie, l'analyse de tels RdP est délicate, mais nous nous intéressons ici à leur langage.

Définition

Un RdP autonome avec la règle de tir maximal est défini par :

- Q un ensemble fini de places,
- T un ensemble fini de transitions, avec $Q \cap T = \varnothing$,
- $F \subseteq (Q \times T) \cup (T \times Q)$ le flot du réseau,
- $W : F \rightarrow \mathbb{N}^+$ la fonction de pondération du flot,
- $M : Q \rightarrow \mathbb{N}$ est le marquage du réseau, on note M_0 le marquage initial.

On dit qu'une transition t est valide pour un marquage M si, comme pour les RdP autonomes, $\forall p \in Q$ tel que $(p,t) \in F$, $W(p,t) \leq M(p)$.

La règle de tir maximal est définit le schéma opérationnel du réseau : à chaque tir de transitions, on franchit simultanément un ensemble I' maximal de transitions. Si $I \subseteq T$ est l'ensemble des transitions valides à partir d'un marquage M, un sous-ensemble $I' \subseteq I$ est maximal si toute transition $t \in I \setminus I'$ est en conflit avec au moins l'une des transitions de I'.

De plus, le fait que les ensembles manipulés ne sont pas des multi-ensembles exclue toute possibilité de réentrance d'une transition. C'est à dire qu'il est impossible de franchir simultanément plusieurs fois la même transition.

❒

3.2.2 Prise en compte du temps

D'après la règle de tir maximal, lorsqu'un tir de transitions a lieu, toute transition valide sans conflit est franchie, quelles que soient les autres transitions franchies. Il en résulte qu'une transition source (i.e. une transition sans place en entrée, c'est à dire toujours valide) est franchie quelles que soient les transitions franchies simultanément.

---

[1] c'est à dire lorsque l'on tire de façon synchrone des ensembles maxima, au sens de l'inclusion, de transitions franchissables avec respect des conflits

[2] cette règle est la même que la règle de vitesse maximale, mais dans le cas de réseaux de Petri autonomes

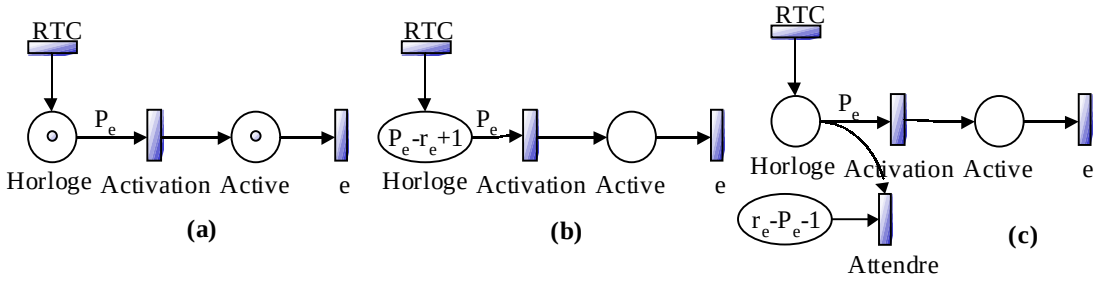

figure 2 : Représentation d'actions périodiques

La figure 2a représente la modélisation d'une action *e* de durée unitaire, périodique de période P_e>1. La figure 2b représente la même action périodique dont la première exécution doit être différée de $r_e \leq P_e+1$ unités de temps. La figure 2c représente la même action avec $r_e > P_e+1$.

Explications de la figure 2a : La transition *RTC* est une transition source, elle est donc toujours valide, et, par respect de la règle de tir maximal, étant donné qu'elle n'est en conflit avec aucune autre transition, elle est franchie à chaque fois qu'un ensemble de transitions est franchi. Il en résulte qu'elle produit à chaque tir d'un ensemble de transitions un jeton dans la place *Horloge*, qui fonctionne comme un horloge locale de l'action périodique. La transition *RTC* marque donc le temps, et chaque tir d'un ensemble maximal de transitions franchissables dure une unité de temps sur cette échelle de temps. Dès lors que *Horloge* contient P_e jetons, c'est à dire aux instants kP_e-1 pour $k \in \mathbb{N}^+$, la transition *Activation* est tirée, produisant un jeton dans la place *Active*, dont la sémantique est de traduire le fait que l'action *e* peut être effectuée. Si la transition *e* n'est en conflit avec aucune autre transition (ce qui le cas ici), elle est tirée aux instants kP_e pour $k \in \mathbb{N}$. L'utilité de la transition source *RTC* est de « rythmer » le tir des transitions : en effet, à chaque tir d'un ensemble de transitions, *RTC* est tirée, tout se passe donc comme si RTC était l'horloge du modèle. Dans la suite le tir d'un ensemble de transitions est donc assimilé au passage d'une unité de temps *RTC*.

Afin de modéliser des actions périodiques dont la première exécution est différée dans le temps de r_e unités de temps, remarquons que lorsque $r_e \leq P_e+1$, une modification du marquage initial suffit (voir figure 2b). Dans le cas où $r_e > P_e+1$, le modèle est augmenté d'une place et d'une transition permettant d'obtenir le comportement adéquat (voir figure 2c).

3.3. Modélisation d'une tâche à l'aide d'un RdP

Une tâche périodique est un ensemble d'actions pouvant être exécutées périodiquement. Chacune des actions composant le corps d'une tâche nécessite l'utilisation d'une ressource : un processeur. Sous l'hypothèse étudiée, les processeurs sont au nombre de *p*, identiques, et une tâche peut migrer indifféremment d'un processeur à un autre. Afin d'éviter les symétries de placement lors d'une énumération des ordonnancements/placements possibles des tâches (tâche τ_i sur processeur p_q, τ_j sur p_r par rapport à τ_i sur p_r et τ_j sur p_q), le modèle ne différencie pas les processeurs. Il en résulte que les séquences obtenues à partir du modèle possèdent des informations sur les tâches exécutées indépendamment de leur placement sur tel ou tel processeur.

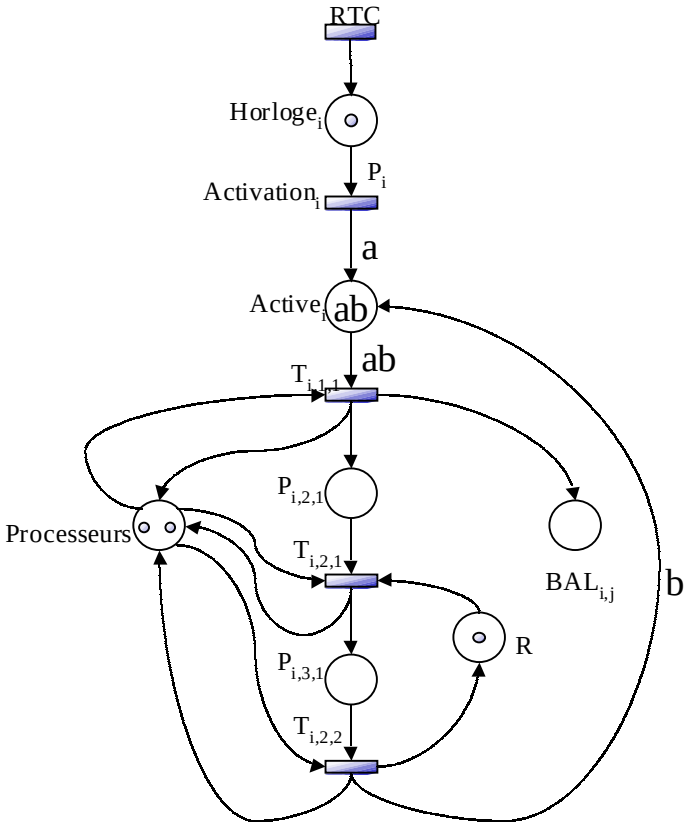

figure 3 : modélisation d'une tâche périodique τ_i

La figure 3 représente le réseau de Petri associé à une tâche τ_i de date de réveil r_i=0, de période P_i, dont le pseudo-code est le suivant :

```
Traitement de durée 1
Envoyer_message(BALi,j)
Prendre_Ressource(R)
Traitement de durée 2
Rendre_Ressource(R)
```

Les transitions *RTC* et *Activation$_i$* couplées à la place *Horloge$_i$* fonctionnent sur le même principe que sur la figure 3 : la transition *Activation$_i$* est tirée toutes les P_i unités de temps. La différence est qu'afin de simplifier la représentation graphique du modèle, la transition activation produit un jeton de couleur *a* dans *Active$_i$*. La sémantique associée à la couleur *a* est l'activation.

La modélisation du corps de τ_i est réalisée de façon classique : le premier traitement de durée 1 est modélisé par la transition $T_{i,1,1}$ utilisant une ressource processeur (rappelons que sur notre échelle temporelle le tir d'un ensemble de transitions prend une unité de temps). Cette transition produit un jeton dans une place boîte aux lettres $BAL_{i,j}$.

Enfin, la ressource *R* est utilisée pendant 2 unités de temps modélisées par $T_{i,2,1}$ et $T_{i,2,2}$.

La dernière transition composant la tâche produit un jeton de couleur *b* dans la place $Active_i$, sa sémantique la suivante : il n'existe pas d'exécution en cours de τ_i. Afin de commencer l'exécution d'une tâche, il faut qu'elle soit active et pas déjà en cours d'exécution. Cela est modélisé par le fait que la première transition $T_{i,1,1}$ modélisant le corps de τ_i consomme un jeton de couleur *a* et un jeton de couleur *b*.

Enfin, les contraintes temporelles à prendre en compte sont données par le délai critique D_i des tâches. Lorsqu'une tâche τ_i reçoit un jeton *a*, elle est réactivée. A ce moment-là, elle doit avoir fini son exécution lors de sa dernière activation, ce qui signifie qu'elle possède (ou reçoit simultanément) un jeton *b*. Si la tâche n'est pas à échéance sur requête (i.e. $D_i \neq P_i$), lorsque l'horloge locale de la tâche dépasse son délai critique D_i, la tâche doit avoir terminé son exécution, ce qui s'écrit $M(Horloge_i) > D_i \Rightarrow M(Active_i) = \{b\}$ où *M* est la fonction de marquage. Si la tâche est à échéance sur requête, $P_i - 1$ unités de temps après (ré)activation de τ_i, $M(Horloge_i) = P_i$, et $M(Active_i) \leq b$. Une unité de temps plus tard, $M(Horloge_i) = 1$ et $Active_i$ doit contenir un jeton *a* et un jeton *b*. Puisque $M(Horloge_i) = 1$ uniquement lorsque la tâche τ_i vient d'être (ré)activée, il est nécessaire, afin que la tâche respecte son échéance, que $M(Horloge_i) = 1 \Rightarrow M(Active_i) = a + b$. Les échéances des tâches, à échéance sur requête ou non, nécessitent donc que

$$(M(Horloge_i) > D_i \Rightarrow M(Active_i) = \{b\}) \qquad [1_i]$$

et $$(M(Horloge_i) = 1 \Rightarrow M(Active_i) = \{a,b\}) \qquad [2_i]$$

pour un marquage donné M.

Ces deux contraintes garantissent les contraintes temps réel des tâches. Un marquage du réseau de Petri n'est valide que si il respecte les contraintes $[1_i]$ et $[2_i]$ pour toute tâche τ_i du système. Cette notion de validité des marquages se ramène à des contraintes terminales sur le réseau de Petri. L'ensemble des comportements valides (langage terminal) du réseau de Petri est donné par l'ensemble de ses comportements pour lesquels seuls des marquages terminaux (i.e. respectant les contraintes) sont atteints.

3.4. Modélisation d'un système de tâches à l'aide d'un RdP

Il reste maintenant à mettre les tâches d'un système de tâches en parallèle, toutes concurrentes pour l'obtention d'un processeur. Etant donné que les processeurs sont synchronisés temporellement, c'est le même top d'horloge *RTC* qui sert à décompter le temps dans chacune des horloges locales des tâches.

A chaque tir d'un ensemble de transitions, les transitions représentant les tâches sont concurrentes pour l'obtention d'un des processeurs, non différenciés.

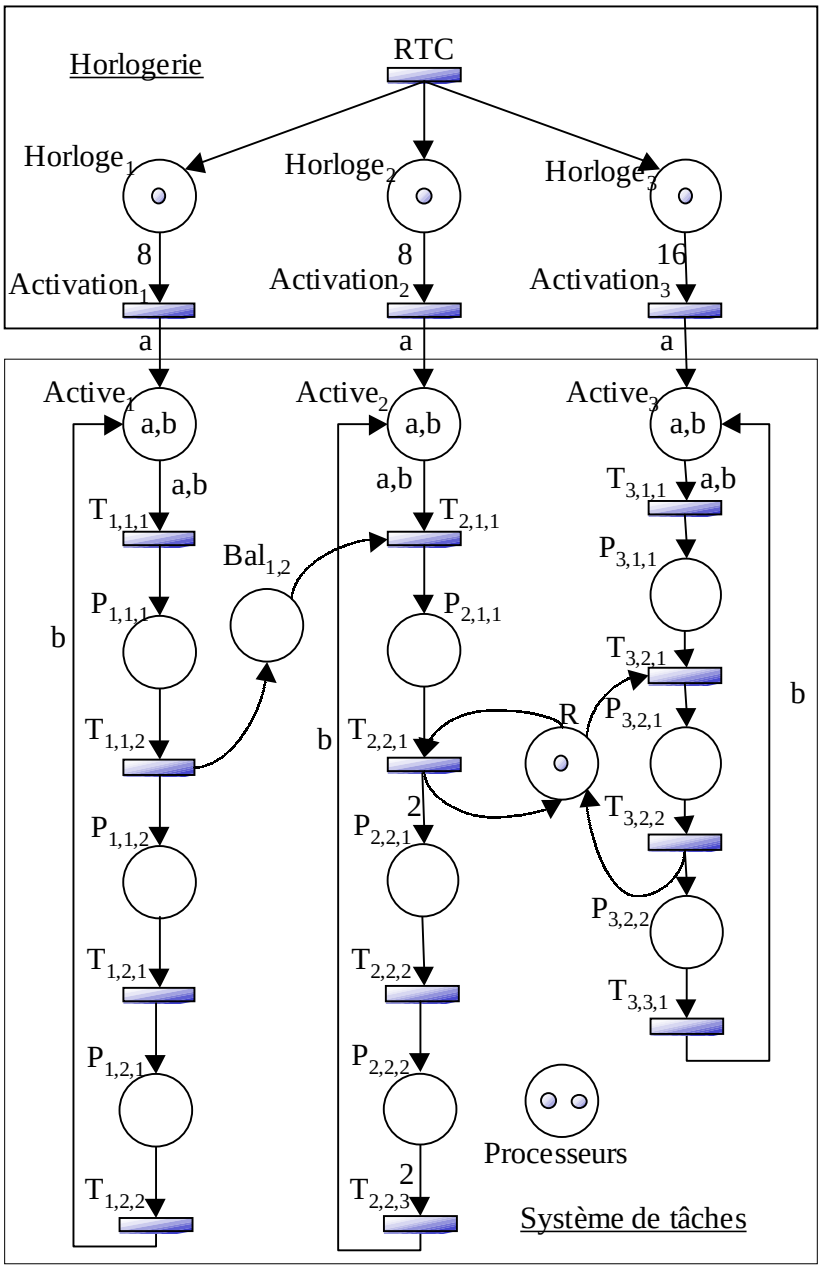

figure 4 : modélisation du système de tâches S_1 ordonnancé sur deux processeurs (les flèches entre la place processeur et les transitions $T_{i,j,k}$ sont omises sur la représentation)

La figure 4 représente le réseau de Petri utilisé pour modéliser les comportements possibles d'un système $S_1=\{\tau_1<0,4,6,8>,\tau_2<0,4,8,8>,\tau_3<0,4,16,16>\}$ où le pseudo-code des tâches est le suivant :

Pseudo-code de τ_1	Pseudo-code de τ_2	Pseudo-code de τ_3

`Traitement de durée 2`
`Envoyer_message(Bal1,2)`
`Traitement de durée 2`
	`Attendre_message(Bal1,2)`
	`Traitement de durée 1`
	`Prendre_Ressource(R)`
		`Traitement de durée 1`
		`Prendre_Ressource(R)`
		`Traitement de durée 2`

	Traitement de durée 1 Rendre_Ressource(R) Traitement de durée 2	Rendre_Ressource(R) Traitement de durée 1

Les délais critiques de tâches sont prises en compte par les contraintes terminales suivantes :

$M(Horloge_1)>6 \Rightarrow M(Active_1) \geq \{b\} \wedge$

$M(Horloge_2)=1 \Rightarrow M(Active_2)=\{a,b\} \wedge$

$M(Horloge_3)=1 \Rightarrow M(Active_3)=\{a,b\}$

Par construction, les comportements infinis du RdP, où tous les marquages atteints respectent les contraintes terminales, correspondent à un sous ensemble des comportements temporels valides du système de tâches modélisé : ce sous ensemble est exactement l'ensemble des ordonnancements conservatifs[3] du système modélisé.

Cependant, dès lors qu'il y a des ressources critiques en jeu, les ordonnancements conservatifs ne sont pas dominants. Il en résulte qu'il faut permettre au modèle d'introduire arbitrairement des temps creux.

3.5. Le problème des temps creux

Dans le cas monoprocesseur, le système de tâches se voit adjoindre une tâche dite oisive $\tau_0<r_0,P(1-U),P,P>$ où :

– $P=PPCM_{i=1..n}\{P_i\}$ est la méta-période du système

– $U=\sum_{i=1}^{n} C_i/P_i$ est la charge processeur du système

Le rôle de la tâche oisive est de modéliser les temps creux laissés par le système de tâches. Cette tâche, considérée comme les autres tâches, permet à la technique d'énumération basée sur les RdP de générer les séquences d'ordonnancement non conservatives.

De même, nous introduisons dans le cas multiprocesseur une tâche oisive $\tau_0<r_0,P(p-U),P,P>$. En effet, sur p processeurs, le nombre de temps creux laissés lors d'une méta-période P est $P(p-U)$.

La particularité de la tâche oisive multiprocesseur par rapport aux autres tâches est qu'elle est parallélisable. Alors qu'il est interdit d'exécuter simultanément une tâche τ_i simultanément sur deux processeurs, il est tout à fait permis de laisser plusieurs processeurs oisifs simultanément.

La première modélisation (voir figure 5a) de τ_0 venant à l'esprit consiste à changer la structure sérielle (i.e. les transitions étant mises en séries, elles ne peuvent être franchies que les unes après les autres, deux transitions ne peuvent donc pas être franchies simultanément) de ses transitions en une structure parallèle. A chaque activation de τ_0, $P(p-U)$ jetons sont produits dans la place $Active_0$. La tâche est composée de p transitions parallèles $T_{0,1,i}$ pour $i=1..p$, représentant le fait que le processeur i est oisif. Cependant, nous ne souhaitons pas à ce niveau différencier les processeurs, la différenciation des processeurs pour τ_0 entraînant de nombreuses symétries : à un instant donné, il est intéressant de savoir combien de processeurs sont

[3] Une séquence d'ordonnancement conservative est telle qu'un temps creux n'a lieu à un instant que si aucune tâche ne peut être exécutée à cet instant

oisifs et non pas que les processeur p_i et p_j sont oisifs. En effet, à chaque instant où $k \leq p$ processeurs sont oisifs, il existe $\binom{p}{k}$ combinaisons possibles de processeurs oisifs.

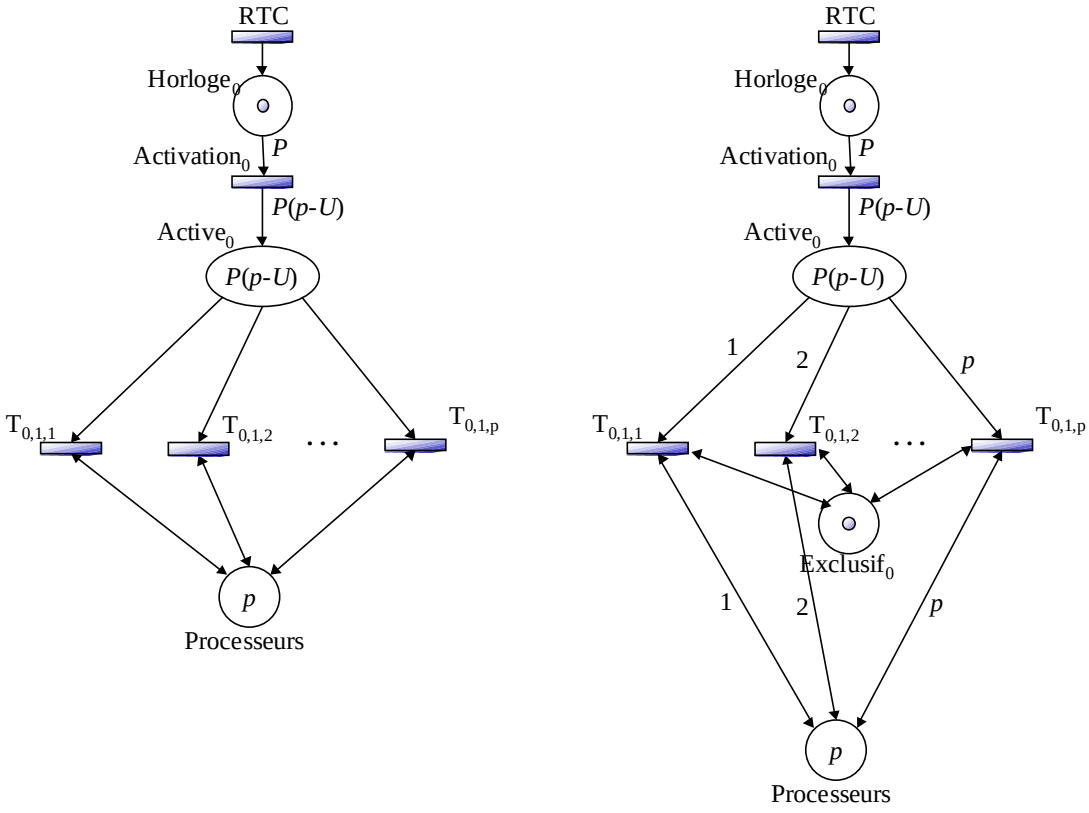

figure 5 : (a) modélisation « naïve » de la tâche oisive (b) modélisation efficace de la tâche oisive

La figure 5b présente donc une solution permettant d'éviter toute symétrie liée à la différenciation des processeurs : si à un instant k processeurs sont oisifs, alors la transition $T_{0,1,k}$ est tirée. Cette transition est la seule transition de τ_0 pouvant être tirée étant donné que toutes les transitions $T_{0,1,j}$ sont mises en exclusion par la place ressource $Exclusif_0$.

Dans les deux cas, le délai critique de la tâche oisive est pris en compte par la contrainte terminale $M(Active_0) \leq P(p\text{-}U)$.

4. Utilisation du modèle pour l'ordonnancement

L'adjonction de la tâche oisive au système permet au RdP d'englober les comportements non conservatifs.

Dans cette partie, nous montrons comment obtenir les séquences d'ordonnancement valides du système modélisé à l'aide du modèle RdP.

Une séquences d'ordonnancement valide sur p processeurs est donnée par :

- Une séquence infinie indépendante du placement de multi-ensembles, de cardinal *p*, composés de noms de tâches, par exemple sur deux processeurs, une séquence indépendante du placement pourrait être $(\{\tau_1,\tau_2\},\{\tau_1,\tau_2\},\{\tau_3,\tau_0\},\{\tau_0,\tau_0\})^*$, ce qui signifie qu'entre les dates 0 et 2, les tâches τ_1 et τ_2 sont exécutées, puis entre les dates 2 et 3, τ_3 est exécutée et un processeur est oisif, enfin entre les instants 3 et 4, les deux processeurs sont oisifs. L'étoile signifie que cette séquence peut être infiniment répétée.
- Un placement des tâches sur les processeurs. En effet, dans la séquence indépendante du placement ci-dessus, nulle mention n'est faite du choix des processeurs.

4.1. Analyse du modèle et placement a posteriori

Le RdP ne différenciant pas les processeurs, sa simulation permet d'obtenir l'ensemble exhaustif des séquences indépendantes du placement. Le placement est fait a posteriori sur une séquence choisie.

4.1.1 Ordonnancement

Chaque transition modélisant une tâche $\tau_i \neq \tau_0$ (hormis sa partie horlogerie) est étiquetée par la lettre τ_i. Les transitions du système d'horlogerie sont étiquetées par le mot vide. Enfin, les transitions modélisant le tâche oisive $T_{0,1,k}$ sont étiquetées par le multi-ensemble de lettres τ_0^k (i.e. *k* fois la lettre τ_0).

Le centre du langage terminal du RdP ainsi étiqueté est alors l'ensemble des séquences valides indépendantes du placement du système de tâches modélisé. En d'autres termes, les mots infinis pour lesquels tous les marquages atteints respectent les contraintes terminales sont exactement l'ensemble des séquences valides indépendantes du placement.

Par exemple, une séquence valide du système de tâches S_1 ordonnancé sur deux processeurs dont le modèle est représenté sur la figure 6 est :
$(\{\tau_1,\tau_3\},\{\tau_1,\tau_3\},\{\tau_1,\tau_2\},\{\tau_1,\tau_3\},\{\tau_2,\tau_3\},\{\tau_2,\tau_0\},\{\tau_2,\tau_0\},\{\tau_0,\tau_0\},\{\tau_1,\tau_0\},\{\tau_1,\tau_0\},\{\tau_1,\tau_2\},$
$\{\tau_1,\tau_2\},\{\tau_2,\tau_0\},\{\tau_2,\tau_0\},\{\tau_0,\tau_0\},\{\tau_0,\tau_0\})^*$.

Afin de connaître complètement une séquence d'ordonnancement associée à cette séquence non placée, il faut appliquer un algorithme de placement.

4.1.2 Placement

Connaissant la séquence d'ordonnancement, il reste à placer les tâches sur les processeurs afin d'éviter les migrations et préemptions inutiles. On pourrait aussi tenter de minimiser les messages inter-processeurs, mais dans la mesure où par hypothèse, les communications inter-processeurs ne sont pas plus coûteuses que les communications n'impliquant qu'un seul processeur, cela paraît superflu.

Nous nous intéressons donc à la minimisation des changements de contexte. Il s'agit d'assurer qu'une tâche ne change pas de processeur lorsqu'elle travaille en continu : en effet, alors qu'une tâche préemptée nécessite de toutes façons un changement de contexte, une tâche que l'on peut ne pas préempter ne devra pas nécessiter de changement de contexte, donc tant qu'une tâche n'est pas préemptée, elle doit rester sur le même processeur.

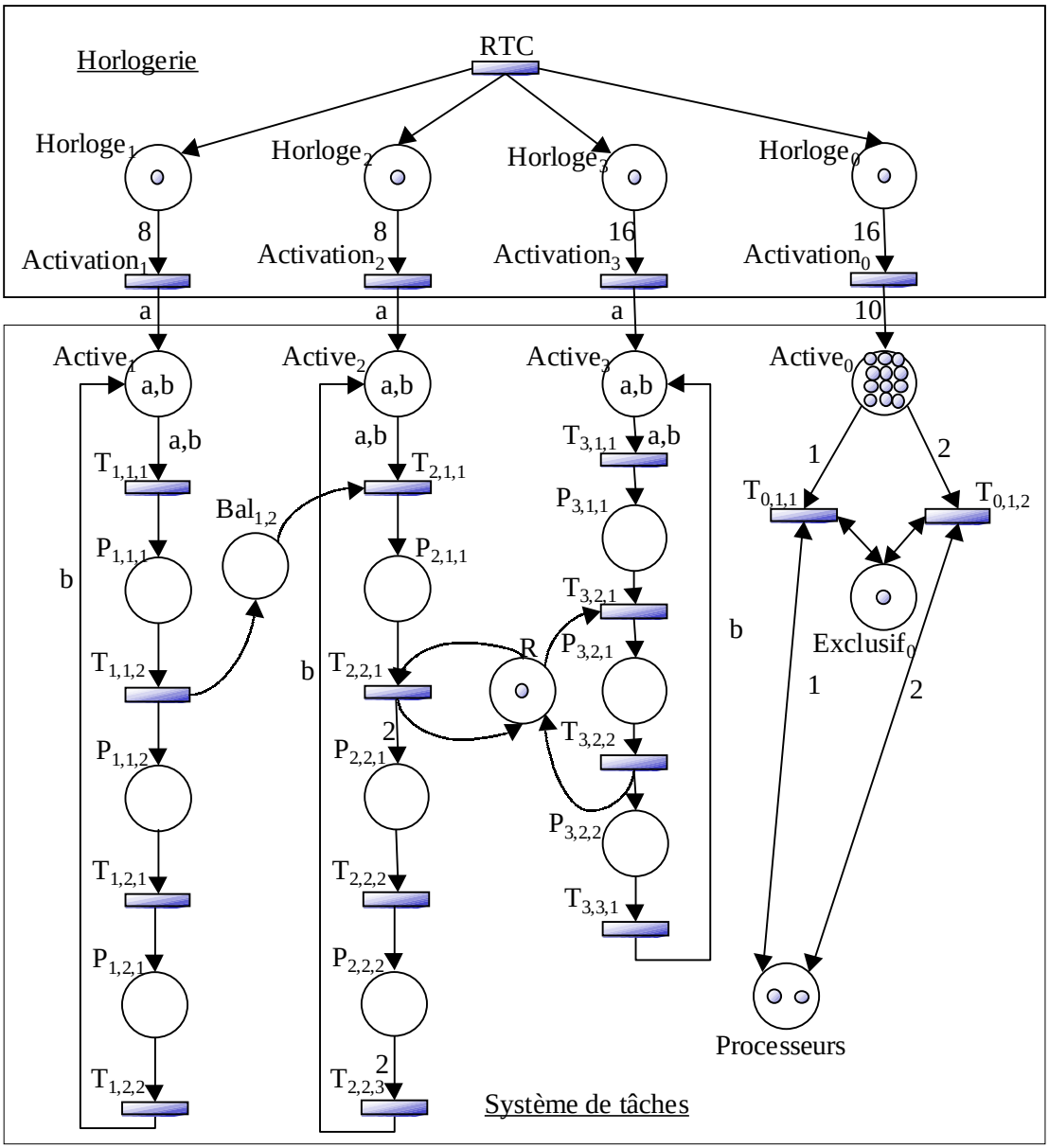

figure 6 : modélisation du système de tâches S_1 sur deux processeurs permettant l'obtention des séquences non conservatives

L'algorithme de placement permettant d'éviter les changements de contexte inutiles est donc le suivant :

```
Fonction Affecter (S : Séquence) retourne Séquence
-- Entrée : S est un tableau de P p-uplets, c'est à dire une séquence
-- fournie par le modèle
-- Retour : renvoie la même séquence S' avec placement, i.e.
-- la tâche de S'(t,i) est affectée au processeur i
S' : Séquence
Début
    Pour i de 1 à p faire
    -- Les processeurs sont alloués de façon arbitraire
```

```
    -- pour commencer
       S'(1,i)←S(1,i)
    Fait
    Pour t de 2 à P faire
       Pour i de 1 à p faire
          S'(t,i)←∅
       Fait
       Pour i de 1 à p faire
          Si ∃j/S(t,i)=S(t-1,j) alors
          -- La tâche n'est pas préemptée et ne doit pas l'être
             Stock←S'(t,j)
             -- On mémorise la tâche qui pourrait déjà être
             -- sur le processeur j afin de la déplacer
             S'(t,j)=S(t,i)
             -- La tâche est affectée sur le même processeur que
             -- précédemment
             Si Stock≠∅ alors
                Trouver k tel que S'(t,k)= ∅
                -- On cherche le 1er processeur libre
                S'(t,k)←Stock
                -- Afin d'y affecter la tâche déplacée
             FinSi
          Sinon
             Trouver k tel que S'(t,k)= ∅
             -- On cherche le 1er processeur libre
             S'(t,k)←S(t,i)
          Finsi
       Fait
    Fait
    Retourner S'
Fin
```

Avec une complexité de $P \times p^3$, cet algorithme élimine les changements de contexte pouvant être évités.

Cependant, même lorsqu'une tâche est préemptée, il peut être intéressant de lui allouer le même processeur lorsqu'elle est à nouveau active, pour profiter de la mémoire cache.

On peut donc améliorer cet algorithme en se basant sur une approche de type *LRU* (*Least Recently Used*) : lorsque une tâche se voit ré-allouer un processeur, le fait de travailler sur le même permet d'aller chercher son contexte dans le cache et d'éviter d'avoir à faire transiter le contexte par la mémoire commune.

4.2. Etude du graphe des marquages

Nous nous intéressons aux mots du centre du langage terminal du RdP. Ceux-ci sont obtenus par construction du graphe des marquages du RdP. Cependant, étant donné que chaque mot est infini, il nous faut borner le graphe des marquages et l'étudier afin de déterminer la profondeur à atteindre et les propriétés du graphe. Notons que nous ne nous intéressons qu'aux séquences valides : dans le graphe des marquages, seuls les états temporellement corrects (i.e. marquages respectant les contraintes terminales) sont stockés.

4.2.1 Tâches simultanées

Lorsque les tâches sont simultanées, c'est à dire lorsque toutes les dates de réveil des tâches sont nulles, toute séquence valide est trivialement cyclique à partir de la date 0, de période *P*. En effet, à la date *0*, toutes les tâches sont activées au même instant, et *P* unités de temps plus tard, le système de tâches se retrouve dans le même état : toutes les tâches sont réactivées au même instant, et leurs instances précédentes sont terminées. Il en résulte que le graphe des marquages peut être borné à un graphe de profondeur *P* où le marquage initial est un marquage d'accueil : tout chemin du graphe part du marquage initial et retourne à ce graphe des marquages après *P* tirs d'ensembles de transitions.

Cette structure permet la mise en œuvre d'algorithmes de recherche de séquences optimales au vu de certains critères (minimisation de temps de réponse, maximisation de la latence,...) comme dans le cas monoprocesseur [3].

4.2.2 Tâches différées

Le problème est plus complexe dans le cas où certaines tâches ont une date de réveil différente de *0*. Pour le cas monoprocesseur, nous avons montré [7] que l'état atteint après le dernier temps creux acyclique avait les mêmes propriétés que l'état initial dans le cas de tâches simultanées. En effet, cet état est atteint toutes les *P* unités de temps pour toute séquence valide.

Cependant, la preuve de ce résultat ne peut être étendue trivialement aux différentes hypothèses multiprocesseur.

Sous les hypothèses $\mathrm{P}\,\overline{\mathrm{M}}$ et $\overline{\mathrm{P}}\ \overline{\mathrm{M}}$, pour des tâches indépendantes, on peut étendre trivialement la preuve de [7] pour montrer que toute séquence d'ordonnancement est cyclique de période $P=PPCM_{i=1..n}\{P_i\}$ à partir de la date du dernier temps creux acyclique. En effet, les tâches étant indépendante, le résultat de cyclicité monoprocesseur s'applique indépendamment à chaque processeur. Le cycle global commence alors dès que le cycle le plus tardif à se mettre en place commence.

Cependant, dès lors que des contraintes de précédence sont en jeu, le dernier temps creux acyclique n'augure pas de l'entrée dans le cycle. Pour illustrer ce fait, considérons une séquence d'ordonnancement *Rate Monotonic* donnée sur la figure 7 du système $S_2=\{\tau_1<0,1,4,4>,\tau_2<0,1,4,4>,\tau_3<0,2,4,4>,\tau_4<0,1,4,4>,\tau_5<2,2,4,4>,\tau_6<0,1,4,4>\}$ où τ_6 précède τ_1 et τ_1 précède τ_4, ordonnancées sous l'hypothèse $\mathrm{P}\,\overline{\mathrm{M}}$ sur deux processeurs, avec les 3 premières tâches affectées au processeur 1, et les 3 suivantes au processeur 2.

La charge de chaque processeur est de 100%, ils sont donc complètement occupés dans la partie cyclique de toute séquence d'ordonnancement. Cependant, la figure 7 montre qu'il existe un et un seul temps creux dans la séquence d'ordonnancement *RM* de S_2 : ce temps creux acyclique est dû à la contrainte de précédence entre τ_1 et τ_4 lors de la montée en charge du système.

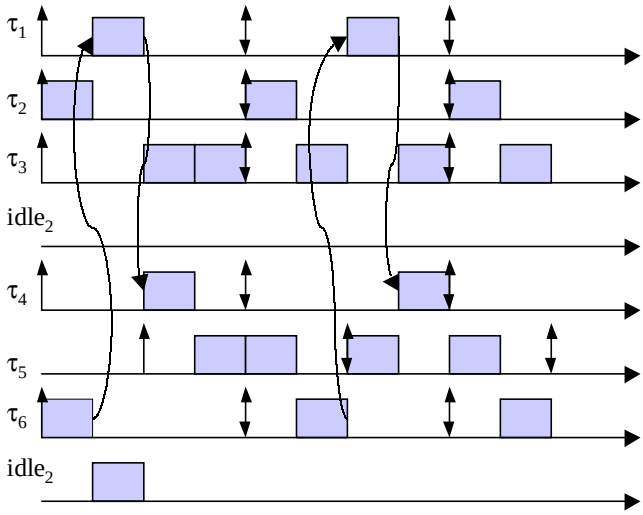

figure 7 : ordonnancement RM du système S_2

La partie cyclique de l'ordonnancement ne commence pas après ce temps creux acyclique mais à la date 4, et a une période égale à P=4.

Enfin, sous l'hypothèse PM_2, la preuve de cyclicité reste ouverte.

5. Discussion

Nous avons montré comment ordonnancer des systèmes de tâches temps réel en environnement multiprocesseur sous l'hypothèse PM_2 à l'aide de réseaux de Petri.

Le modèle RdP pour les hypothèses $\overline{M}$ découle de façon immédiate du modèle d'ordonnancement monoprocesseur de [9] : ce modèle consiste à considérer le modèle de tâches monoprocesseur pour chaque processeur, puis de fusionner les transitions *RTC*.

Ce modèle permet d'obtenir des séquences d'ordonnancement optimales au vu de différents critères lorsque les tâches sont simultanées.

Cependant, contrairement au cas monoprocesseur, le problème de la cyclicité des ordonnancements de tâches différées reste ouvert, et ne nous permet pas pour l'heure d'utiliser les algorithmes de recherche de séquences optimales dans le graphe des marquages obtenu. Enfin, différentes techniques d'optimisations doivent être étudiées afin de réduire en pratique la complexité spatiale et temporelle théorique importante de cette méthode.

6. Bibliographie

[1] J. Blazewicz, *Deadline scheduling of tasks - a survey*, Foundations of Control Engineering, 1 (4), pp. 203-216, 1976.

[2] J. Carlier and P. Chretienne, *Timed Petri net schedules*, Advances in Petri Nets 1988, Lecture Notes in Computer Science, 340 , pp. 62-84, 1988.

[3] A. Choquet-Geniet, E. Grolleau and F. Cottet, *Etude hors-ligne d'une application temps réel à contraintes strictes*, Technique et Science Informatiques, TSI, 19 (10), pp. 1373-1398, 2000.

[4] P. Chrétienne, *Les réseaux de Petri temporisés*, Université Paris VI, 1983.

[5] E. M. Clarke, E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications,* ACM Transactions on Programming Languages and Systems, 8 (2), pp. 244-263, 1986.

[6] M. L. Dertouzos and A. K. Mok, *Multiprocessor on-line scheduling of hard real-time tasks,* IEEE Transactions on Software Engineering, 15 (12), pp. 1497-1506, 1989.

[7] E. Grolleau and A. Choquet-Geniet, *Cyclicité des ordonnancements de systèmes de tâches périodiques différées,* in Teknea , *Real-Time Systems, RTS'2000*, Paris, 2000.

[8] E. Grolleau and A. Choquet-Geniet, *Off-line computation of real-time schedules by means of Petri nets, Workshop On Discrete Event Systems, WODES2000*, Kluwer Academic Publishers, Ghent, Belgium, pp. 309-316, 2000.

[9] E. Grolleau and A. Choquet-Geniet, *Scheduling real-time systems by means of Petri nets, IFAC 25th Workshop on Real-Time Programming, WRTP'00,* Palma de Mallorca, pp. 95-100, 2000.

[10] J. Jonsson, *Effective complexity reduction for optimal scheduling of distributed real-time applications,* in IEEE , *International Conference on Distributed Computing Systems,* Austin, Texas, pp. 360-369, 1999.

[11] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines,* in J. V. Leeuwen, *Algorithms and Complexity,* MIT Press, pp. 869-935, 1990.

[12] G. Koren, E. Dar and A. Amir, *The power of migration in multiprocessor scheduling of real-time systems,* Siam Journal of Computing, 30 (2), pp. 511-527, 2000.

[13] M. Lin, L. Kerlsson and L. T. Yang, *Heuristic techniques: Scheduling partially ordered tasks in a multi-processor environment with Tabu search and Genetic Algorithms, Seventh International Conference on Parallel and Distributed Systems, ICPADS'00*, Iwate, Japan, 2000.

[14] C. L. Liu and J. W. Layland, *Scheduling algorithms for multiprogramming in real-time environment,* Journal of the ACM, 20 (1), pp. 46-61, 1973.

[15] J. W. S. Liu and R. Ha, *Efficient methods of validating timing constraints,* in S. H. Son, *Advances in Real-Time Systems,* pp. 199-223, 1995.

[16] A. K. Mok and M. L. Dertouzos, *Multiprocessor scheduling in a hard real-time environment, 7th Texas Conference on Computer Systems*, pp. 5.1-5.12, 1978.

[17] C. Ramchandani, *Analysis of asynchronous concurrent systems by timed Petri nets,* MIT, Cambridge, 1974.

[18] J. A. Stankovic, M. Spuri, M. D. Natale and G. Buttazzo, *Implications of classical scheduling results fo real-time systems,* IEEE Computer, 28 (6), pp. 1-24, 1995.

[19] J. Xu, *Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations,* IEEE Transactions on Software Engineering, 19 (2), pp. 139-153, 1993.