Chapitre 2

Ordonnancement temps réel multiprocesseur

Je dédie ce chapitre à mon grand-père,
Robert Gaston PIEREUSE.

2.1. Introduction

Dans ce chapitre nous nous intéressons, comme dans les précédents, aux systèmes temps réel dont le bon fonctionnement dépend non seulement des résultats des calculs, mais également des instants auxquels ces résultats sont produits. Ces systèmes jouent un rôle important dans notre société, les applications comprennent : les problèmes de trafic aérien, le contrôle de centrales nucléaires, les communications multimédia, la robotique, les systèmes embarqués, *etc*.

Nous allons plus particulièrement considérer dans ce chapitre des architectures composées de *plusieurs* processeurs pour l'exécution du système temps réel. Les applications temps réel complexes nécessitent en effet soit plusieurs processeurs pour satisfaire les échéances soit des architectures monoprocesseur très puissantes, coûteuses et volumineuses comme les superordinateurs. Il est reconnu aujourd'hui, que le coût d'une architecture multiprocesseur (c.-à-d. composée d'une collection de processeurs peu puissants) est bien moindre que celui d'une architecture monoprocesseur (par exemple un superordinateur) pour une puissance de calcul (totale) équivalente ; ceci justifie par des raisons économiques l'intérêt d'étudier les problèmes d'ordonnancement multiprocesseur. Notons aussi que pour des raisons pratiques, p. ex. pour les systèmes embarqués, il sera préférable et parfois même crucial de distribuer les

Chapitre rédigé par Joël GOOSSENS.

calculs sur une plate-forme composées de plusieurs processeurs, ces derniers étant souvent spécialisés.

L'objet de ce chapitre n'est certainement pas d'étudier de manière exhaustive les techniques existantes en ordonnancement multiprocesseur, mais de présenter à la fois les techniques classiques, déjà bien établies et d'autre part d'initier le lecteur aux techniques plus récentes qui ont permis une avancée importante dans les solutions apportées à l'ordonnancement multiprocesseur. En particulier, nous allons insister dans ce chapitre sur les techniques par augmentation de ressources.

Ce chapitre est organisé de la manière suivante : dans la section 2.2 nous allons introduire notre modèle de calcul et nous donnerons les définitions et concepts de base ; dans la section 2.4, nous présenterons une taxinomie des plates-formes multiprocesseurs ; dans la section 2.5 nous introduirons les différentes familles d'algorithmes d'ordonnancement multiprocesseur ; nous aborderons ensuite, dans la section 2.6, avec plus de détails, les techniques d'ordonnancement globales, et, en particulier dans la section 2.6.3, les techniques par augmentation de ressources ; nous présenterons les algorithmes PFAIR qui s'appliquent dans un cas assez particulier mais qui fournissent un résultat d'optimalité dans la section 2.6.4 ; enfin avant de conclure nous introduirons brièvement les techniques plus classiques et antérieures de partitionnement à la section 2.7.

2.2. Modélisation des applications

Dans cette section, nous allons introduire le modèle de calcul afin de représenter les applications temps réel. En particulier, le modèle de *tâches périodiques* sera présenté, ce modèle déterministe de la charge, est l'un des plus populaire dans la littérature, la plupart des algorithmes d'ordonnancement sont basés sur ce modèle. Bien que par le passé, dans les problèmes d'ordonnancement monoprocesseur, les chercheurs faisaient peu (ou pas) de distinction entre les modèles (strictement) périodique et *sporadique* ; nous verrons que, dans le cas multiprocesseur, cette distinction est fondamentale.

Avant de définir les modèles périodique et sporadique proprement dit, nous introduisons un modèle préalable et plus général que ces derniers. En effet, nous allons auparavant introduire la notion de *travail* et d'*instance* temps réel.

DÉFINITION 2.1.– *Un* travail j *sera caractérisé par le tuple* (a, e, d). *Un instant d'arrivée* a, *un temps d'exécution* e *et une échéance absolue* d. *Le travail* j *doit recevoir* e *unités d'exécution dans l'intervalle* $[a, d)$. *Une* instance *temps réel est une collection (finie ou infinie) de travaux :* $J = \{j_1, j_2, \ldots\}$.

D'ordinaire, les applications temps réel sont constituées de calculs (ou d'opérations) dont la nature est *récurrente*. Nous distinguons par ailleurs les tâches périodiques des tâches sporadiques.

DÉFINITION 2.2.– *Une* tâche périodique τ_i *est caractérisée par le tuple* (O_i, T_i, D_i, C_i)*, où*

– *la* date d'arrivée O_i*, est l'instant du premier travail pour la tâche* τ_i *;*

– *le* temps d'exécution C_i*, qui spécifie une limite supérieure sur le temps d'exécution de chaque travail de la tâche* τ_i *;*

– *l'*échéance *relative* D_i*, dénote la séparation entre l'arrivée du travail et l'échéance (un travail qui arrive à l'instant* t *a une échéance à l'instant* $t + D_i$*) ;*

– *une* période T_i *dénotant la durée qui sépare deux arrivées successives de travaux pour* τ_i*.*

Nous pouvons constater que la notion de tâche périodique est moins générale que celle de travail dans la mesure où une tâche périodique $\tau_i = (O_i, T_i, D_i, C_i)$ définit un nombre infini de travaux, chacun avec le même temps d'exécution, arrivant à chaque instant $O_i + kT_i$ pour tout entier $k \geq 0$, le travail arrivant à l'instant $O_i + kT_i$ devant se terminer avant ou à l'instant $O_i + kT_i + D_i$.

Considérons à présent la notion de tâche *sporadique* :

DÉFINITION 2.3.– *Une* tâche sporadique τ_i *est caractérisée par le triplet* (T_i, D_i, C_i)*, où*

– *le* temps d'exécution C_i*, qui spécifie une limite supérieure sur le temps d'exécution de chaque travail de la tâche* τ_i *;*

– *l'*échéance *relative* D_i*, dénote la séparation entre l'arrivée du travail et l'échéance (un travail qui arrive à l'instant* t *a une échéance à l'instant* $t + D_i$*) ;*

– *une* période T_i *dénotant la durée* minimale *qui sépare deux arrivées successives de travaux pour* τ_i*.*

À la différence de la notion de tâche périodique, la période d'une tâche sporadique correspond à la durée *minimale*, plutôt qu'*exacte*, qui sépare deux arrivées successives de travaux d'une même tâche.

Un système périodique (ou sporadique) τ est constitué d'une collection *finie* de tâches périodiques (ou sporadiques) $\tau = \{\tau_1, \ldots, \tau_n\}$.

Trois grandeurs nous seront utiles dans les sections suivantes : premièrement, le *facteur d'utilisation d'une tâche* qui est le quotient entre son temps d'exécution et sa période, plus formellement $U(\tau_i) \stackrel{\text{def}}{=} C_i/T_i$. Ce qui correspond, du moins pour les systèmes monoprocesseurs, à la portion du temps processeur utilisée par la tâche, sous l'hypothèse que le système est ordonnançable et pour des durées de temps suffisamment élevées. Par extension, nous définissons *l'utilisation du système* comme étant la somme des utilisations de chacune des tâches, plus formellement

$U(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U(\tau_i)$. De nombreuses conditions d'ordonnançabilité sont basées sur cette grandeur, par exemple, il n'est pas difficile d'admettre que $U(\tau) \leq 1$ est une condition nécessaire (et non suffisante) pour l'ordonnançabilité des systèmes périodiques sur des plates-formes monoprocesseurs. Par extension, la condition $U(\tau) \leq m$ est une condition nécessaire pour l'ordonnançabilité de τ sur une plate-forme composée de m processeurs (de capacité 1, *cf. infra*). Enfin, le plus grand facteur d'utilisation caractérisera souvent l'ensemble périodique, plus formellement $U_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} U(\tau_i)$.

Cas particuliers de systèmes périodiques. Pour des raisons d'ordre pratique et théorique, il est intéressant de faire la distinction entre les types de tâches périodiques (ou sporadiques) suivants :

échéance sur requête, où l'échéance de chaque tâche coïncide avec la période (c.-à-d., $D_i = T_i \ \forall i$, chaque travail de τ_i doit se terminer avant l'occurrence du travail suivant de τ_i) ;

échéance contrainte, où l'échéance n'est pas supérieure à la période (c.-à-d., $D_i \leq T_i \ \forall i$) ;

échéance arbitraire, s'il n'y a pas de contrainte entre l'échéance et la période ;

à départ simultané, si toutes les tâches démarrent au même instant, ce qui correspond sans nuire à la généralité au cas $O_i = 0 \ \forall i$.

Hypothèses. Dans ce chapitre, nous supposons que les tâches et les travaux sont *indépendants*, c.-à-d., qu'à l'exclusion du (ou des) processeur(s), les tâches ne sont pas en concurrence sur des ressources communes (comme de la mémoire, des périphériques, *etc.*), il n'y a pas de section critique, il n'y a pas de communication entre les tâches, enfin il n'y a pas davantage de contrainte de préséance.

2.3. Ordonnancement

L'algorithme d'ordonnancement est responsable d'attribuer, au cours du temps, le (ou les) processeur(s) aux travaux actifs[1] afin de satisfaire toutes les échéances (ce qui n'est pas nécessairement possible). Les algorithmes d'ordonnancement que nous considérons ici attribuent des priorités aux tâches (ou aux travaux), le système attribue les ressources (typiquement le(s) processeur(s)) aux travaux actifs le(s) plus prioritaire(s). À cet égard, nous allons distinguer trois types d'algorithmes d'ordonnancement, du moins général au plus général.

1. Un travail $j = (a, e, d)$ est actif à l'instant t si $t \geq a$ et si le travail n'a pas déjà reçu e unités d'exécutions depuis son arrivée.

Les algorithmes à priorité fixe au niveau des tâches, assignent des priorités fixes à chaque tâche lors de la *conception* du système ; de plus, pendant l'exécution du système, chaque travail *hérite* de la priorité de sa tâche. Dès lors, tous les travaux issus d'une même tâche ont la même priorité. Un exemple d'algorithme à priorité fixe au niveau des tâches, pour des tâches périodiques, est RM (*Rate Monotonic*, [LIU 73]).

Les algorithmes à priorité fixe au niveau des travaux, assignent pendant l'exécution du système, des priorités fixes aux travaux. C.-à-d. que les travaux d'une même tâche peuvent avoir des priorités différentes mais la priorité d'un travail ne varie pas au cours du temps. EDF (*Earliest Deadline First*, [LIU 73]) est un tel algorithme.

Les algorithmes à priorité dynamique au niveau des travaux, définissent à chaque instant une priorité pour chaque travail (actif), c'est le cas le plus général, il n'y aucune restriction et la priorité d'un travail peut, par exemple, varier au cours du temps. LLF (*Least Laxity First*, [MOK 78]) est un exemple.

Problèmes d'ordonnancement temps réel. La théorie de l'ordonnancement des systèmes temps réel se focalise principalement sur deux problèmes :

1) Le problème de *faisabilité* : étant donné les spécifications des tâches (ou des travaux) et les contraintes sur l'environnement d'ordonnancement (p. ex. si les priorités sont fixes/dynamiques au niveau des tâches/travaux, si les préemptions sont admises, *etc.*), il s'agit de *déterminer* l'existence d'un ordonnancement qui satisfasse toutes les échéances.

2) Le problème d'*ordonnançabilité* en-ligne : étant donné les spécifications des tâches (et des contraintes de l'environnement) supposées être faisables, il s'agit de *déterminer* un algorithme d'ordonnancement qui construit un ordonnancement qui satisfasse toutes les échéances.

Ces deux problèmes sont de natures bien différentes, en particulier en regard de la complexité algorithmique ; cependant le premier problème (la faisabilité) perd de son intérêt si l'on a connaissance d'un algorithme d'ordonnancememt *optimal* avec la définition suivante :

DÉFINITION 2.4.– *Un algorithme d'ordonnancement R est* optimal *si, un système faisable est ordonnançable sous l'égide de l'algorithme R.*

Nous verrons que, malheureusement dans le cas multiprocesseur, il n'est pas possible en toute généralité de définir un algorithme d'ordonnancement optimal.

À travers les différents chapitres de cet ouvrage (p. ex., les chapitres 1, 4 et 6), nous pouvons constater que depuis les travaux initiaux de LIU et de LAYLAND (1973), une

impressionnante littérature existe à propos de l'ordonnancement *monoprocesseur*, elle concerne l'existence d'algorithmes d'ordonnancement optimaux, l'existence de bons tests (c.-à-d. avec des complexités polynomiales voire pseudo-polynomiales) pour les problèmes de faisabilité ou d'ordonnançabilité tout en considérant des tâches (ou des travaux) dépendants. Outre le respect strict des échéances, des qualités de service peuvent être considérées (p. ex., pour économiser les batteries des systèmes embarqués, sujet qui sera traité au chapitre 4). Des solutions sophistiquées pour l'ordonnancement *conjoint* de tâches critiques (temps réel) et non critiques existent, *etc.* En revanche, nous disposons à ce jour de peu de résultats à propos de l'ordonnancement multiprocesseur.

2.4. Taxinomie des plates-formes multiprocesseurs

Après avoir présenté la modélisation des applications temps réel et introduit la notion d'ordonnancement, nous allons décrire les ressources du système qui permettront d'exécuter les applications à savoir le(s) processeur(s). À l'opposé des systèmes monoprocesseurs, sur une plate-forme multiprocesseur, les applications disposent de *plusieurs* processeurs *simultanément* pour réaliser leurs calculs. Cette simultanéité – ce parallélisme – est cependant *limitée* :

– un processeur exécute au plus un travail à chaque instant ;

– un travail s'exécute sur au plus un processeur à chaque instant.

Nous allons distinguer trois types de plates-formes, de la moins générale à la plus générale.

Processeurs identiques. Plate-forme multiprocesseur, constituée de plusieurs processeurs *identiques*, dans la mesure où les processeurs sont interchangeables et ont la même puissance de calcul.

Processeurs uniformes. En revanche, chaque processeur d'une plate-forme *uniforme* est caractérisé par sa capacité de calcul, avec l'interprétation suivante : lorsqu'un travail s'exécute sur un processeur de capacité de calcul s pendant t unités de temps, il réalise $s \times t$ unités de travail.

Processeurs indépendants. Pour ce type de plate-forme on définit un taux d'exécution $r_{i,j}$ associé à chaque couple travail-processeur (J_i, P_j), avec l'interprétation suivante : le travail J_i réalise $(r_{i,j} \times t)$ unités de travail lorsqu'il s'exécute sur le processeur P_j pendant t unités de temps. Dès lors la vitesse de progression sur un même processeur varie éventuellement d'un travail à l'autre. Ce modèle permet, en outre, de considérer des processeurs *spécialisés*, qui ne peuvent que réaliser certaines tâches ; en particuliers si la grandeur $r_{i,j}$ est nulle cela signifie que le processeur P_j est incapable de prendre en charge le travail J_i.

Les plates-formes identiques correspondent à des plates-formes *homogènes*, les plates-formes uniformes et indépendantes correspondent à des plates-formes *hétérogènes*.

Par ailleurs, nous considérons des systèmes *fortement couplés*. La caractéristique principale de ce type de système est l'existence d'une base commune du temps (ce sera requis pour l'ordonnancement global des travaux *cf. infra*), une mémoire commune et par conséquent une vue globale de l'état du système à chaque instant.

2.5. Familles d'algorithmes multiprocesseurs : généralités

Dans cette section nous allons distinguer principalement deux techniques d'ordonnancement multiprocesseur : les stratégies *globales* et les stratégies par *partitionnement*. Notons qu'il est parfaitement possible de définir des stratégies intermédiaires (p. ex. semi-partitionnées), le lecteur intéressé pourra consulter [LEU 04], chapitre 30, pour une catégorisation.

Nous supposerons, dans cette description, de devoir ordonnancer un ensemble de n tâches périodiques (ou sporadiques) sur une architecture multiprocesseur composée de m processeurs identiques.

Stratégie par partitionnement, il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints : $\tau^1, \tau^2, \ldots, \tau^m$ (avec $(\bigcup \tau_i) = \tau$) et d'ordonnancer ensuite chaque sous-ensemble τ^i sur le processeur P_i avec une stratégie d'ordonnancement *« locale »* monoprocesseur. Les tâches sitôt assignées aux processeurs ne sont pas autorisées à *migrer* d'un processeur à l'autre.

Stratégie globale, il s'agit d'appliquer sur l'entièreté de la plate-forme multiprocesseur une stratégie d'ordonnancement globalement et d'attribuer à chaque instant les m processeurs au m tâches les plus prioritaires (si toutefois il y a au moins m tâches actives, sinon des processeurs seront laissés oisifs). Par exemple en utilisant *globalement* RM, EDF, *etc.* Pour les techniques globales, outre la préemption des tâches, on autorise aussi la *migration* de ces dernières. Une tâche peut dès lors commencer son exécution sur un premier processeur (disons P_i), être préemptée par l'arrivée d'une nouvelle tâche plus prioritaire pour reprendre son exécution sur un autre processeur (disons P_j avec $j \neq i$), ce phénomène est appelé la *migration* de tâche et caractérise les stratégies globales.

Notons que nous avons supposé ordonnancer des tâches (périodiques ou sporadiques), nous pouvons imaginer adapter ces définitions pour l'ordonnancement de travaux, ce qui est immédiat pour les stratégies globales et sera plus délicat pour le stratégies par partitionnement. De même nous avons supposé disposer de processeurs

identiques, il est parfaitement possible d'envisager d'adapter ces définitions pour des plates-formes hétérogènes.

Approches incomparables. Il est important de noter que dans la plupart des cas, les stratégies globales et partitionnées sont *incomparables*. Par exemple, LEUNG et WHITEHEAD, dans [LEU 82], ont montré que pour des priorités fixes au niveau des tâches, les deux approches sont incomparables pour l'ordonnancement de tâches périodiques, dans la mesure où *(i)* il y a des systèmes périodiques qui sont ordonnançables utilisant m processeurs identiques à l'aide d'une approche partitionnée mais pour lesquelles aucune assignation de priorités fixes aux tâches, à l'aide d'une stratégie globale, disposant des m mêmes processeurs ordonnance le système et *(ii)* il y a des systèmes périodiques ordonnançables avec une approche globale et des priorités fixes aux tâches, sur m processeurs mais pour lesquelles aucun partitionnement de m sous-ensembles n'existe. Ce résultat souligne l'intérêt d'étudier les deux approches : l'approche partitionnée et l'approche globale.

Hypothèses. Dans la suite de ce chapitre et sauf mention contraire nous adoptons les hypothèses suivantes : outre l'hypothèse déjà introduite et qui veut que nous considérons l'ordonnancement de tâches *indépendantes*, nous supposons par ailleurs que le système est *préemptif*, nous supposons de surcroît que les temps de préemption sont négligeables (par rapport aux durées d'exécution des tâches) et pour les stratégies globales nous supposons que les temps de migration sont également négligeables.

2.6. Techniques globales

2.6.1. *Optimalité inexistante*

Nous allons commencer par présenter un premier résultat négatif qui concerne l'absence d'algorithme d'ordonnancement en-ligne et optimal. Mais tout d'abord définissons la notion d'algorithme en-ligne.

DÉFINITION 2.5.– *Les algorithmes d'ordonnancement* en-ligne *prennent leurs décisions à chaque instant sur la base des caractéristiques des travaux actifs arrivés jusque là, sans connaissance des travaux qui arriveront dans le futur.*

HONG et LEUNG, dans [HON 88], ont montré l'absence d'algorithme en-ligne et optimal.

THÉORÈME 2.1.– *Pour tout $m > 1$, aucun algorithme d'ordonnancement en-ligne et optimal ne peut exister pour des systèmes avec deux ou plus d'échéances distinctes.*

PREUVE : Nous prouvons la propriété pour le cas $m = 2$. Il sera facile de constater que la preuve peut être généralisée pour $m > 2$. Supposons qu'il existe un algorithme

en-ligne et optimal pour deux processeurs et considérons le scénario suivant. À l'instant 0, trois travaux J_1, J_2 et J_3 arrivent avec $d_1 = d_2 = 4$, $d_3 = 8$, $e_1 = e_2 = 2$ et $e_3 = 4$. L'algorithme ordonnance ces trois travaux sur deux processeurs à partir de l'instant 0. Nous avons deux cas à considérer, suivant que J_3 s'exécute ou non dans l'intervalle $[0, 2)$.

Cas 1 : J_3 s'exécute dans l'intervalle $[0, 2)$. Dans ce cas, au moins l'un des deux travaux J_1 ou J_2 n'a pas terminé son exécution à l'instant 2. Puisque J_1 et J_2 sont identiques, nous supposons que c'est J_2 qui n'a pas terminé à l'instant 2. À présent, considérons le scénario où J_4 et J_5 arrivent à l'instant 2 avec $d_4 = d_5 = 4$ et $e_4 = e_5 = 2$. Clairement, l'ordonnancement construit par l'algorithme optimal en-ligne n'est pas ordonnançable. Cependant, le système est faisable sur deux processeurs : mais il ne faut pas faire progresser le travail J_3 dans l'intervalle $[0, 2)$.

Cas 2 : J_3 ne s'exécute pas dans l'intervalle $[0, 2)$. Dans ce cas, à l'instant 4, J_3 s'est exécuté pour au plus deux unités. Donc, à l'instant 4, le temps de calcul restant de J_3 est au moins de deux unités. À présent, considérons le scénario où J_4 et J_5 arrivent à l'instant 4 avec $d_4 = d_5 = 8$ et $e_4 = e_5 = 4$. À nouveau, l'algorithme optimal en-ligne échoue dans la construction d'un ordonnancement alors que le système est faisable sur deux processeurs : mais il faut faire progresser le travail J_3 dans l'intervalle $[0, 2)$. □

Remarquons que si nous considérons l'ordonnancement de tâches strictement périodiques, l'arrivée des travaux futurs est connue, d'une certaine manière nous connaissons le futur pour ces systèmes. Pour cette raison (et ce n'est pas contradictoire) nous disposons d'algorithmes optimaux pour l'ordonnancement de tâches périodiques en multiprocesseur. Ce sera la cas des algorithmes PFAIR présentés à la section 2.6.4. Cependant, pour des tâches sporadiques l'absence d'algorithme en-ligne et optimal persiste et d'autres solutions que PFAIR (qui n'offre pas une solution satisfaisante pour l'ordonnancement de systèmes sporadiques) existent, nous introduirons ces méthodes à la section 2.6.3.

2.6.2. *Anomalies d'ordonnancement*

Outre l'absence d'un algorithme (en-ligne) optimal, il y a un deuxième phénomène qui apparaît en ordonnancement multiprocesseur à savoir les anomalies d'ordonnancement. Nous parlons d'anomalies dans la mesure où un changement qui est intuitivement positif dans un système ordonnançable peut le rendre non ordonnançable. Remarquons que ce phénomène n'est pas intrinsèque à l'ordonnancement multiprocesseur et apparaît aussi, par exemple, en ordonnancement monoprocesseur *non-préemptif* (voir la section 1.3.4).

Un changement intuitivement positif, est par exemple un changement qui diminue le facteur d'utilisation d'une tâche comme augmenter la période ou diminuer un temps

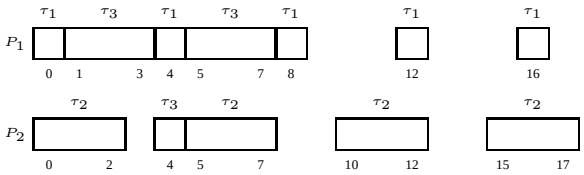

Figure 2.1. *Le système est ordonnançable*

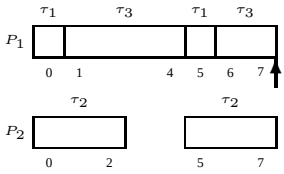

Figure 2.2. *Le système rate une échéance à l'instant 8*

d'exécution (ce qui est équivalent à augmenter la vitesse des processeurs). Cela peut être aussi de donner plus de ressources au système comme ajouter des processeurs.

Les algorithmes d'ordonnancement multiprocesseurs sont sujets à des anomalies. Nous allons illustrer ce phénomène à l'aide d'un exemple, pour les algorithmes à priorité fixe. Nous allons considérer un système qui est ordonnançable avec une assignation de priorité mais lorsqu'une période est augmentée et les priorités conservées ce système n'est plus ordonnançable !

Soit le système $\tau_1 = (T_1 = 4, D_1 = 2, C_1 = 1), \tau_2 = (T_2 = 5, D_2 = 3, C_2 = 3), \tau_3 = (T_3 = 20, D_3 = 8, C_3 = 7)$. Ce système est ordonnançable sur 2 processeurs (en donnant la plus haute priorité à τ_1, une priorité intermédiaire à τ_2 et la plus basse priorité à τ_3) comme l'illustre la Figure 2.1. En revanche, si l'on augmente la période de τ_1 de 4 à 5 (et que l'on ne change pas les priorités), le système résultant n'est *pas* ordonnançable : τ_3 rate son échéance à l'instant 8, comme l'illustre la Figure 2.2.

Examinons plus attentivement ces deux ordonnancements et focalisons-nous sur l'intervalle $[0, 7]$: dans l'ordonnancement de la Figure 2.1, il y a dès l'instant 3 et pendant une unité de temps un processeur oisif, dans l'ordonnancement de la Figure 2.2, il y a dès l'instant 2 et pendant *deux* unités de temps un processeur oisif. Dans les deux cas, τ_3 ne peut pas progresser sur le processeur oisif en particulier car il s'exécute déjà sur l'autre processeur et que le parallélisme au sein d'un travail n'est pas autorisé. Dès lors, dans l'ordonnancement de la Figure 2.1, le premier travail de τ_3 atteint de justesse son échéance et rate son échéance dans l'ordonnancement de la Figure 2.2.

Nous disposons à présent des éléments pour justifier la différence importante qui distingue les systèmes périodiques et sporadiques en multiprocesseur ; alors que cela n'avait peu (ou pas) d'importance en ordonnancement monoprocesseur puisque (i) il existe des algorithmes en-ligne optimaux (comme EDF) et (ii) qu'étudier le cas (strictement) périodique est suffisant, car il est plus pessimiste que le cas sporadique. Nous venons de voir que ce n'est malheureusement plus le cas pour les systèmes multiprocesseurs.

2.6.3. *Techniques par augmentation de ressources*

Nous considérons dans cette section, l'ordonnancement par des techniques globales et en-ligne, les décisions d'ordonnancement sont prises à chaque instant basées sur les caractéristiques des travaux actifs et sans connaissance des travaux futurs. Pour des architectures monoprocesseurs, plusieurs algorithmes en-ligne, comme EDF et LLF sont connus comme étant *optimaux* dans la mesure où si un ensemble de travaux est faisable alors l'algorithme pourra ordonnancer cet ensemble de travaux (voir sections 1.2.1 à 1.2.4). Sur des plates-formes multiprocesseurs, cependant il n'existe pas d'algorithmes en-ligne optimaux. Une avancée considérable dans l'ordonnancement en-ligne fut obtenue par PHILIPS, STEIN, TORNG et WEIN [PHI 97], ces chercheurs ont explorés l'utilisation d'une technique par *augmentation de ressources*. PHILIPS *et al.* ont étudié la manière dont un algorithme en-ligne, lorsque l'on lui fournit des processeurs plus puissants que ceux nécessaires pour la faisabilité, peut se comporter par rapport à ce résultat négatif d'absence d'optimalité. Dans [PHI 97], les auteurs ont montré qu'une application immédiate de ce résultat appliqué à EDF pour des plates-formes identiques est la suivante :

Si un ensemble de travaux est faisable sur m processeurs identiques, alors le même ensemble de travaux est ordonnançable avec EDF sur m processeurs identiques dont la puissance de calcul de chacun est $(2 - \frac{1}{m})$ fois plus grande que le système d'origine.

Il s'agit d'une technique par augmentation de ressources dans la mesure où l'on fournit plus de ressources (puisque les processeurs sont plus rapides) pour l'ordonnançabilité (avec EDF) que nécessaire pour la faisabilité.

En exploitant la même idée, d'autres résultats ont émané de la communauté scientifique, nous allons dans la suite de cette section, présenter en détail l'un d'entre eux pour l'ordonnancement de tâches périodiques à échéance sur requête pour des plates-formes uniformes et sous l'égide de l'algorithme d'ordonnancement RM appliqué globalement sur la plate-forme. Nous souhaitons en effet donner au lecteur, pour ces techniques plus récentes, une bonne intuition de leurs utilisations et propriétés. Nous nous baserons principalement sur [BAR 03, FUN 01] dans cette section.

Nous commençons par caractériser plus formellement les plates-formes uniformes :

DÉFINITION 2.6.– *Soit π dénotant une* plate-forme uniforme*.*

– Le nombre de processeurs de π étant dénoté par $m(\pi)$.

– Pour tout i, $1 \leq i \leq m(\pi)$, la vitesse (la capacité de calcul) du i^e processeur le plus rapide de π est dénotée par $s_i(\pi)$.

– La capacité de calcul totale de π est dénotée par $S(\pi)$: $S(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{m(\pi)} s_i(\pi)$.

La notion d'algorithme *conservatif* (ou expédient) est bien défini dans la littérature, cependant pour des systèmes multiprocesseurs *non-identiques* elle est ambiguë et nécessite une clarification :

DÉFINITION 2.7.– *Un algorithme d'ordonnancement pour un plate-forme uniforme est dit* conservatif *s'il satisfait les conditions suivantes :*

1) Il ne laisse jamais un processeur oisif lorsqu'il y a des travaux en attente.

2) S'il doit mettre des processeurs dans l'état oisif ils devront être les plus lents.

3) Il exécute toujours les travaux les plus prioritaires sur les processeurs les plus rapides.

Remarquons que cette hypothèse est relativement forte, dans la mesure où elle peut induire des *« cascades »* de migrations si par exemple une tâche de priorité élevée se termine, quitte un processeur rapide alors que d'autres tâches moins prioritaires sont encore actives.

Deux grandeurs nous seront utiles dans la suite de cette section : $\lambda(\pi)$ et $\mu(\pi)$ que nous définissons de la manière suivante :

DÉFINITION 2.8.– *Pour une plate-forme uniforme π, nous définissons le paramètre $\lambda(\pi)$ de la manière suivante :* $\lambda(\pi) \stackrel{\text{def}}{=} \max_{i=1}^{m(\pi)} \left\{ \frac{\sum_{j=i+1}^{m(\pi)} s_j(\pi)}{s_i(\pi)} \right\}$.

DÉFINITION 2.9.– *Pour une plate-forme uniforme π, nous définissons le paramètre $\mu(\pi)$ de la manière suivante :* $\mu(\pi) \stackrel{\text{def}}{=} \max_{i=1}^{m(\pi)} \left\{ \frac{\sum_{j=i}^{m(\pi)} s_j(\pi)}{s_i(\pi)} \right\}$.

Ces paramètres mesurent le *« degré »* de similarité entre π et la plate-forme identique dans la mesure où $\lambda(\pi) = (m(\pi) - 1)$ et $\mu(\pi) = m(\pi)$ si π est constituée de processeurs identiques et les deux paramètres deviennent progressivement plus petits lorsque les processeurs ont des vitesses de plus en plus dissemblables, à la limite nous avons $\lambda(\pi) = 0$ et $\mu(\pi) = 1$.

Nous allons à présent définir le *travail* réalisé par un algorithme d'ordonnancement.

DÉFINITION 2.10.– *Soit I un ensemble de travaux et π une plate-forme uniforme. Soit A un algorithme, et t un instant, alors $W(A, \pi, I, t)$ est le travail réalisé par l'algorithme A pour les travaux de I dans l'intervalle $[0, t)$ s'exécutant sur la plate-forme π.*

THÉORÈME 2.2.– *Soit π_o et π_1 deux plates-formes uniformes. Soit A_o un algorithme d'ordonnancement et A_1 un algorithme d'ordonnancement convervatif. Si la condition suivante est satisfaite pour π_o et π_1 :*

$$S(\pi_1) \geq S(\pi_o) + \lambda(\pi_1) \cdot s_1(\pi_o) \tag{2.1}$$

alors pour tout ensemble de travaux I et tout instant $t \geq 0$,

$$W(A_1, \pi_1, I, t) \geq W(A_o, \pi_o, I, t). \tag{2.2}$$

PREUVE : La preuve est par contradiction. Supposons que la propriété soit fausse, c.-à-d. qu'il existe un ensemble de travaux I et un instant pour lequel l'algorithme conservatif A_1 a exécuté sur la plate-forme π_1 strictement moins de travail que l'algorithme A_o sur π_o. Soit $J_a = (r_a, c_a, d_a)$ un travail de I avec la date d'arrivée la plus petite et tel que il existe un instant t_o satisfaisant

$$W(A_1, \pi_1, I, t_o) < W(A_o, \pi_o, I, t_o)$$

avec une quantité de travail réalisée par J_a avec A_1 à l'instant t_o strictement plus petite que celle réalisée avec A_o à l'instant t_o. Étant donné notre choix de r_a, nous devons avoir

$$W(A_1, \pi_1, I, r_a) \geq W(A_o, \pi_o, I, r_a).$$

Dès lors, la quantité de travail réalisée par A_o dans l'intervalle $[r_a, t_o)$ est strictement plus grande que la quantité de travail réalisée par A_1 dans le même intervalle.

Soit x_ℓ qui dénote la somme cumulée des durées dans l'intervalle $[r_a, t_o)$, pendant lesquelles A_1 utilise ℓ processeurs, $1 \leq \ell \leq m(\pi_1)$. Par définition, nous avons que $t_o - r_a = x_1 + x_2 + \cdots + x_{m(\pi_1)}$. Nous allons à présent faire deux observations.

1) Puisque A_1 est un algorithme conservatif, le travail J_a, qui n'a pas terminé son exécution à l'instant t_o dans l'ordonnancement défini par A_1, doit s'exécuter à chaque instant où (au moins) un processeur est oisif dans l'ordonnancement de A_1. Pendant ces instants où ℓ processeurs sont non oisifs, $\ell < m(\pi_1)$, ces processeurs non oisifs ont une capacité de calcul $\geq s_\ell(\pi_1)$. Par conséquent, J_a a exécuté au moins $\left(\sum_{j=1}^{m(\pi_1)-1} x_j s_j(\pi_1)\right)$ unités à l'instant t_o dans l'ordonnancement généré par A_1 sur π_1, alors qu'il aurait exécuté au plus $s_1(\pi_o)\left(\sum_{j=1}^{m(\pi_1)} x_j\right)$ unités dans l'ordonnancement généré par A_o sur π_o. Par conséquence nous obtenons

$$\sum_{j=1}^{m(\pi_1)-1} x_j s_j(\pi_1) < s_1(\pi_o)\left(\sum_{j=1}^{m(\pi_1)} x_j\right). \tag{2.3}$$

Multiplions les deux côtés de l'inégalité 2.3 par $\lambda(\pi_1)$ et notons que

$$(x_j s_j(\pi_1)\lambda(\pi_1)) \geq \left(x_j s_j(\pi_1) \frac{\sum_{k=j+1}^{m(\pi_1)} s_k(\pi_1)}{s_j(\pi_1)} \right) = x_j \left(\sum_{k=j+1}^{m(\pi_1)} s_k(\pi_1) \right), \quad (2.4)$$

nous obtenons

$$\sum_{j=1}^{m(\pi_1)-1} \left(x_j \left(\sum_{k=j+1}^{m(\pi_1)} s_{k(\pi_1)} \right) \right) < s_1(\pi_o)\lambda(\pi_1) \left(\sum_{j=1}^{m(\pi_1)} x_j \right). \quad (2.5)$$

2) La quantité totale de travail réalisée par A_1 sur π_1 dans l'intervalle $[r_a, t_o)$ est donnée par

$$\sum_{j=1}^{m(\pi_1)} \left(x_j \sum_{k=1}^{j} s_k(\pi_1) \right), \quad (2.6)$$

alors que la quantité totale de travail réalisée par A_o sur π_o durant le même intervalle est borné par la capacité de π_o et par conséquent est $\leq (\sum_{j=1}^{m(\pi_1)} x_j) \cdot S(\pi_o)$. Nous obtenons donc l'inégalité suivante

$$\sum_{j=1}^{m(\pi_1)} \left(x_j \sum_{k=1}^{j} s_k(\pi_1) \right) < \left(\sum_{j=1}^{m(\pi_1)} x_j \right) S(\pi_o). \quad (2.7)$$

Sommons 2.4 et 2.7, nous obtenons

$$\begin{aligned}
& \sum_{j=1}^{m(\pi_1)-1} \left(x_j \left(\sum_{k=j+1}^{m(\pi_1)} s_{k(\pi_1)} \right) \right) + \sum_{j=1}^{m(\pi_1)} \left(x_j \sum_{k=1}^{j} s_k(\pi_1) \right) \\
< & \left(\sum_{j=1}^{m(\pi_1)} x_j \right) (s_1(\pi_o)\lambda(\pi_1) + S(\pi_o)) \\
\equiv & \; x_{m(\pi_1)} S(\pi_1) + \sum_{j=1}^{m(\pi_1)-1} [x_j(S(\pi_1))] \\
< & \left(\sum_{j=1}^{m(\pi_1)} x_j \right) (s_1(\pi_o)\lambda(\pi_1) + S(\pi_o)) \\
\equiv & \; x_{m(\pi_1)} S(\pi_1) + \sum_{j=1}^{m(\pi_1)-1} (x_j S(\pi_1)) \\
< & \left(\sum_{j=1}^{m(\pi_1)} x_j \right) (s_1(\pi_o)\lambda(\pi_1) + S(\pi_o)) \\
\equiv & \; S(\pi_1) \cdot \left(\sum_{j=1}^{m(\pi_1)} x_j \right) \\
< & \left(\sum_{j=1}^{m(\pi_1)} x_j \right) (s_1(\pi_o)\lambda(\pi_1) + S(\pi_o)) \\
\equiv & \; S(\pi_1) < s_1(\pi_o)\lambda(\pi_1) + S(\pi_o),
\end{aligned}$$

ce qui contredit l'hypothèse faite dans l'énoncé du théorème. □

Soit un système périodique τ et une plate-forme π et supposons que la relation suivante est satisfaite :

$$S(\pi) \geq 2 \cdot U(\tau) + \mu(\pi) \cdot U_{\max}(\tau). \tag{2.8}$$

Nous nous intéressons à l'ordonnancement de τ sur la plate-forme π pour la stratégie globale RM. Sans nuire à la généralité nous pouvons supposer que l'algorithme RM résout les ambiguïtés en donnant une plus grande priorité à τ_i que $\tau_{i+1}, \forall i, 1 \leq i < n$. Remarquons que l'ordonnançabilité pour l'algorithme RM de τ_k dépend seulement

des travaux de $\{\tau_1, \tau_2, \ldots, \tau_k\}$ et n'est pas affectée par les travaux $\tau_{k+1}, \ldots, \tau_n$. Pour $k = 1, 2, \ldots, n$ nous définissons le sous-ensemble $\tau^{(k)}$ de la manière suivante :

$$\tau^{(k)} \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_k\}.$$

LEMME 2.1.– *Le système $\tau^{(k)}$ est faisable sur une plate-forme π_o avec les propriétés suivantes :*

1) $S(\pi_o) = U(\tau^{(k)})$ *et*

2) $s_1(\pi_o) = U_{\max}(\tau^{(k)})$.

PREUVE : Choisissons π_o, une plate-forme uniforme composée de k processeurs, avec une puissance de calcul égale $C_1/T_1, C_2/T_2, \ldots, C_k/T_k$, respectivement. Il est aisé de voir que $\tau^{(k)}$ est faisable sur π_o : un algorithme optimal, OPT, pourra simplement ordonnancer chaque tâche exclusivement sur le processeur dont la capacité correspond au facteur d'utilisation de la tâche. □

À partir de la condition 2.1, du lemme 2.1 précédent et du fait que RM est un algorithme conservatif, si les paramètres $S(\pi)$ et $\lambda(\pi)$ satisfont

$$S(\pi) \geq U(\tau^{(k)}) + \lambda(\pi) U_{\max}(\tau) \tag{2.9}$$

alors nous devons avoir que

$$\begin{aligned} & W(\text{RM}, \pi, \tau^{(k)}, t) \geq W(\text{OPT}, \pi_o, \tau^{(k)}, t) \\ \equiv \quad & \left(W(\text{RM}, \pi, \tau^{(k)}, t) \geq t \cdot \left(\sum_{j=1}^{k} U(\tau_j) \right) \right) \end{aligned} \tag{2.10}$$

où π_o et OPT sont définis dans la preuve du lemme 2.1 précédant. Rappelons-nous que nous supposons que la condition 2.8 est satisfaite. Puisque

$$\Big(2 \cdot U(\tau) \geq 2 \cdot U(\tau^{(k)}) \geq U(\tau^{(k)})\Big) \quad \text{et} \quad \left(\mu(\pi) \geq \lambda(\pi)\right),$$

nous pouvons conclure que

$$\begin{aligned} & S(\pi) \geq 2 \cdot U(\tau) + \mu(\pi) \cdot U_{\max}(\tau) \ \text{(par la condition 2.8)} \\ \Rightarrow \quad & S(\pi) \geq U(\tau^{(k)}) + \lambda(\pi) \cdot U_{\max}(\tau^{(k)}). \end{aligned}$$

Dès lors, l'inégalité 2.10 est vérifiée pour tout $t \geq 0$; c.-à-d., à chaque instant t, la quantité de travail réalisée par $\tau^{(k)}$ par l'algorithme RM sur π est au moins celle réalisée pour $\tau^{(k)}$ par l'algorithme OPT sur π_o.

LEMME 2.2.– *Tout les travaux de τ_k respectent leurs échéances lorsque $\tau^{(k)}$ est ordonnancé sur π avec l'algorithme* RM.

PREUVE : Supposons que les $(\ell - 1)$ premiers travaux de τ_k ont respecté leurs échéances avec l'algorithme RM ; nous allons prouver que le ℓ^{e} travail de τ_k va également respecter son échéance. La propriété sera alors vraie par induction sur ℓ en commençant par le cas trivial $\ell = 1$.

Le ℓ^{e} travail de τ_k arrive à l'instant $(\ell - 1)T_k$, a une échéance à l'instant ℓT_k et nécessite C_k unité d'exécution. À partir de l'inégalité 2.10 nous obtenons

$$W(\text{RM}, \pi, \tau^{(k)}, (\ell - 1)T_k) \geq (\ell - 1)T_k \left(\sum_{j=1}^{k} U(\tau_j) \right). \tag{2.11}$$

Aussi, au moins $(\ell - 1) \cdot T_k \cdot (\sum_{j=1}^{k-1} U(\tau_j))$ unités de cette exécution par l'algorithme RM étaient attribuées à $\tau_1, \tau_2, \ldots, \tau_{k-1}$ — cela provient du fait qu'il y a exactement $(\ell - 1)T_k U(\tau_k)$ unités du travail de τ_k qui ont été générées avant l'instant $(\ell - 1)T_k$ le reste de ce travail exécuté par l'algorithme RM doit par conséquent être généré par $\tau_1, \tau_2, \ldots, \tau_{k-1}$.

La demande d'exécution cumulée et générée par les travaux des tâches $\tau_1, \ldots, \tau_{k-1}$ qui arrive avant l'échéance du ℓ^{e} travail de τ_k est bornée par

$$\begin{aligned}
&\sum_{j=1}^{k-1} \left\lceil \frac{\ell T_k}{T_j} \right\rceil C_j \\
&< \sum_{j=1}^{k-1} \left(\frac{\ell T_k}{T_j} + 1 \right) C_j \\
&= \ell T_k \sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} C_j.
\end{aligned}$$

Comme nous l'avons examiné auparavant (dans la discussion qui suit l'inégalité 2.11) au moins $(\ell - 1) \cdot T_k \cdot \sum_{j=1}^{k-1} U(\tau_j)$ de cette demande est réalisée avant l'instant $(\ell - 1)T_k$, dès lors, au plus

$$\left(T_k \sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} C_j \right) \tag{2.12}$$

reste à exécuter *après* l'instant $(\ell - 1)T_k$.

La quantité de capacité processeur inutilisée par $\tau_1, \ldots, \tau_{k-1}$ dans l'intervalle $[(\ell - 1)T_k, \ell T_k)$ est dès lors plus grande que

$$S(\pi) \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} C_j \right). \tag{2.13}$$

Observons que *l'entièreté de cette capacité n'est pas disponible* pour le ℓ^{e} travail de τ_k ; en particulier, à chaque instant où plusieurs processeurs sont disponibles simultanément, τ_k peut s'exécuter seulement sur l'un de ces processeurs et la capacité processeur supplémentaire est perdue. Donc nous devons déterminer la partie de la capacité processeur identifiée dans l'expression (2.13) ci-dessus qui peut être garantie pour τ_k. Pour déterminer cela, observons que puisque l'algorithm RM est conservatif, si plusieurs processeurs sont simultanément disponibles alors τ_k s'exécute sur le *plus rapide* processeur disponible. Supposons que τ_k s'exécute sur le j^{e} plus rapide processeur à un instant (pour un j, $1 \leq j \leq m(\pi)$), alors les $(j+1)^{\text{e}}$, $(j+2)^{\text{e}}$, …, $m(\pi)^{\text{e}}$ processeurs plus rapides pourront être oisifs à cet instant, en revanche les processeurs plus rapides que le j^{e} plus rapide sont occupés et ne contribuent pas à la capacité processeur supplémentaire perdue. Dès lors, la fraction de cette capacité inutilisée à cet instant est au moins $(s_j(\pi)/[s_j(\pi) + s_{j+1}(\pi) + \cdots + s_{m(\pi)}(\pi)])$. Par définition du paramètre $\mu(\pi)$, ceci est $\geq \frac{1}{\mu(\pi)}$. Ainsi, une borne inférieure pour la quantité disponible pour l'exécution du ℓ^{e} travail de τ_k dans l'intervalle $[(\ell - 1)T_k, \ell T_k)$ est donnée par

$$\frac{1}{\mu(\pi)} \cdot \left[S(\pi) \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} C_j \right) \right].$$

Afin que le ℓ^{e} travail de τ_k respecte son échéance, il suffit que cette quantité soit au moins aussi grande que son temps d'exécution, c.-à-d.,

$$\frac{1}{\mu(\pi)} \cdot \left[S(\pi) \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} C_j \right) \right] \geq C_k$$

ce qui est vrai par définition de $U_{\max}(\tau)$:

$$U_{\max}(\tau) \geq U(\tau_k)$$

$$\equiv \frac{1}{\mu(\pi)} \cdot [U_{\max}(\tau) \cdot \mu(\pi)] \geq U(\tau_k)$$

(par la condition 2.8, et le fait que $\sum_{j=1}^{k-1} U(\tau_j) \leq U(\tau)$*)* $\Rightarrow$

$$\frac{1}{\mu(\pi)} \cdot \left[S(\pi) - 2 \cdot \sum_{j=1}^{k-1} U(\tau_j) \right] \geq U(\tau_k)$$

(puisque $T_k \geq T_j$ *for* $j < k$*)* $\Rightarrow$

$$\frac{1}{\mu(\pi)} \cdot \left[S(\pi) - \left(\sum_{j=1}^{k-1} U(\tau_j) + \sum_{j=1}^{k-1} \frac{C_j}{T_k} \right) \right] \geq U(\tau_k)$$

et par conséquent prouve la propriété. □

THÉORÈME 2.3.– *Soit τ un ensemble de tâches périodiques et π une plate-forme multiprocesseur uniforme,*

$$S(\pi) \geq 2 \cdot U(\tau) + \mu(\pi) \cdot U_{\max}(\tau)$$

est une condition suffisante pour assurer l'ordonnançabilité avec RM *sur la plate-forme π.*

PREUVE : Lorsque τ et π satisfont la condition du théorème (qui est exactement la condition 2.8), nous avons montré (avec le lemme 2.2 ci-dessus) que l'algorithme RM ordonnance $\tau^{(k)}$ dans la mesure où tous les travaux de la tâche la moins prioritaire τ_k respectent leur échéance. L'exactitude du théorème 2.3 suit immédiatement par induction sur k. □

Nous pouvons à présent appliquer le théorème 2.3 aux plates-formes multiprocesseurs *identiques* pour obtenir :

COROLLAIRE.– *Tout ensemble de tâches périodiques dont le facteur d'utilisation de chaque tâche n'excède pas un tiers et dont l'utilisation totale n'excède pas $m/3$ est ordonnançable sous l'égide de l'algorithme* RM *avec m processeurs identiques de capacité 1.*

PREUVE : Soit τ un ensemble de tâches périodiques avec les antécédents du corollaire : $U(\tau) \leq m/3$ et $U_{\max}(\tau) \leq 1/3$. Soit π une plate-forme de m processeurs

identiques de capacité 1, par définition,

$$\mu(\pi) \stackrel{\text{def}}{=} \max_{i=1}^{m(\pi)} \left\{ \frac{\sum_{j=i}^{m(\pi)} s_j(\pi)}{s_i(\pi)} \right\} \;=\; \left\{ \frac{m}{1} \right\} \;=\; m.$$

Par le théorème 2.3, τ est ordonnançable avec RM sur π si

$$\begin{aligned} & S(\pi) \geq 2 \cdot U(\tau) + \mu(\pi) \cdot U_{\max}(\tau) \\ \equiv \quad & \left(m \geq 2 \cdot \frac{m}{3} + m \cdot \frac{1}{3} \right) \\ \equiv \quad & (m \geq m), \end{aligned}$$

ce qui est vrai et prouve la propriété. □

2.6.4. *Les algorithmes* PFAIR

Dans cette section nous allons présenter brièvement la famille d'algorithmes PFAIR. Nous nous limiterons à la définition de cette famille et de ses propriétés importantes, ce sujet pourrait faire l'objet d'un chapitre à part entier, le lecteur pourra se référer à [LEU 04], chapitre 31 pour plus de détails. Dans [BAR 96], BARUAH *et al.* établissent les fondements théoriques pour les algorithmes PFAIR, en particulier les auteurs définissent la notion de *« proportionate fairness »* (que nous pouvons traduire par *équité proportionnée*). Les algorithmes PFAIR diffèrent fortement des algorithmes classiques dans la mesure où il est explicitement requis d'exécuter les tâches à un taux régulier (quasi constant). Dans les ordonnancements de tâches périodiques, chaque tâche τ_i s'exécute approximativement avec un taux de $U(\tau_i)$, si du moins l'on considère de grands intervalles. Cependant, sur de petits intervalles, le taux d'exécution de τ_i peut varier de manière importante. Dans les ordonnancements PFAIR, chaque tâche est exécutée quasiment à un taux constant en divisant la tâche en une série de sous-tâches. Afin d'assurer le taux d'exécution de la tâche, les sous-tâches doivent s'exécuter dans des intervalles de tailles identiques appelés des *fenêtres*. Différentes sous-tâches d'une tâche peuvent s'exécuter sur des processeurs différents (la migration est donc autorisée) mais pas simultanément (c.-à-d. que le parallélisme au sein d'une tâche est interdit). Dans le cas particulier de l'ordonnancement multiprocesseur identique et pour des tâches (strictement) périodiques à échéance sur requête, PFAIR est une stratégie optimale.

Nous allons à présent caractériser de manière plus précise les ordonnancements PFAIR, nous considérons l'ordonnancement de tâches périodiques, à échéance sur requête et à départ simultané. Nous supposons que l'allocation des processeurs est discrète.

Formalisons à présent la notion d'ordonnancement, il s'agit d'une fonction $\mathcal{S}$: $\tau \times \mathbb{Z} \mapsto \{0, 1\}$, où τ est l'ensemble de tâches périodiques. Lorsque $\mathcal{S}(\tau_i, t) = 1$ cela signifie que τ_i est ordonnancé dans l'intervalle $[t, t + 1)$.

Dans un ordonnancement (idéal) parfaitement « *fair* », chaque tâche τ_i doit recevoir exactement $U(\tau_i) \cdot t$ unités de processeur dans l'intervalle $[0, t)$ (ce qui implique que toutes les échéances sont satisfaites). Cependant, un tel ordonnancement n'est pas possible dans le cas discret. En revanche, PFAIR tente à chaque instant de répliquer le plus fidèlement possible cet ordonnancement idéal. La différence entre l'ordonnancement idéal et l'ordonnancement construit est formalisée par la notion de *retard* (*lag* dans la littérature d'origine). Plus formellement, le retard d'une tâche τ_i à l'instant t, dénoté par $\mathrm{retard}(\tau_i, t)$ est défini de la manière suivante :

$$\mathrm{retard}(\tau_i, t) \stackrel{\mathrm{def}}{=} U(\tau_i) \cdot t - \sum_{\ell=0}^{t-1} \mathcal{S}(\tau_i, \ell). \tag{2.14}$$

Un ordonnancement est dit PFAIR si et seulement si

$$-1 < \mathrm{retard}(\tau_i, t) < 1 \qquad \forall \tau_i \in \tau, t \in \mathbb{Z}. \tag{2.15}$$

De manière informelle, l'équation 2.15 demande que l'erreur d'allocation pour chaque tâche soit inférieure à une unité et ceci à chaque instant, ce qui implique que τ_i doit avoir reçu soit $\lfloor U(\tau_i) \cdot t \rfloor$ soit $\lceil U(\tau_i) \cdot t \rceil$.

Il est aisé de voir que la contrainte de « *Pfairness* » implique nécessairement le respect des échéances. Une tâche périodique τ_i doit en effet recevoir C_i unités de processeur dans chaque intervalle $[\ell \cdot T_i, (\ell + 1) \cdot T_i)$, avec $\ell \geq 0$. À l'instant $t = \ell \cdot T_i$, $U(\tau_i) \cdot t = (C_i/T_i) \cdot \ell \cdot T_i = \ell \cdot C_i$, qui est un entier. Par l'équation 2.15, à l'instant $t = \ell \cdot T_i$ l'allocation de la tâche dans un ordonnancement PFAIR correspond à l'ordonnancement idéal. Puisque toutes les échéances sont satisfaites dans l'ordonnancement idéal elles le sont aussi dans l'ordonnancement PFAIR.

Enfin, dans [BAR 96], BARUAH *et al.* ont montré la condition de faisabilité suivante :

THÉORÈME 2.4.– *Soit τ un système périodique à échéance sur requête et départ simultané, il existe un ordonnancement* PFAIR *sur m processeurs identiques de capacité 1 si et seulement si*

$$U(\tau) \leq m. \tag{2.16}$$

En ce qui concerne l'ordonnançabilité, à ce jour, trois algorithmes d'ordonnancement PFAIR optimaux ont été proposés : PF [BAR 96], PD [BAR 95] et PD^2 [AND 00].

Ces algorithmes donnent des priorités aux sous-tâches en utilisant une stratégie proche d'EDF, ils différent dans la manière de résoudre les cas d'égalités de priorité, c.-à-d. les cas d'ambiguïtés. Remarquons que l'existence de ces algorithmes optimaux ne contredit pas le théorème 2.1, nous considérons l'ordonnancement de tâches strictement périodiques, l'arrivée des travaux futurs est connue, d'une certaine manière nous connaissons le futur pour ces systèmes.

2.7. Techniques par partitionnement

Dans cette section, nous allons présenter brièvement, des techniques plus classiques et moins récentes qui de surcroît font l'hypothèse que les tâches sont *statiques*, à échéance sur requête et sporadiques. Lorsque l'ensemble de tâches est connu lors de la conception du système, une méthode habituelle consiste à assigner *statiquement* les tâches aux processeurs, en d'autres termes à réaliser un *partitionnement*. Lors de l'exécution du système, chaque tâche s'exécute uniquement sur *le* processeur assigné à cette dernière (c.-à-d. que les migrations sont interdites) avec une stratégie *locale* (monoprocesseur). Généralement ce partitionnement est réalisé avec une approche *hors ligne* sur base des caractéristiques statiques des tâches (du facteur d'utilisation en particulier).

Dans sa forme la plus simple, le problème de partitionnement peut être formulé de la manière suivante : étant donné n utilisations de tâches périodiques, à échéance sur requêtes, nous devons former m sous-ensembles (disjoints) tel que l'utilisation cumulée de chaque sous-ensemble soit suffisamment petite pour pouvoir ordonnancer ce sous-ensemble sur un seul processeur. Par exemple, si la stratégie locale d'ordonnancement de chaque processeur est EDF, nous devons former des sous-ensembles dont l'utilisation cumulée n'excède pas l'unité (étant donné l'optimalité d'EDF, voir section 1.2.3). Trouver un partitionnement optimal est équivalent à un problème d'emballage optimal (*bin-packing* dans la littérature d'origine), c.-à-d. de déterminer la manière de placer le plus grand nombre d'objets possible dans le plus petit nombre de contenants. Les objets étant les tâches et les contenants étant les processeurs. Malheureusement ce problème est NP-difficile au sens fort. Des heuristiques de complexité polynomiale sont proposées pour résoudre ce type de problème. *First fit*, *best fit* sont de tels exemples. Avec *first fit*, chaque tâche est assignée au premier processeur (c.-à-d., au processeur d'indice le plus petit) qui peut l'accepter. Au contraire avec *best fit* chaque tâche est assignée à un processeur (i) qui peut accepter la tâche et (ii) avec la capacité disponible la plus petite.

On distingue deux manières de mesurer les performances des heuristiques, soit l'heuristique détermine elle-même le nombre de processeurs requis et le partitionnement, soit le nombre de processeurs est fixé au départ. Dans le premier cas de figure, on s'intéresse au rapport entre entre le nombre de processeurs requis par l'heuristique

et le nombre m_o de processeurs requis par le partitionnement optimal, plus précisément on s'intéresse à ce rapport à la limite pour $m_o \to \infty$. Dans [COF 81], COFFMAN *et al.* ont montré que ce rapport n'excède pas 1.7 pour l'heuristique *first fit*.

Dans le second cas, où le nombre de processeurs est fixé, on détermine le plus grand facteur d'utilisation de l'ensemble de tâches pour lequel on est assuré que l'heuristique déterminera un partitionnement disposant de m processeurs. Notons que le pire cas pour cette utilisation, que ce soit pour une technique heuristique ou même pour un partitionnement optimal, est seulement de $(m+1)/2$, même si un algorithme optimal monoprocesseur comme EDF est utilisé. En d'autres termes, il existe des systèmes périodiques avec une utilisation à peine supérieure à $(m+1)/2$ qui ne peuvent pas être ordonnancés par un partitionnement. Pour nous convaincre de ce fait, considérons un ensemble de $m+1$ tâches, chacune avec un temps d'exécution de $1+\epsilon$ ($\epsilon > 0$) et une période de 2 ; cet ensemble ne peut pas être partitionné sur m processeurs puisque deux tâches ne peuvent jamais résider sur le même processeur.

Pour des stratégies locales à priorités fixes au niveau des tâches, comme RM, ce facteur d'utilisation maximal est même moindre. En effet, pour l'heuristique *first fit* et la stratégie RM, cette borne, que nous dénotons par $U_{\text{RM,FF}}$ a été caractérisée par OH et BAKER de la manière suivante [OH 98] :

THÉORÈME 2.5.–

$$(\sqrt{2}-1) \times m \leq U_{\text{RM,FF}} \leq (m+1)/(1+2^{\frac{1}{m+1}}).$$

2.8. Conclusion

Dans ce chapitre nous avons considéré l'ordonnancement d'applications temps réel sur des architectures multiprocesseurs. Nous avons présenté des techniques classiques et déjà bien établies comme PFAIR et les techniques par partitionnement. Nous avons également abordé des techniques plus récentes qui ont permis de grandes avancées dans la problématique de l'ordonnancement multiprocesseur. En particulier, nous avons insisté sur les techniques par augmentation de ressources que nous croyons prometteuses pour les développements ultérieurs.

L'étude nous a aussi permis de montrer, s'il en était encore nécessaire, que dans cette discipline il faut être extrêmement prudent, éviter de prendre ses désirs pour une réalité et se méfier de ses intuitions. En particulier, nous avons montré que le problème d'ordonnancement multiprocesseur n'est pas une triviale extension du problème d'ordonnancement monoprocesseur, comme l'illustre par exemple le phénomène d'anomalie d'ordonnancement ou d'inexistence d'algorithmes en-ligne et optimaux. Nous

avons aussi montré que les techniques et solutions sont de natures très différentes que celles déjà bien établies en ordonnancement monoprocesseur. Notons aussi que nous sommes loin d'avoir aujourd'hui une bonne connaissance de l'ordonnancement temps réel multiprocesseur, nous sommes au balbutiement de cette nouvelle discipline. Il nous semble que la communauté scientifique à un défi à relever en la matière. Il y a lieu d'encourager les chercheurs à faire preuve d'imagination et de perspicacité dans des travaux futurs afin de relever ce défi.

2.9. Bibliographie

[AND 00] ANDERSON J., SRINIVASAN A., « Early-release fair scheduling », *12th Euromicro Conference on Real-Time Systems*, p. 35–43, 2000.

[BAR 95] BARUAH S., GEHRKE J., PLAXTON C. G., « Fast scheduling of periodic tasks on multiple resources », *9th International Parallel Processing*, p. 280–288, 1995.

[BAR 96] BARUAH S., COHEN N., PLAXTON C. G., VARVEL D., « Proportionate progress : A notion of fairness in resource allocation », *Algorithmica*, vol. 15, p. 600–625, 1996.

[BAR 03] BARUAH S., GOOSSENS J., « Rate-Monotonic Scheduling on Uniform Multiprocessors », *IEEE Transactions on Computers*, vol. 52, n°7, p. 966–970, 2003.

[COF 81] COFFMAN E. G., GAREY M. R., JOHNSON D. S., *Approximation algorithms for bin-packing—a survey*, Analysis and Design of Algorithms in Combinatorial Optimization, G. Ausiello and M. Lucertini, 1981.

[FUN 01] FUNK S., GOOSSENS J., BARUAH S., « On-line Scheduling On Uniform Multiprocessors », *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, UK, IEEE Computer Society Press, p. 183–192, December 2001, ISBN 0-7695-1420-0.

[HON 88] HONG K. S., LEUNG J. Y.-T., « On-Line Scheduling of Real-Time Tasks », *IEEE Computer Society*, 1988.

[LEU 82] LEUNG J. Y.-T., WHITEHEAD J., « On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks », *Performance Evaluation*, vol. 2, p. 237-250, 1982.

[LEU 04] LEUNG J., Ed., *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*, Chapman Hall/CRC Press, 2004.

[LIU 73] LIU C. L., LAYLAND J. W., « Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment », *Journal of the Association for Computing Machinery*, vol. 20, n°1, p. 46–61, January 1973.

[MOK 78] MOK A., DERTOUZOS M., « Multiprocessor scheduling in a hard real-time environment », *Proceedings of the Seventh Texas Conference on Computing Systems*, 1978.

[OH 98] OH D., BAKER T., « Utilization bound for n-processor rate monotone scheduling with static processor assignment », *Real-Time Systems*, vol. 15, n°2, p. 183–192, 1998.

[PHI 97] PHILLIPS C. A., STEIN C., TORNG E., WEIN J., « Optimal Time-Critical Scheduling via Resource Augmentation », *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, p. 140–149, mai 1997.