Ordonnancement temps réel multiprocesseur

http://goo.gl/etM4Wq

Joël Goossens Pascal Richard

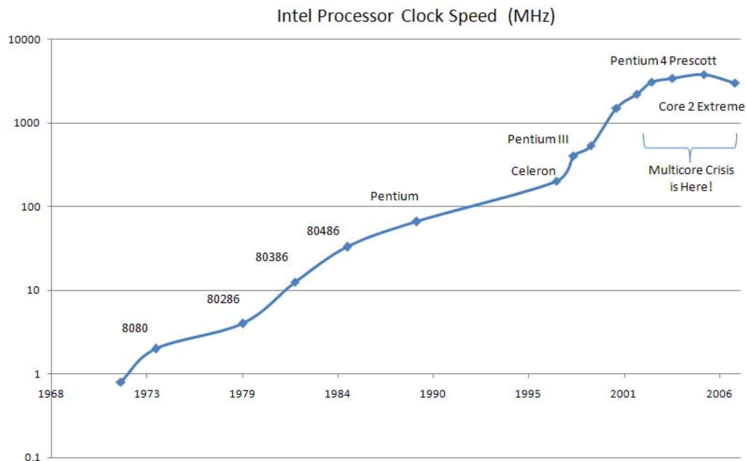
Université Libre de Bruxelles, LIAS/ENSMA et Université de Poitiers

26 août 2013

Objectifs de l'exposé

- Présenter les résultats importants en théorie l'ordonnancement multiprocesseur
- Vous convaincre que ce n'est pas une simple généralisation du cas monoprocesseur
- Montrer les propriétés/problèmes **intrinsèques** au multiprocesseur
- Identifier des questions importantes laissées ouvertes

Motivations I



Motivations II

- Pour bénéficier d'une grande puissance de calcul les architectures parallèles sont incontournables pour au moins deux raisons :
  - pour des raisons physiques : nous arrivons à la limite de la matière
  - pour des raisons économiques : la production de CI à haute densité est extrêmement coûteuse

Modélisation des applications I

Les applications seront modélisées par un ensemble de travaux ou par un ensemble de tâches.

Definition 1 (Travail et instance)

Un **travail** j sera caractérisé par le tuple (a, e, d). Un instant d'arrivée a, un temps d'exécution e et une échéance 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 : $J = \{j_1, j_2, \ldots\}$.

Classiquement dans les applications temps réel, les calculs/opérations sont de nature **récurrente**, on parle alors de tâches périodiques et sporadiques.

Modélisation des applications II

Definition 2 (Tâches périodiques et sporadiques)

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

- la **date d'arrivée** (*offset*) 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 travaux 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 echéance à l'instant $t + D_i$)
- une **période** T_i dénotant la durée qui sépare deux arrivées successives de travaux pour τ_i

$\tau_i = (O_i, T_i, D_i, C_i)$ génère un nombre infini de travaux (ou de **requêtes**), chacun avec le même temps d'exécution, à chaque instant $(O_i + k \cdot T_i)$

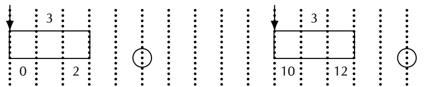
Modélisation des applications III

pour tout entier $k \geqslant 0$, le travail généré à l'instant $(O_i + k \cdot T_i)$ a une échéance à l'instant $(O_i + k \cdot T_i + D_i)$.

Modélisation des applications IV

Exemple de tâche périodique et deux de ses travaux

$$\tau_1 = (O_1 = 0, T_1 = 10, C_1 = 3, D_1 = 5)$$



$$U(\tau_1) = 3/10$$

Modélisation des applications V

Les tâches **sporadiques** sont similaires aux tâches périodiques, avec la différence que T_i corresponds à la durée **minimum**, plutôt que **exacte**, qui sépare deux arrivées successives de travaux pour τ_i. Une tâche sporadique est caractérisée par trois paramètres : (T_i, D_i, C_i).
Un **système périodique** τ est constitué d'une collection **finie** de tâches périodiques (ou sporadiques) : $\tau = \{\tau_1, \ldots, \tau_n\}$.

Definition 3 (Utilisation)

L'**utilisation** $U(\tau_i)$ d'une tâche est le rapport entre son temps d'exécution et sa période : $U(\tau_i) \stackrel{\text{def}}{=} C_i/T_i$. L'utilisation d'un système périodique (ou sporadique) est la somme des utilisations des tâches : $U(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U(\tau_i)$

Cas particuliers de systèmes périodiques

D'un point de vue pratique et théorique, il est intéressant de distinguer les cas suivants :

échéance sur requête
: où l'échéance de chaque tâche coïncide avec la période (c.-à-d., $D_i = T_i \; \forall i$, chaque instance doit se terminer avant l'occurrence de l'instance suivante pour la même tâche)

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

échéance arbitraire
: s'il n'y a pas de contraintes 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 $O_i = 0 \; \forall i$

Notations et hypothèses

- Nous considérons l'ordonnancement de n tâches périodiques/sporadiques : $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$
- Sur une plate-forme **parallèle** composée de m processeurs identiques : $\pi_1, \pi_2, \ldots, \pi_m$
- Parallélisme limité : l'ordonnancement sur une architecture multiprocesseur devra respecter les contraintes suivantes :
  - un processeur exécute au plus une tâche à chaque instant ;
  - une tâche s'exécute sur au plus un processeur à chaque instant.

La plate-forme est **parallèle** mais les tâches (code) sont **séquentielles**

Les priorités

- Classe FTP : algorithmes à priorité **fixe** au niveau des tâches : qui assignent des priorités aux tâches (p. ex. RM).
- Classe FJP : algorithmes à priorité fixe au niveau des travaux : qui assignent des priorités **fixes** aux travaux (p. ex. EDF).
- Classe DP : algorithmes à priorité dynamique (au niveau des travaux) : qui assignent des priorités **dynamiques** aux travaux. Dans ce cas la priorité d'un même travail peut évoluer au cours du temps.

En-ligne et hors-ligne

- L'ordonnancement peut être construit **hors-ligne**, durant la **conception** du système
- L'ordonnancement peut être construit **en-ligne**, durant l'**exécution du système**

Préemptions et migrations I

- On distingue en général trois classes pour l'ordonnancement multiprocesseur :
  - Ordonnancement **sans migration** ou **partitioné**. Il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints : $\tau^1, \tau^2, \ldots, \tau^m$ et d'ordonnancer ensuite l'ensemble τ^i sur le processeur π_i avec une stratégie d'ordonnancement **locale monoprocesseur**. Les tâches ne sont donc pas autorisées à **migrer** d'un processeur à l'autre.

Préemptions et migrations II

- Ordonnancement avec **migration restreinte** : les tâches sont autorisées à migrer, les travaux ne peuvent pas migrer.
- **Stratégie globale** ou à **migration totale**. Il s'agit d'appliquer globalement sur l'entièreté de la plate-forme multiprocesseur une stratégie d'ordonnancement et d'attribuer à chaque instant les m processeurs aux m tâches/travaux les plus prioritaires.

Un exemple de partition

	C	T	D
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

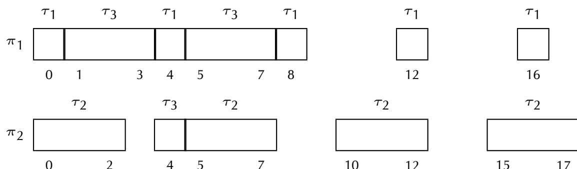
$$U(\tau) = 1.2$$

- Nous pouvons ordonnancer τ_1 et τ_2 sur un seul processeur
- et τ_3 sur un second processeur.

Un exemple d'ordonnancement pour « global DM »

	C	T	D
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

$$U(\tau) = 1.2$$

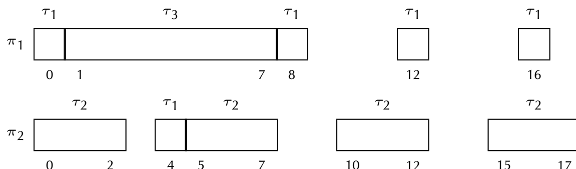


Un exemple d'ordonnancement avec migration restreinte

	C	T	D
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

$U(\tau) = 1.2$

$\tau_1 \succ \tau_2 \succ \tau_3$



Hypothèse

Dans la suite de l'exposé (sauf mention contraire) nous supposons ordonnancer des tâches à **échéance sur requête** ($D_i = T_i$).

$\langle x, y \rangle$ où x pour la classe ordonnanceur et y classe migration

Notation	Sémantique
$\langle w, x \rangle = \langle y, z \rangle$	Equivalence
$\langle w, x \rangle \otimes \langle y, z \rangle$	Incomparables
$\langle w, x \rangle \subset \langle y, z \rangle$	$\langle y, z \rangle$ Domine

Classes d'algorithmes			Partitionné (P)			Migration restreinte (MR)			Migration totale (MT)		
			FTP $\langle 1, 1 \rangle$	FJP $\langle 2, 1 \rangle$	DP $\langle 3, 1 \rangle$	FTP $\langle 1, 2 \rangle$	FJP $\langle 2, 2 \rangle$	DP $\langle 3, 2 \rangle$	FTP $\langle 1, 3 \rangle$	FJP $\langle 2, 3 \rangle$	DP $\langle 3, 3 \rangle$
(P)	FTP	$\langle 1, 1 \rangle$	$=$	$\subset$	$\subset$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
	FJP	$\langle 2, 1 \rangle$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
	DP	$\langle 3, 1 \rangle$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
(MR)	FTP	$\langle 1, 2 \rangle$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$	$\otimes$	$\times$	$\subset$
	FJP	$\langle 2, 2 \rangle$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$=$	$\subseteq$	$\otimes$	$\times$	$\subset$
	DP	$\langle 3, 2 \rangle$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$\supseteq$	$=$	$\otimes$	$\otimes$	$\subset$
(MT)	FTP	$\langle 1, 3 \rangle$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$
	FJP	$\langle 2, 3 \rangle$	$\otimes$	$\otimes$	$\otimes$	$\times$	$\times$	$\otimes$	$\supset$	$=$	$\subset$
	DP	$\langle 3, 3 \rangle$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$=$

$\langle 2, 1\rangle \otimes \langle 2, 3\rangle$ (partitionné et global incomparables) I

- L'ordonnancement partitionné (qui interdit la migration) n'est pas un cas particulier de l'ordonnancement global (qui permet mais n'oblige pas la migration)
- Il y a des systèmes ordonnançables avec des stratégies globales mais pour lesquels aucun partitionnement n'existe.
- Moins intuitif probablement, il y a des systèmes partitionnables qui ne peuvent pas être ordonnancés avec des priorités fixes.

Leung a montré cela en 1982 pour des tâches périodiques et des priorités fixes au niveau des tâches, plus récemment Baruah [Baruah, 2007] a montré cela pour des priorités fixes au niveau des travaux (et des tâches sporadiques).

$\langle 2, 1\rangle \otimes \langle 2, 3\rangle$ (partitionné et global incomparables) II

Nous commençons par prouver $\langle 2, 1\rangle \not\subseteq \langle 2, 3\rangle$

Voici un exemple pour $m = 2$ et

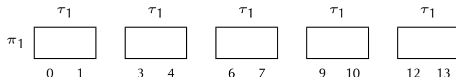
	C	T	D
τ_1	2	3	2
τ_2	3	4	3
τ_3	5	12	12

$U = 1.8333$

- Le partitionnement échoue car $U(\tau_i) + U(\tau_j) > 1 \quad \forall i \neq j$.
- Nous pouvons ordonnancer avec un algorithme à priorité fixe au niveau des travaux en donnant la priorité la plus faible aux travaux de τ_3.

$\langle 2, 1\rangle \otimes \langle 2, 3\rangle$ (partitionné et global incomparables) III

- Vérifions l'ordonnancabilité de τ_3, pour chaque suite de 12 slots, τ_1 peut s'exécuter au plus pendant 8 slots :



  - Si τ_1 s'exécute moins que pendant 8 slots, τ_3 est ordonnançable.
  - Sinon les travaux de τ_1 doivent être séparés de 3 unités. Étant donné les paramètres de τ_2 il y a nécessairement un slot supplémentaire pour l'exécution de τ_3.

$\langle 2,1\rangle \otimes \langle 2,3\rangle$ (partitionné et global incomparables) IV

Nous terminons par prouver que $\langle 2,3\rangle \not\subseteq \langle 2,1\rangle$

Voici un autre exemple pour $m = 2$ et

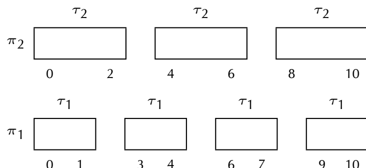
	C	T	D
τ_1	2	3	2
τ_2	3	4	3
τ_3	4	12	12
τ_4	3	12	12

$U = 2$

- Un partitionnement satisfait : $(\tau_1 + \tau_3), (\tau_2 + \tau_4)$ avec EDF sur chaque processeur.
- Pour se convaincre qu'il n'y ai pas d'assignation de priorité fixe au niveau des travaux faisable, nous considérons le cas à départ simultané et l'intervalle $[0, 12)$.

$\langle 2, 1\rangle \otimes \langle 2, 3\rangle$ (partitionné et global incomparables) V

- Pour la faisabilité nous devons servir **d'abord** tous les travaux de τ_1 et τ_2 $(C = D)$
- Peu importe les priorités des travaux moins prioritaires de τ_3 et τ_4 le moins prioritaire va rater son échéance et un processeur sera oisif dans l'intervalle $[11, 12)$.



Borne d'utilisation U_B

À la fois pour les tests d'ordonnançabilité et afin de caractériser les performances des algorithmes d'ordonnancement on utilise principalement deux mesures.

Définition 1 (Borne d'utilisation U_B)

$U(\tau) \leqslant U_B \;\Rightarrow\; \tau$ est ordonnançable

U_B dépend naturellement de l'ordonnanceur, plus U_B est grand (et proche de m) plus l'ordonnanceur est bon.

- Seule la classe $\langle 3, 3 \rangle$ permet d'avoir $U_B = m$, dans tous les autres cas $U_B \leqslant (m+1)/2$.

Uniquement sensé pour des systèmes à échéance sur requête

$U_B \leqslant (m+1)/2$

Example 1 ($U_B \leqslant (m+1)/2$)

$(m+1)$ tâches identiques avec $C_i = 1 + \epsilon$ et $T_i = 2$ (ϵ infinitésimal). Comme toutes les périodes sont identiques cela revient à étudier un ordonnanceur partitionné. Il y a alors nécessairement deux tâches sur le même processeur qui ne sont pas ordonnançables (processeur trop chargé). À la limite nous obtenons donc $U_B = (m+1)/2$.

Dans le pire des cas on n'exploite que **50%** de la capacité de la plate-forme !

Augmentation de ressource I

Définition 2 (Optimalité)

Un ordonnanceur optimal ordonnance tous les systèmes ordonnançables

L'optimalité des algorithmes en-ligne est rare voire impossible.

- Comment comparer les performances d'algorithmes en-ligne sous-optimaux ?
- L'augmentation de ressource quantifie, dans le pire cas, quelle augmentation de ressource doit-on fournir à l'algorithme en-ligne pour ordonnancer ce qu'arrive à ordonnancer un algorithme optimal hors-ligne (clairvoyant).
- Le **facteur d'accélération** consiste à octroyer des processeurs plus rapides à l'algorithmes en-ligne.

Augmentation de ressource II

Théorème 1 (Facteur d'accélération de $(2 - \frac{1}{m})$ [Phillips et al., 1997])

Si un ensemble de travaux en faisable sur une plate-forme composée de m processeurs de vitesse unitaire, alors le même ensemble de travaux est ordonnançable par EDF *sur une une plate-forme composée de m processeurs de vitesse* $(2 - \frac{1}{m})$

- Plus le facteur d'accélération est petit plus l'ordonnanceur est proche de l'optimal

Techniques par partitionnement $\Rightarrow$ Bin Packing

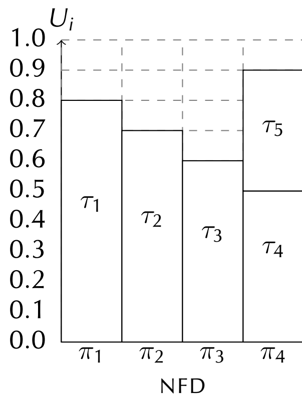
- Le problème de partitionnement est en fait un problème bien connu sous le nom de **Bin Packing** dans lequel il faut placer dans m boîtes de tailles identiques, k objets de tailles différentes.
- Dans notre cas, les objets sont les tâches, leurs tailles sont leurs utilisations $U(\tau_i)$, la taille de chaque boîte est l'utilisation maximale que l'on peut atteindre : $\ln 2$ pour RM, 1 pour EDF.
- Ce problème est NP-Difficile, on utilise dès lors des heuristiques pour **approcher** la solution optimale.

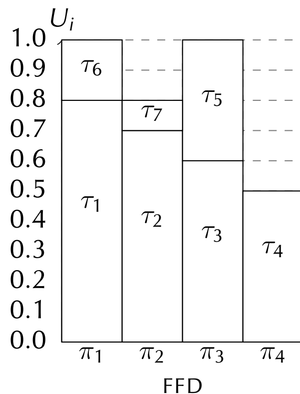
Heuristiques de partitionnement I

Algorithmes gloutons (qui ne reviennent jamais sur un choix précédent) qui reposent sur deux décisions :

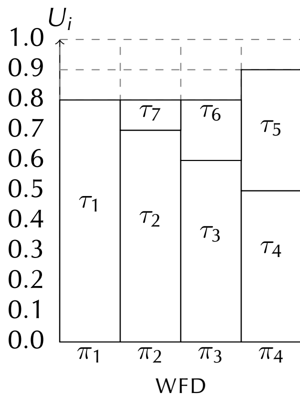
- **Tri** préalable qui défini dans quel ordre on va placer les objets
  - Souvent on trie les tâches de manière décroissante sur le facteur d'utilisation
- Heuristique pour décider où placer l'objet courant, les règles les plus populaires sont
  - *First-Fit*
  - *Best-Fit*
  - *Worst-Fit*
  - *Next-Fit*

Heuristiques de partitionnement II





Heuristiques de partitionnement III



Ratio asymptotique du nombre de processeurs

Définition 3

Pour l'algorithme A le ratio asymptotique du nombre de processeurs est le plus petit nombre réel $\rho(A)$ avec

$$\frac{A(I)}{\text{OPT}(I)} \leqslant \rho(A)$$

pour toutes les instances I du problème. $\text{OPT}(I)$ et $A(I)$ dénote respectivement le nombre de processeurs nécessaire par un algorithme optimal et l'algorithme A.

- Par exemple pour *First-Fit-Decreasing* et EDF ce ratio est égal à 3/2 (le minimum possible)
- Pour Rate Monotonic Next-Fit ce ratio est égal à 2.67

Bornes d'utilisation I

Définition 4

Un algorithme d'allocation est dit **raisonnable** s'il peut toujours allouer une tâche sur un processeur s'il en existe un pouvant accepter l'allocation.

Par exemple First-Fit et Best-Fit sont raisonnables.
Les bornes dépendent de l'algorithme d'allocation (et de l'ordonnanceur local, nous considérons EDF dans la suite)
D'abord deux nouvelles grandeurs

- $U_{\max}(\tau) = \max\{U(\tau_i) \mid i = 1, \ldots, n)\}$
- $\beta = \left\lfloor \frac{1}{U_{\max}(\tau)} \right\rfloor$ nombre maximun de tâches d'utilisation $U_{\max}(\tau)$

Bornes d'utilisation II

Résultats à propos des bornes d'utilisation :

- Chaque algorithme raisonnable possède une borne d'utilisation supérieure ou égale à $m - (m-1)U_{\max}(\tau)$
- Aucun algorithme (raisonnable ou pas) ne peut admettre une borne d'utilisation supérieure à $\frac{\beta m+1}{\beta+1}$.
- Les algorithmes *Worst-Fit Decreasing*, *First-Fit Decreasing* et *Best-Fit Decreasing* garantissent la borne supérieure $\frac{\beta m+1}{\beta+1}$.

Augmentation de ressource

Nous résumons ici les facteurs d'accélérations pour des algorithmes d'allocation polynomiaux.

- Chaque algorithme raisonnable avec les tâches triées de manière décroissante sur le facteur d'utilisation ont un facteur d'accélération de $\frac{4}{3} - \frac{1}{3m}$
- *Worst-Fit* et *Worst-Fit Increasing* ont un facteur d'accélération de $2 - \frac{2}{m}$
- les autres FF, FFI, BF et BFI ont un facteur d'accélération de $2 - \frac{2}{m+1}$

Optimalité impossible — Propriété

Le premier résultat important, mais malheureusement **négatif**, est l'absence d'algorithme **optimal en ligne**, mais d'abord une définition.

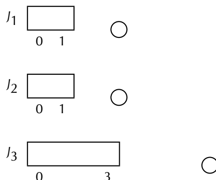
Définition 5 (Algorithme en ligne)

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

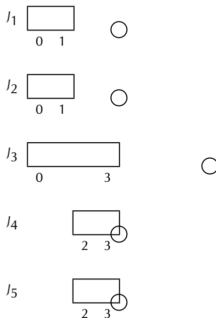
Théorème 2 ([Hong and Leung, 1988])

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.

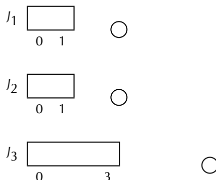
Optimalité impossible — Preuve

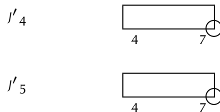


Optimalité impossible — Preuve

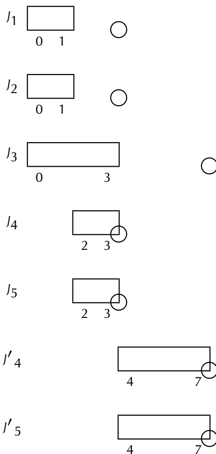


Optimalité impossible — Preuve





Optimalité impossible — Preuve



Optimalité impossible dans le cas sporadique I

- Le contre-exemple de Hong concerne l'ordonnancement de travaux, il ne peut pas être utilisé pour montrer que dans le cas particulier de travaux générés par des tâches **sporadiques** l'optimalité est impossible.
- Dans [Fisher et al., 2010] nous avons montré que le système sporadique suivant est faisable sur deux processeurs et l'ordonnançabilité nécessite la **clairvoyance** pour prendre une décision optimale.

Optimalité impossible dans le cas sporadique II

	C_i	D_i	T_i
τ_1	2	2	5
τ_2	1	1	5
τ_3	1	2	6
τ_4	2	4	100
τ_5	2	6	100
τ_6	4	8	100

- La difficulté fut de montrer la faisabilité d'un tel système (non verrons pourquoi par la suite).

Anomalies d'ordonnancement – Définition

Définition 6 (Anomalie)

Nous disons qu'un algorithme d'ordonnancement souffre d'**anomalies** si un changement qui est intuitivement positif dans un système ordonnançable peut le rendre non ordonnançable.

Définition 7 (Changement intuitivement positif)

C'est un changement qui diminue le facteur d'utilisation d'une tâche comme augmenter la période, diminuer un temps 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.

Anomalies d'ordonnancement – Exemple I

- Les algorithmes d'ordonnancement multiprocesseurs sont sujet à des anomalies.
- Par exemple, pour les algorithmes à priorité fixe au niveau des tâches, il existe des systèmes qui sont ordonnançables avec une assignation de priorité mais lorsque une **période** est augmentée et les priorités conservées ces systèmes ne sont plus ordonnançables !

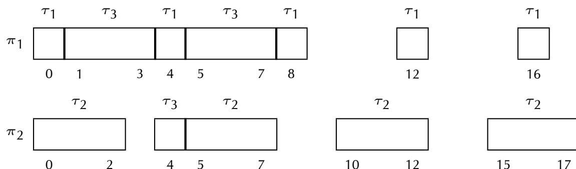
Anomalies d'ordonnancement – Exemple II

C'est le cas du système

	C	T	D
τ_1	1	4	2
τ_2	3	5	3
τ_3	7	20	8

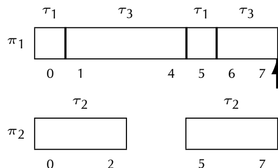
$$U(\tau) = 1.2$$
$$U_{\max}(\tau) = 0.6$$

Ce système est ordonnançable sur 2 processeurs pour le « global DM » $(\tau_1 \succ \tau_2 \succ \tau_3)$



Anomalies d'ordonnancement – Exemple III

En revanche, si l'on augmente la période de τ_1 de 4 à 5, et pour la même stratégie (global DM) le système résultant n'est **pas** ordonnançable : τ_3 rate son échéance à l'instant 8.



Conséquences des anomalies d'ordonnancement

- Pour vérifier l'ordonnancement de tâches **sporadiques** il n'est d'aucune utilité d'étudier l'ordonnançabilité de l'ensemble périodique (et à départ simultané) correspondant.
- En d'autres termes, le cas périodique ne correspond pas au **pire cas** en multiprocesseur.
- À ce jour, on ne connaît pas le pire cas pour l'étude de tâches sporadiques (et échéance contrainte ou arbitraire).
- Voilà pourquoi il est fastidieux de vérifier la faisabilité de tâches sporadiques.

Prédictibilité I

- Nous venons de constater qu'à propos de l'ordonnancement de tâches sporadiques nous ne disposons pas de la notion de pire cas.
- En revanche, pour l'ordonnancement de tâches périodiques/sporadiques les méthodes sont prédictibles, mais commençons par une définition.

Définition 8 (Prédictible)

Un algorithme d'ordonnancement est **prédictif** si la diminution des durées d'exécution des travaux n'entrave pas l'ordonnançabilité du système. Cette notion a été étendue pour les autres caractéristiques des tâches/travaux (périodes, échéances) sous la nom de **viabilité**.

Prédictibilité II

- Le caractère prédictible des algorithmes d'ordonnancement est important car il permet de vérifier l'ordonnançabilité en se focalisant sur les **pire temps d'exécution** des travaux ou tâches et non pas sur les temps d'exécution **exacts**.
- HA et LIU en 1994 ont montré que les ordonnanceurs qui attribuent des priorités fixes aux travaux (ou aux tâches) sont prédictibles.
- Propriété étendue par Cucu et al. en 2010 pour des architectures hétérogènes [Cucu-Grosjean and Goossens, 2010].

Deux cas « simples »

- Système à échéance sur requête : test exact et optimalité
- Système périodique : test exact

Systèmes à échéance sur requête — Une condition nécessaire et suffisante I

- Remarquons que l'on a aucun espoir d'ordonnancer des systèmes avec
  - $U(\tau) > m$ (on demande trop de ressources) ou
  - $U_{\max}(\tau) > 1$ (le parallélisme au sein des travaux étant interdit)
- Par ailleurs, cette double condition est aussi suffisante pour les systèmes à échéance sur requête :

Théorème 3 ([Baruah et al., 1996])

Pour un ensemble sporadique à échéance sur requête la condition suivante est ***suffisante et nécessaire*** *pour l'ordonnançabilité sur m processeurs :*

$$U(\tau) \leqslant m \qquad \text{et} \qquad U_{\max}(\tau) \leqslant 1$$

Systèmes à échéance sur requête — Une condition nécessaire et suffisante II

- Ceci suppose de pouvoir préempter des tâches à des moments arbitraires et d'exécuter des tâches pendant des durées arbitrairement petites.

Ordonnancement *fair*

- 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 $[t, t + 1)$ et $\mathcal{S}(\tau_i, t) = 0$ sinon.
- Dans un ordonnancement (idéal) parfaitement « *fair* », chaque tâche τ_i doit recevoir exactement $U(\tau_i) \times t$ 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.

La notion de **retard** (*lag*)

- 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 $\text{retard}(\tau_i, t)$ est défini de la manière suivante :

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

Ordonnancement PFAIR I

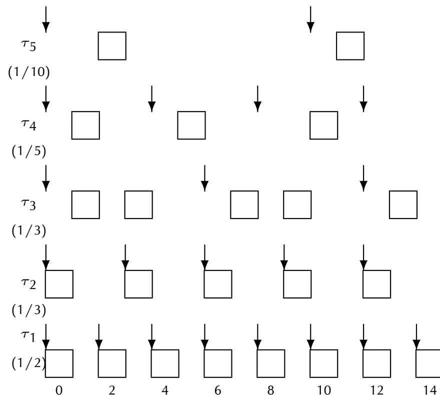
Définition 9 (Ordonnancement PFAIR)

Un ordonnancement est dit PFAIR si et seulement si :

$$-1 < \mathrm{retard}(\tau_i, t) < 1 \qquad \forall \tau_i \in \tau, t \in \mathbb{Z}. \tag{2}$$

- De manière informelle, l'équation 2 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 $\lceil U(\tau_i) \cdot t \rceil$.

PFAIR Example



Optimalité de PFAIR

Théorème 4 ([Baruah et al., 1996])

Soit τ un système sporadique à échéance sur requête et départ simultané, il existe un ordonnancement PFAIR *sur m processeurs identiques de capacité 1 si et seulement si :*

$$U(\tau) \leqslant m \qquad \text{et} \qquad U_{\max} \leqslant 1. \tag{3}$$

Ordonnanceur(s) PFAIR

En ce qui concerne les ordonnanceurs, à ce jour, trois algorithmes d'ordonnancement PFAIR optimaux ont été proposés : PF, PD et PD^2.

Systèmes périodiques à priorité fixe au niveau des tâches

Définition 10 (Intervalle d'étude)

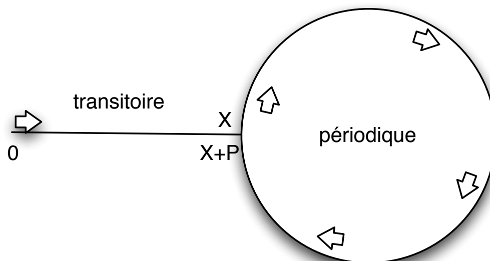
Pour un algorithme d'ordonnancement et un ensemble de tâches, un **intervalle d'étude** est un intervalle **fini** tel que le système est ordonnançable si et seulement si toutes les échéances dans l'intervalle d'étude sont respectées.

- Les premiers intervalles d'études sont basés sur des propriétés de **périodicité** des ordonnancements

Périodicité de l'ordonnancement

Théorème 5 ([Cucu and Goossens, 2007])

En multiprocesseur les ordonnanceurs déterministes et sans mémoire produisent des ordonnancements périodiques.



Système à départ différé et à échéance contrainte

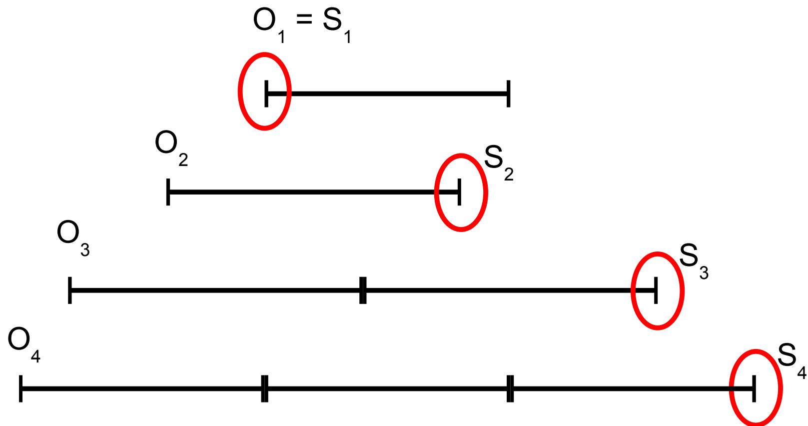
Théorème 6 ([Cucu and Goossens, 2007])

Soit $\mathcal{A}$ un ordonnanceur à priorité fixe au niveau des tâches et soit τ un système de n tâches périodiques à départ différé et à échéance contrainte, si τ est ordonnançable sous l'égide de l'algorithme $\mathcal{A}$ alors l'ordonnancement est lui-même périodique dès l'instant S_n avec la définition suivante :

$$S_i \stackrel{\text{def}}{=} \begin{cases} O_1 & \textit{si } i = 1 \\ \max\{O_i, O_i + \lceil \frac{S_{i-1} - O_i}{T_i} \rceil T_i\} & \textit{sinon} \end{cases}$$

et $P_i = \text{ppcm}\{T_1, T_2, \ldots, T_i\}$.

Illustration S_n



Tests d'ordonnançabilité

Théorème 7

Un système périodique à départ différé et à échéance contrainte pour un ordonnanceur à priorité fixe au niveau des tâches est ordonnançable ***si et seulement si***

- *Toutes les échéances sont rencontrées dans l'intervalle* $[0, S_n + P_n)$
- *et le système à l'instant* $S_n + P_n$ *est dans le même état qu'à l'instant* S_n.

Tests exacts et complexité

- Sans connaissance du pire cas, les auteurs de [Baker and Cirinei, 2007] proposent un algorithme force brute qui explore « tous » les scénarios à la recherche d'un état d'erreur.
- Plus récemment en appliquant des méthodes de vérification formelle les auteurs de [Lindström et al., 2011] ont proposés un algorithme efficace.
- Les mêmes auteurs ont prouvé que la complexité du problème d'ordonnancabilité est **PSpace-complete**.

Conditions Suffisantes

Théorème 8 ([Srinivasan and Baruah, 2002])

*Pour un ensemble sporadique à échéance sur requête la condition suivante est **suffisante** pour l'ordonnançabilité sur m processeurs avec « global* EDF *» :*

$$U(\tau) \leqslant m - (m-1)U_{\max}(\tau)$$

ou encore

$$m \geqslant \frac{U(\tau) - U_{\max}(\tau)}{1 - U_{\max}(\tau)}$$

L'algorithme EDF$^{(k)}$ I

- Afin d'alléger les notations nous allons supposer pour l'algorithme EDF$^{(k)}$ que $U(\tau_1) \geqslant U(\tau_2) \geqslant \cdots \geqslant U(\tau_n)$.
- Nous introduisons aussi la notation $\tau^{(i)}$ pour dénoter le système constitué des $(n - i + 1)$ tâches avec les plus petites utilisations de τ :

$$\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_i, \tau_{i+1}, \ldots, \tau_n\}$$

- Remarquons dès lors qu'à partir du Théorème précédent nous pouvons déduire que

$$m \geqslant \left\lceil \frac{U(\tau) - U(\tau_1)}{1 - U(\tau_1)} \right\rceil$$

L'algorithme EDF$^{(k)}$ II

- Si l'on n'est pas dans l'obligation d'utiliser le « véritable » EDF on peut, dans de nombreux cas, utiliser **moins** que $\lceil \frac{U(\tau)-U(\tau_1)}{1-U(\tau_1)} \rceil$ processeurs. Par exemple si l'on considère l'algorithme EDF$^{(k)}$

Définition 11 (EDF$^{(k)}$)

- Pour tout $i < k$, les travaux de τ_i ont une priorité maximale (ce qui peut se faire pour un ordonnanceur EDF en ajustant les échéances à $-\infty$).
- Pour tout $i \geqslant k$, les travaux de τ_i reçoivent une priorité définie par EDF.

L'algorithme EDF$^{(k)}$ III

- En d'autres termes, l'algorithme EDF$^{(k)}$ donne une plus grande priorité aux travaux des $(k-1)$ premières tâches de τ et ordonnance les autres travaux en utilisant EDF.
- Notons que le « véritable » EDF correspond à EDF$^{(1)}$.

L'algorithme EDF$^{(k)}$ IV

Théorème 9 ([Goossens et al., 2003])

Un système sporadique à échéance sur requête τ est ordonnancable sur m processeurs par l'algorithme EDF$^{(k)}$*, avec*

$$m = (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1 - U(\tau_k)} \right\rceil \qquad (4)$$

L'algorithme EDF$^{(k)}$ – Example I

Considérons le système τ composé de 5 tâches :

	C	T
τ_1	9	10
τ_2	14	19
τ_3	1	3
τ_4	2	7
τ_5	1	5

$U(\tau) \approx 2.457$
$U_{\max}(\tau) = 0.9$

Nous pouvons vérifier pour ce système, que l'équation 4 est minimisée pour $k = 3$; dès lors, τ peut être ordonnancé avec l'algorithme EDF$^{(3)}$ sur **3 processeurs**.

L'algorithme $\text{EDF}^{(k)}$ – Example II

- En revanche, pour le « véritable » EDF on ne peut garantir que l'ordonnançabilité du système mais seulement si l'on dispose de $\lceil \frac{U(\tau)-U(\tau_1)}{1-U(\tau_1)} \rceil \approx \lceil 1.557/0.1 \rceil =$ **16 processeurs**.
- Des études expérimentales montrent que cette amélioration n'est pas anecdotique.

Perspectives futures

- Systèmes à échéance implicite : optimalité sans fairness, i.e, avec peu de migration/préemption
- Solutions pour des architectures hétérogènes
- Techniques hybrides par exemple semi-partitionnées : peu de migration sans grande perte de capacité
- Solutions approchées par exemple schéma d'approximation polynomial (PTAS).

Conclusion

- Présenté des résultats importants de l'ordonnancement multiprocesseur
- Montré les propriétes/problèmes intrinsèques au multiprocesseur
- Identifié des questions laissées ouvertes

Questions

Bibliographie I

[Baker and Cirinei, 2007] Baker, T. and Cirinei, M. (2007).
Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks.
In *Proceedings of the 11th international conference on Principles of distributed systems*, pages 62–75. Springer-Verlag.

[Baruah, 2007] Baruah, S. (2007).
Techniques for multiprocessor global schedulability analysis.
In *Real-Time Systems Symposium*, pages 119–128. IEEE Computer Society.

[Baruah et al., 1996] Baruah, S., Cohen, N., Plaxton, C. G., and Varvel, D. (1996).
Proportionate progress : A notion of fairness in resource allocation.
Algorithmica, 15 :600–625.

Bibliographie II

[Cucu and Goossens, 2007] Cucu, L. and Goossens, J. (2007).
Feasibility intervals for multiprocessor fixed-priority schedul ing of arbitrary deadline periodic systems.
In Society, I. C., editor, *Design Automation and Test in Europe (DATE'07)*, pages 1635–1640.

[Cucu-Grosjean and Goossens, 2010] Cucu-Grosjean, L. and Goossens, J. (2010).
Predictability of fixed-job priority schedulers on heterogeneous multiprocessor real-time systems.
Information Processing Letters, 110 :399–402.

Bibliographie III

[Fisher et al., 2010] Fisher, N., Goossens, J., and Baruah, S. (2010).
Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible.
Real-Time Systems : The International Journal of Time-Critical Computing, 45(1–2) :26–71.

[Goossens et al., 2003] Goossens, J., Funk, S., and Baruah, S. (2003).
Priority-driven scheduling of periodic task systems on multiprocessors.
Real-Time Systems : The International Journal of Time-Critical Computing, 25 :187–205.

[Hong and Leung, 1988] Hong, K. and Leung, J. (1988).
On-line scheduling of real-time tasks.
In *Real-Time Systems Symposium*.

Bibliographie IV

[Lindström et al., 2011] Lindström, M., Geeraerts, G., and Goossens, J. (2011).
A faster exact multiprocessor schedulability test for sporadic tasks.
In Burns, A. and George, L., editors, *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS 2011) September 29–30 Nantes France*, pages 25–34.
Best student paper award.

[Phillips et al., 1997] Phillips, C., Stein, C., Torng, E., and Wein, J. (1997).
Optimal time-critical scheduling via resource augmentation.
In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 140–149. ACM.

Bibliographie V

[Srinivasan and Baruah, 2002] Srinivasan, A. and Baruah, S. (2002).
Deadline-based scheduling of periodic task systems on multiprocessors.
Information Processing Letters, 84(2) :93–98.