Energy-efficient Scheduling

Guillaume Aupy
basé sur un travail fait avec Anne Benoit

8 avril 2013

EJCIM 2013

Motivation

- Ordonnancement classique : choisir quelle tâche assigner à quel processeur (et à quel moment) pour minimiser le temps d'exécution.

- **Cas plus général**
  On a plus de temps que nécessaire (pas besoin d'aller le plus vite possible). Comment l'utiliser efficacement ?
  - Avantages environnementaux : ↘ Énergie.
  - Problème de tolérance aux fautes : ↘ Fiabilité.

Objectif : utiliser efficacement la technique "speed scaling"

Motivation

- ~~Ordonnancement classique : choisir quelle tâche assigner à quel processeur (et à quel moment) pour minimiser le temps d'exécution.~~

- **Cas plus général**
  On a plus de temps que nécessaire (pas besoin d'aller le plus vite possible). Comment l'utiliser efficacement ?
  - Avantages environnementaux : ↘ Énergie.
  - Problème de tolérance aux fautes : ↘ Fiabilité.

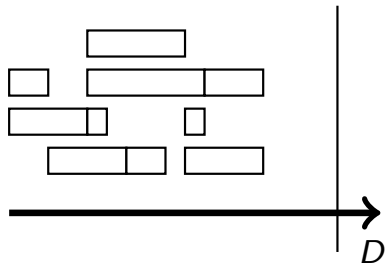
Objectif : utiliser efficacement la technique "speed scaling"

Motivation

- ~~Ordonnancement classique : choisir quelle tâche assigner à quel processeur (et à quel moment) pour minimiser le temps d'exécution.~~

- **Cas plus général**
  On a plus de temps que nécessaire (pas besoin d'aller le plus vite possible). Comment l'utiliser efficacement ?
  - Avantages environnementaux : ↘ Énergie.
  - Problème de tolérance aux fautes : ↘ Fiabilité.

Objectif : utiliser efficacement la technique "speed scaling"

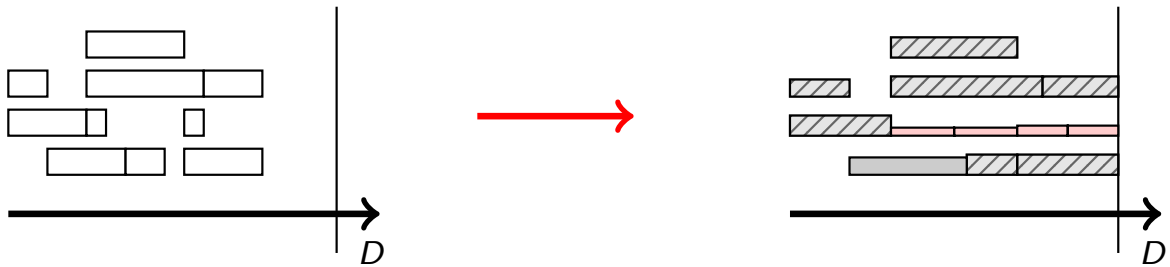


Motivation

- ~~Ordonnancement classique : choisir quelle tâche assigner à quel processeur (et à quel moment) pour minimiser le temps d'exécution.~~

- **Cas plus général**
  On a plus de temps que nécessaire (pas besoin d'aller le plus vite possible). Comment l'utiliser efficacement ?
  - Avantages environnementaux : ↘ Énergie.
  - Problème de tolérance aux fautes : ↘ Fiabilité.

Objectif : utiliser efficacement la technique "speed scaling"



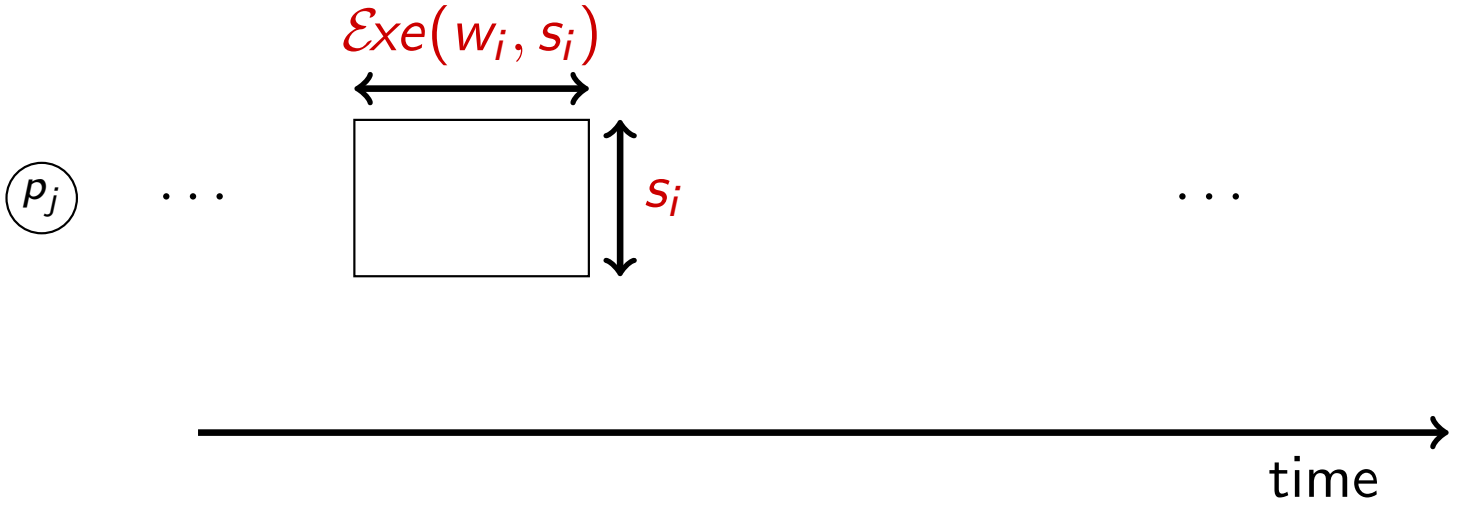
Architecture

p processeurs identiques.

Speed Scaling : consiste à modifier la vitesse d'execution s d'une tâche, $s \in [s_{\min}, s_{\max}]$.

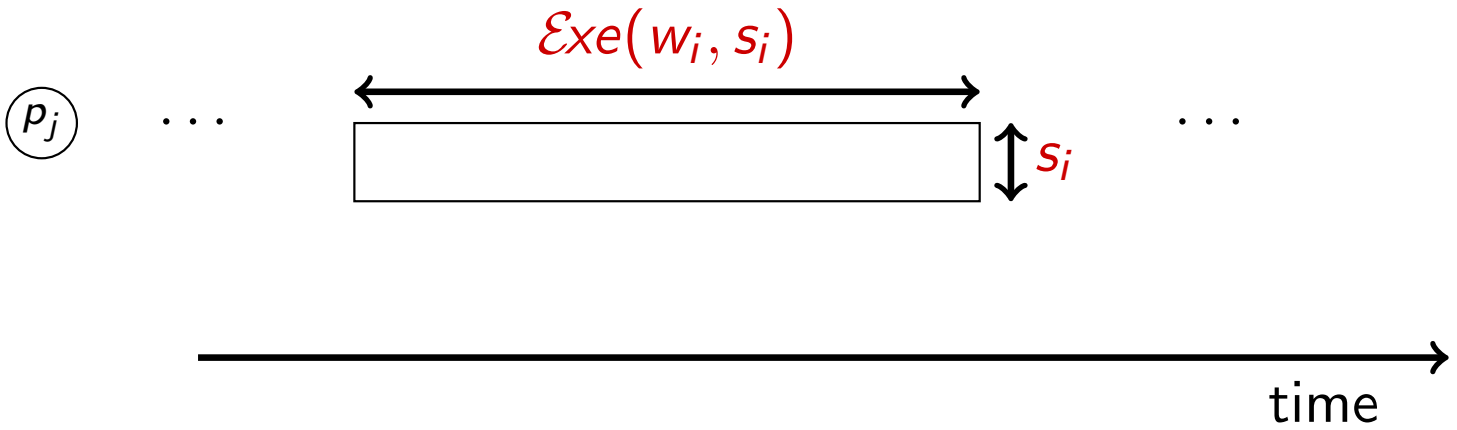
Plus précisement

Soit une tâche T_i de poids w_i exécutée sur le processeur p_j :



Plus précisement

Soit une tâche T_i de poids w_i exécutée sur le processeur p_j :



Hypothèse

Dans ce modèle on supposera en plus l'existence d'une vitesse $s_{\mathtt{rel}}$, telle que, pour toute tâche T_i exécutée à s_i :

Si $s_i < s_{\mathtt{rel}}$ alors T_i nécessite une deuxième exécution.

Hypothèse

Dans ce modèle on supposera en plus l'existence d'une vitesse $s_{\mathtt{rel}}$, telle que, pour toute tâche T_i exécutée à s_i :

Si $s_i < s_{\mathtt{rel}}$ alors T_i nécessite une deuxième exécution.

Pourquoi ?

Fiabilité

Faute transiente = faute locale (pas d'impact sur le système, redémarrage immédiat)

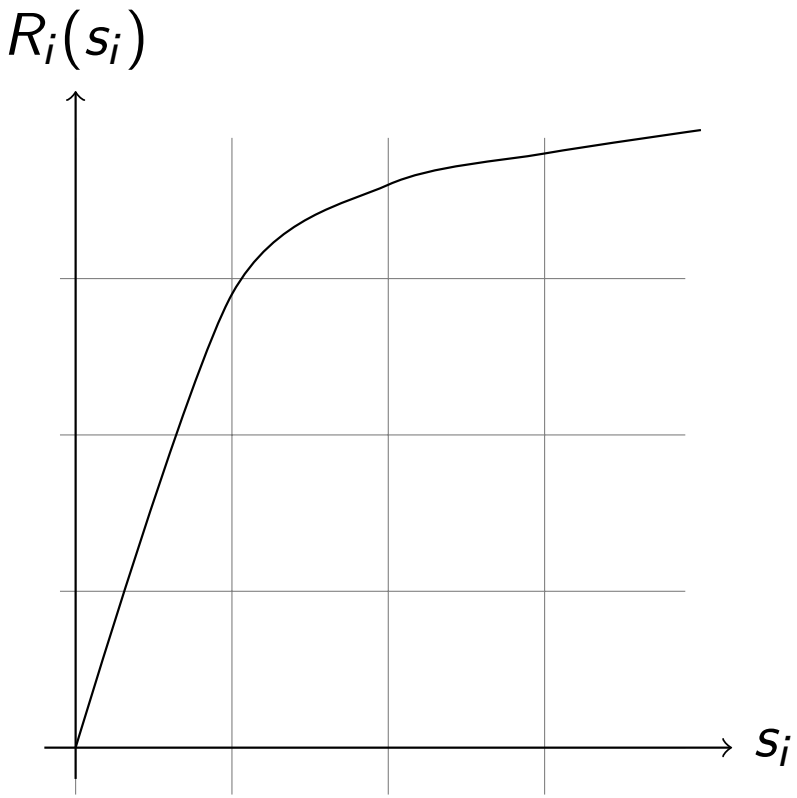
Le taux de fautes transientes suit une loi de Poisson de paramètre :

$$\lambda(s) = \lambda_0 e^{d \frac{s_{\max} - s}{s_{\max} - s_{\min}}}$$

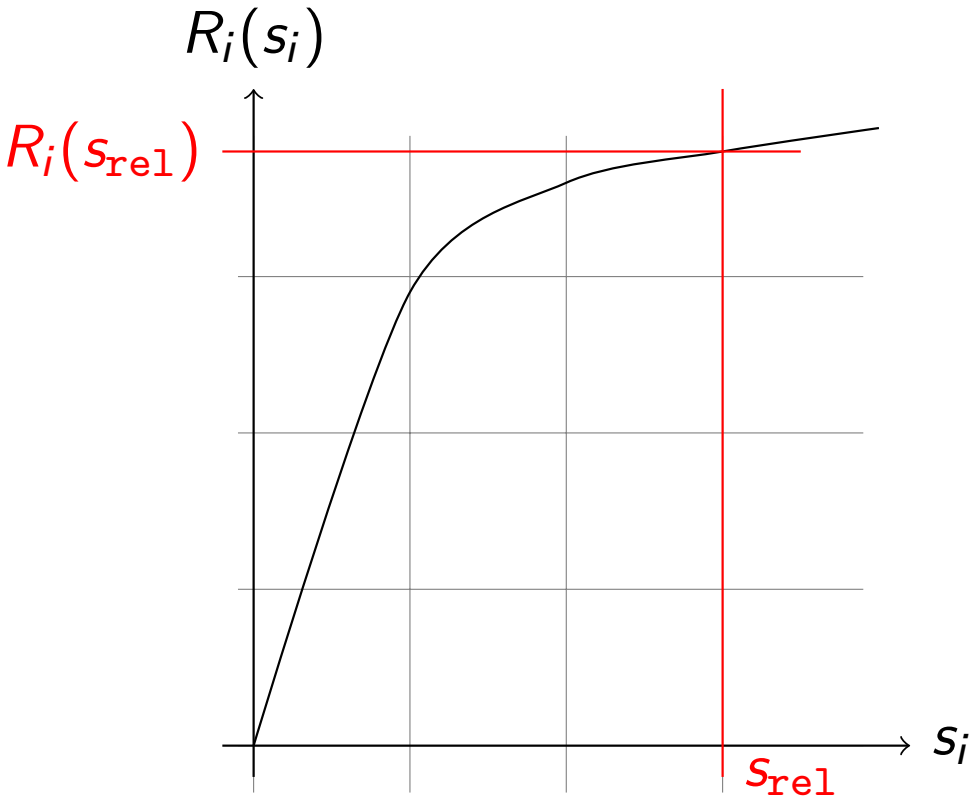
Fiabilité de T_i exécutée à s_i :

$$\begin{aligned} R_i(s_i) &= e^{-\lambda(s_i)\mathcal{E}xe(w_i, s_i)} \\ &\approx 1 - \lambda(s_i)\mathcal{E}xe(w_i, s_i) \end{aligned}$$

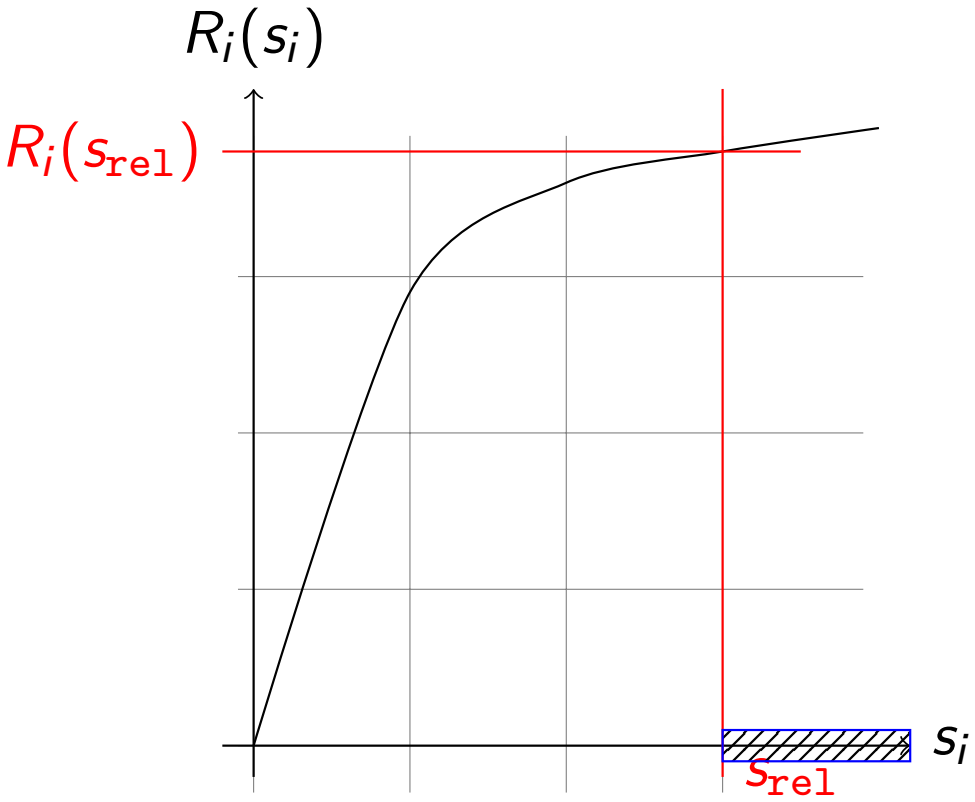
Fiabilité



Fiabilité



Fiabilité



Fiabilité

Fiabilité de T_i exécutée à s_i :

$$R_i(s_i) = e^{-\lambda(s_i)\mathcal{E}xe(w_i,s_i)}$$
$$\approx 1 - \lambda(s_i)\mathcal{E}xe(w_i, s_i)$$

Fiabilité de deux exécutions

$$R_i = 1 - (1 - R_i(s_i^{(1)}))(1 - R_i(s_i^{(2)}))$$

Fiabilité

Fiabilité de T_i exécutée à s_i :

$$R_i(s_i) = e^{-\lambda(s_i)\mathcal{E}xe(w_i,s_i)}$$
$$\approx 1 - \lambda(s_i)\mathcal{E}xe(w_i, s_i)$$

Fiabilité de deux exécutions

$$R_i = 1 - (1 - R_i(s_i^{(1)}))(1 - R_i(s_i^{(2)}))$$

$1 - (1 - R_i(s_{\min}))^2 \geq R_i(s_{\texttt{rel}})$

$\hookrightarrow$ Deux exécutions garantissent la fiabilité.

Temps d'exécution

Le temps d'exécution de T_i à la vitesse s_i est :

$$\mathcal{E}xe(w_i, s_i) = \frac{w_i}{s_i}$$

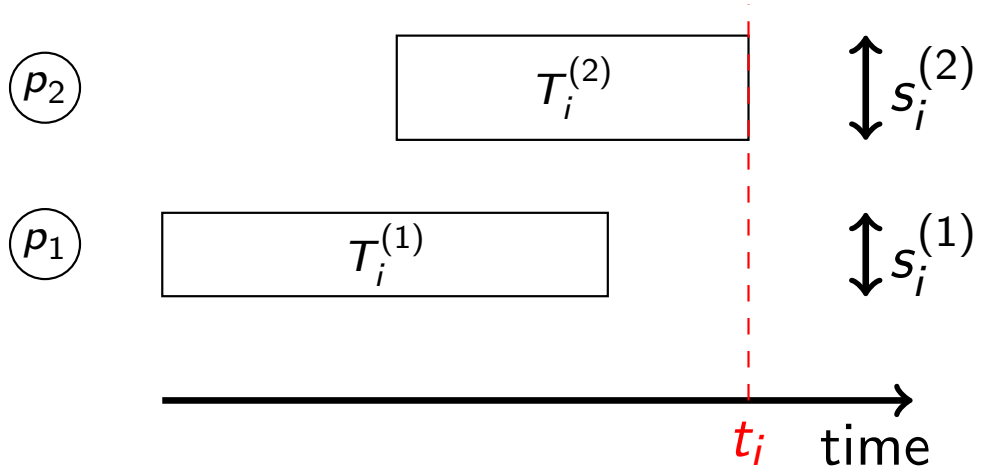
On note t_i le temps de fin de la dernière exécution de T_i :

Temps d'exécution

Le temps d'exécution de T_i à la vitesse s_i est :

$$\mathcal{E}xe(w_i, s_i) = \frac{w_i}{s_i}$$

On note t_i le temps de fin de la dernière exécution de T_i :



Temps d'exécution

Contrainte de temps d'exécution :

On demande $\forall i,\ t_i \leq D$ (la deadline D est une contrainte fixée par l'utilisateur)

Pourquoi diminuer la vitesse alors ?

- on perd en fiabilité
- on perd en temps d'exécution

Énergie

L'exécution d'une tâche T_i à vitesse s_i consomme :

$$E_i(s_i) = \mathcal{E}xe(w_i, s_i)s_i^3 = w_i s_i^2$$

$\rightarrow$ (c'est l'énergie dynamique du modèle classique d'énergie)

Énergie consommée avec deux exécutions :

$$E_i = w_i \left(s_i^{(1)}\right)^2 + w_i \left(s_i^{(2)}\right)^2$$

Énergie

L'exécution d'une tâche T_i à vitesse s_i consomme :

$$E_i(s_i) = \mathcal{E}xe(w_i, s_i)s_i^3 = w_i s_i^2$$

$\rightarrow$ (c'est l'énergie dynamique du modèle classique d'énergie)

Énergie consommée avec deux exécutions :

$$E_i = w_i \left(s_i^{(1)}\right)^2 + w_i \left(s_i^{(2)}\right)^2$$

Résumé

On cherche donc ici à décider pour chaque tâche :

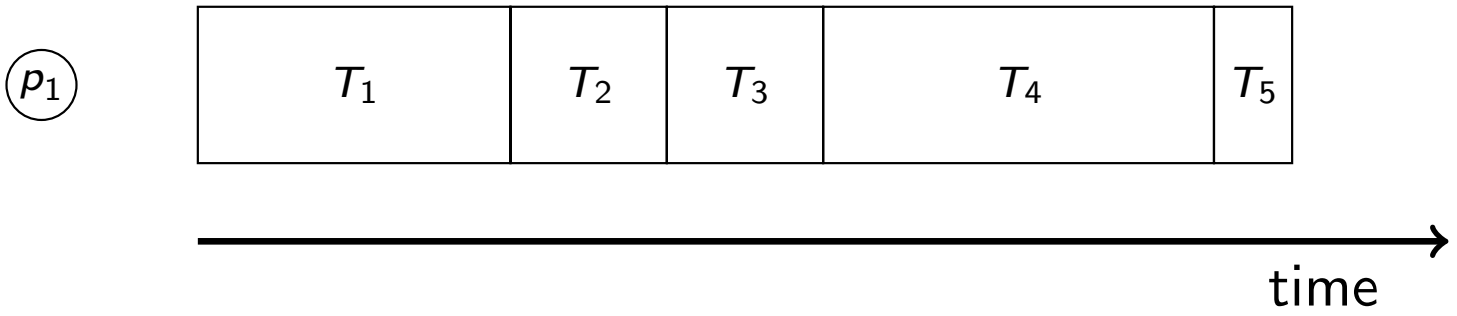
- du nombre d'exécutions (une ou deux)
- de la vitesse de ces exécutions
- de la localisation de ces exécutions (quels processeurs)

Dans le but de minimiser l'énergie sous contrainte :

- $\forall i,\ t_i \leq D$ (fin d'exécution bornée)
- $\forall i,\ s_i \geq s_{\mathtt{rel}}$ ou deux exécutions (fiabilité minimale)

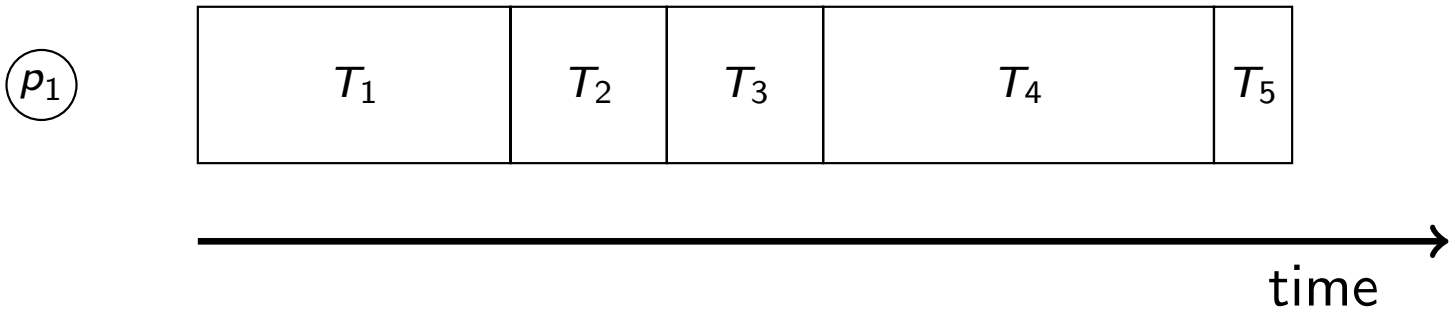
Chaîne de tâches

Soit n tâches $T_1, \ldots, T_n$ de poids respectifs $w_1, \ldots, w_n$ à exécuter, telles que pour tout i, T_i doit être exécutée avant T_{i+1}



Chaîne de tâches

Soit n tâches $T_1, \ldots, T_n$ de poids respectifs $w_1, \ldots, w_n$ à exécuter, telles que pour tout i, T_i doit être exécutée avant T_{i+1}



On se concentre pour l'instant sur le cas $p = 1$

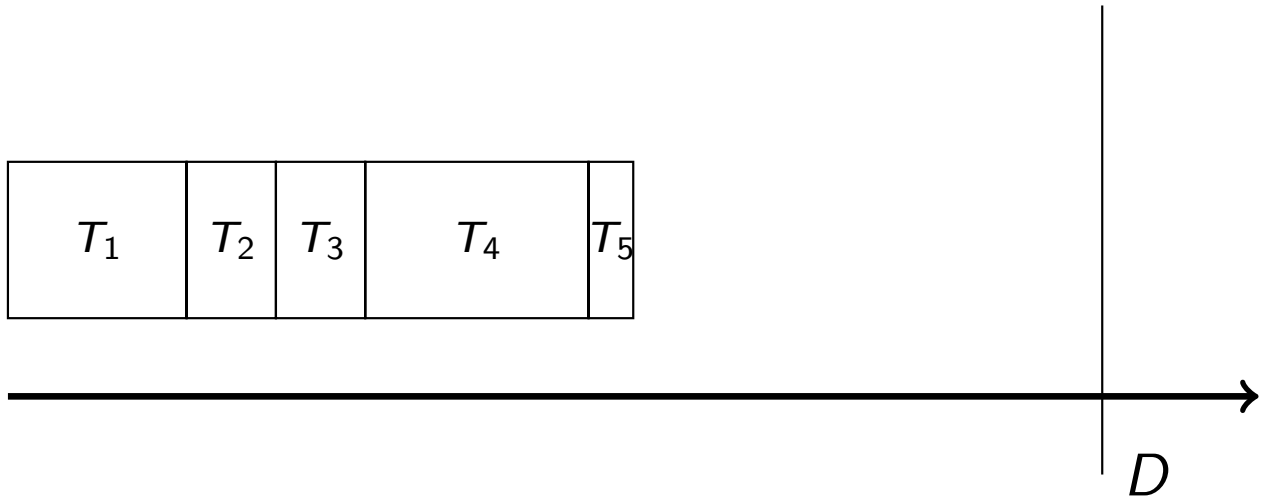
Lemme (Sans re-exécution)

Soit $S = \sum w_i$. La solution optimale sans re-exécution est d'exécuter toutes les tâches à $\max(s_{rel}, \frac{S}{D})$.

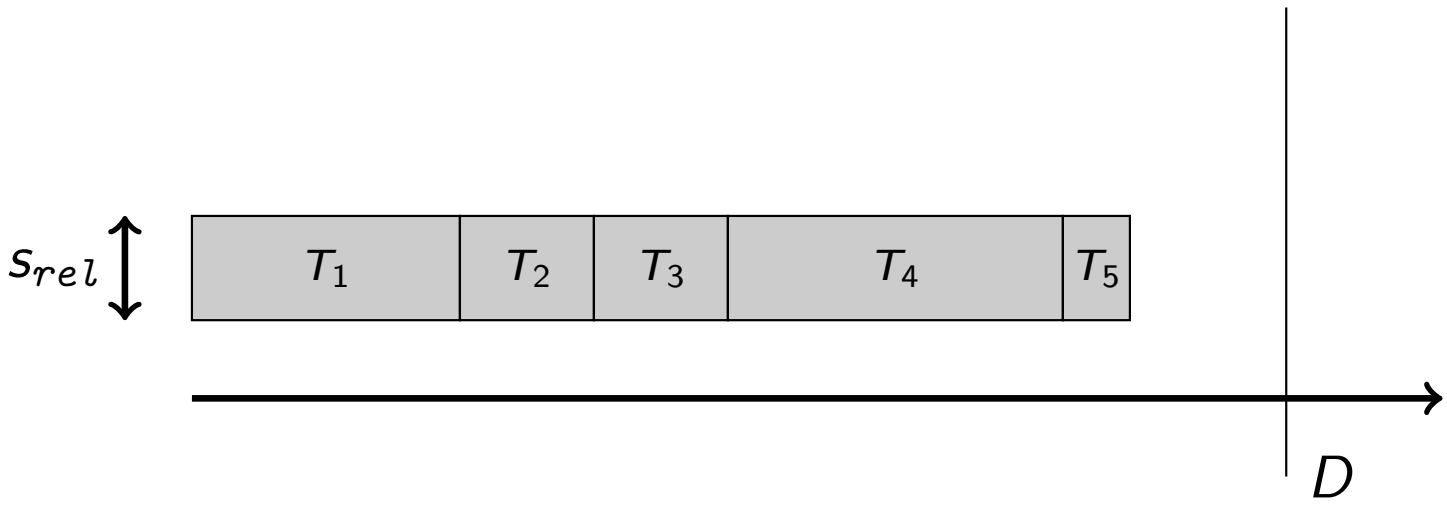
Lemme (Vitesses de re-exécution)

Si une tâche est exécutée deux fois, les deux exécutions sont à la même vitesse.

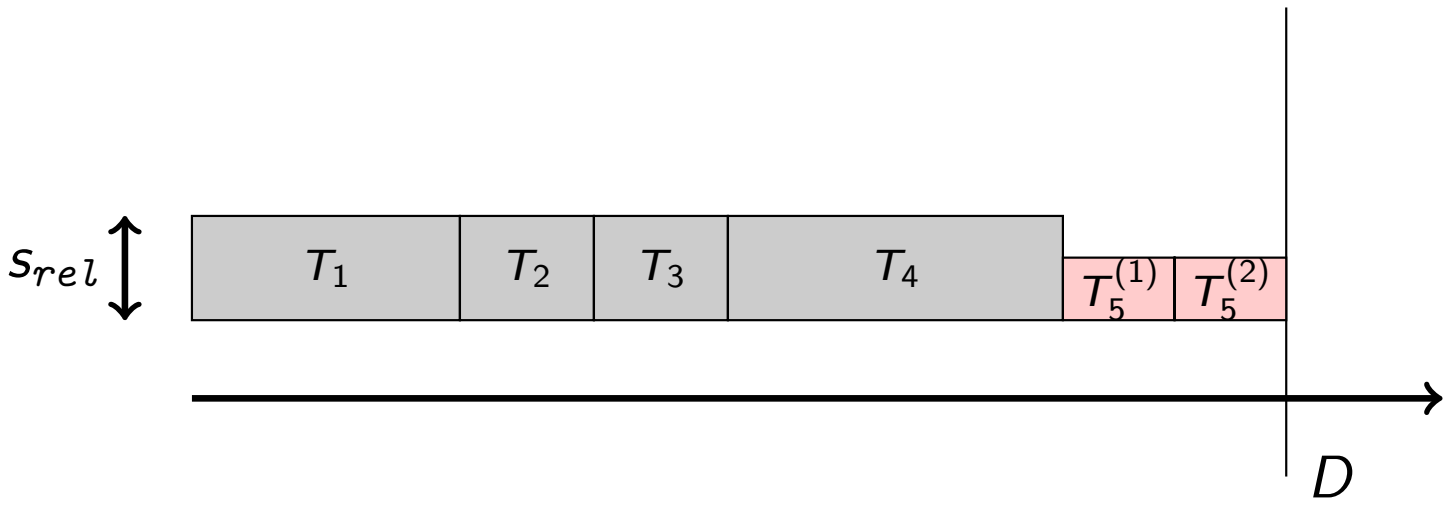
Lemme (Vitesse des tâches exécutées une seule fois)



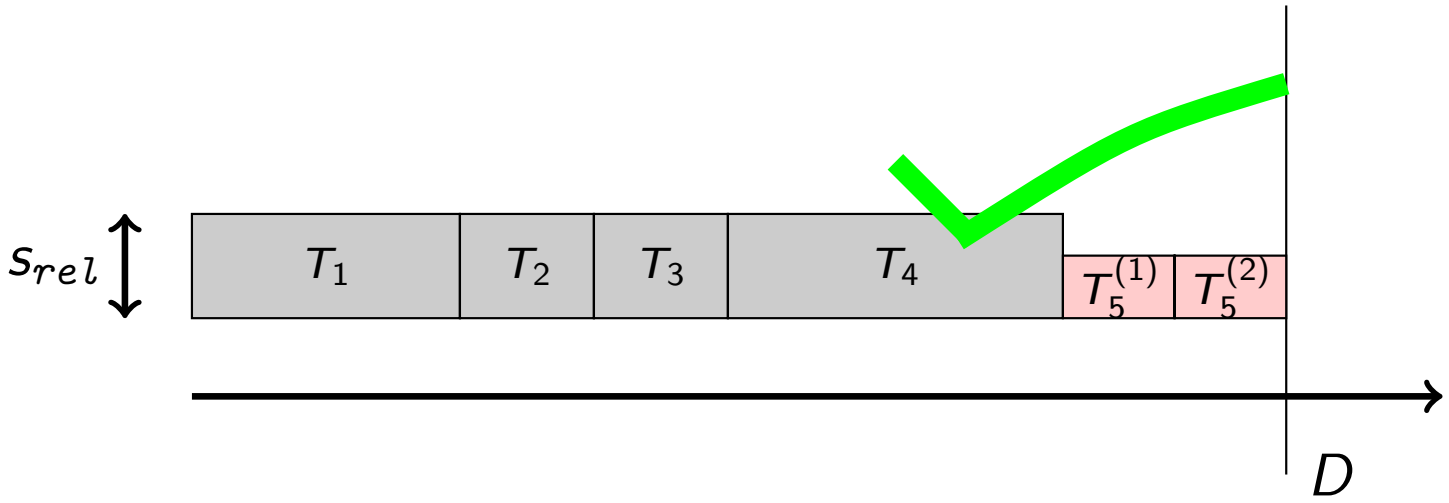
Lemme (Vitesse des tâches exécutées une seule fois)



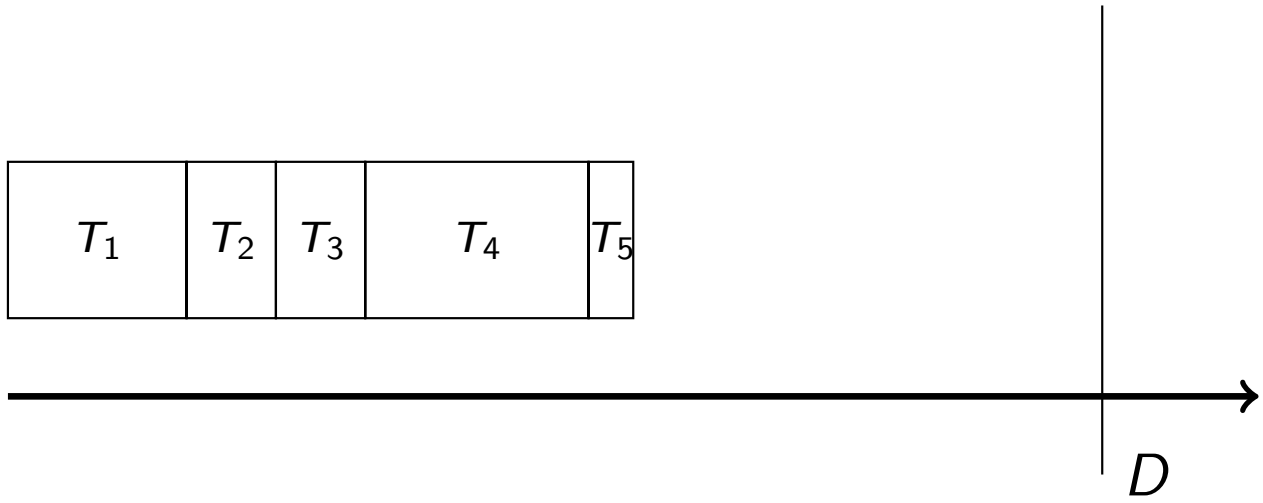
Lemme (Vitesse des tâches exécutées une seule fois)



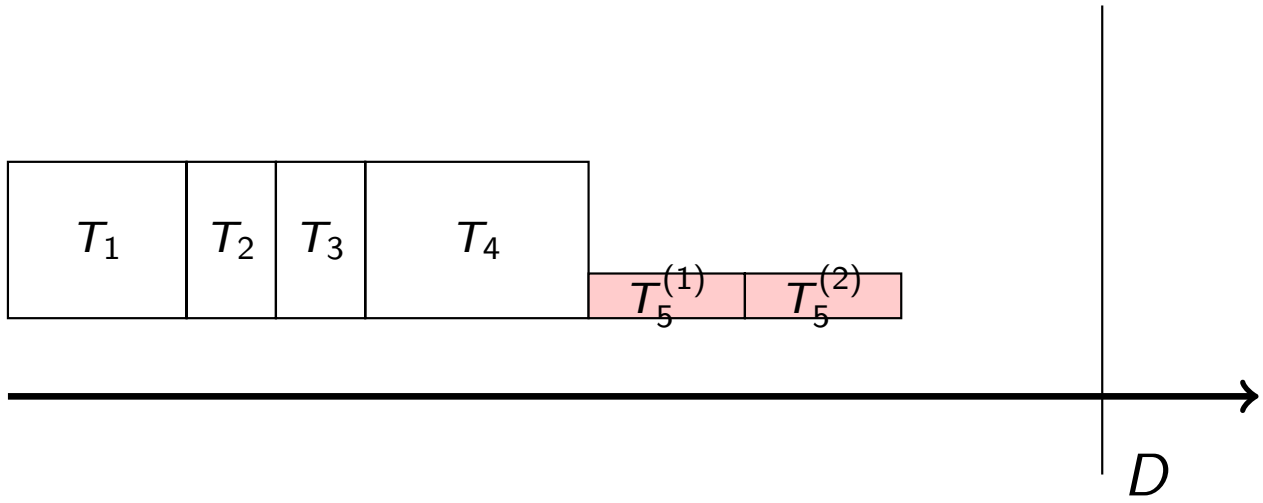
Lemme (Vitesse des tâches exécutées une seule fois)



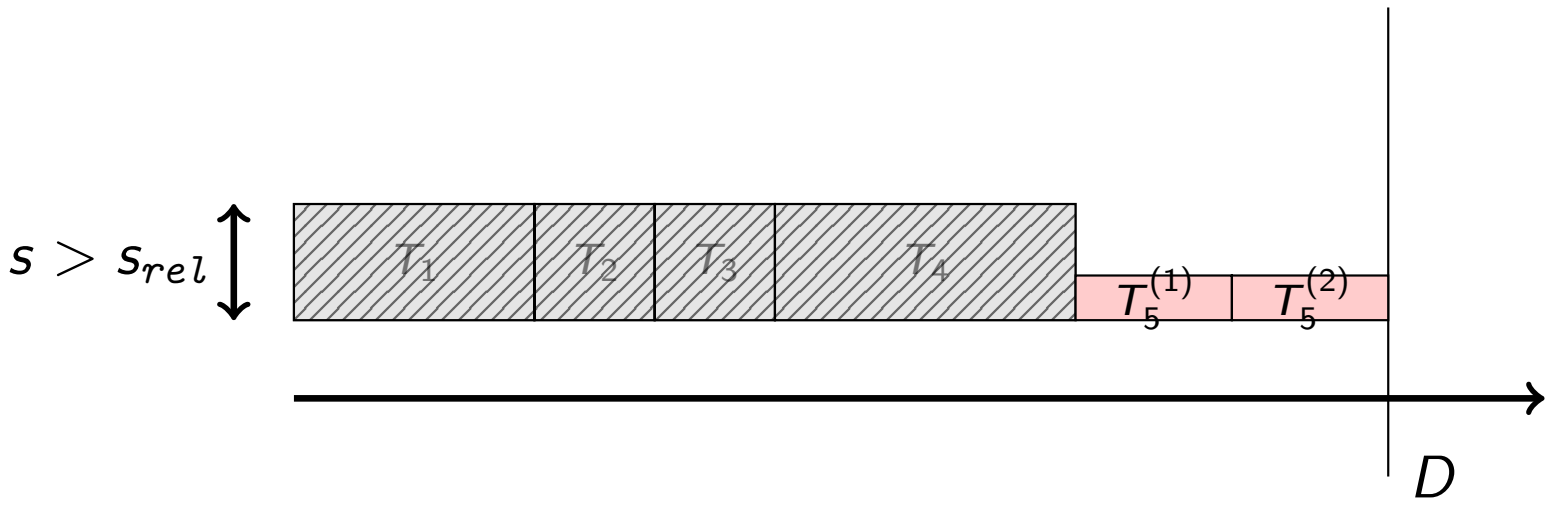
Lemme (Vitesse des tâches exécutées une seule fois)



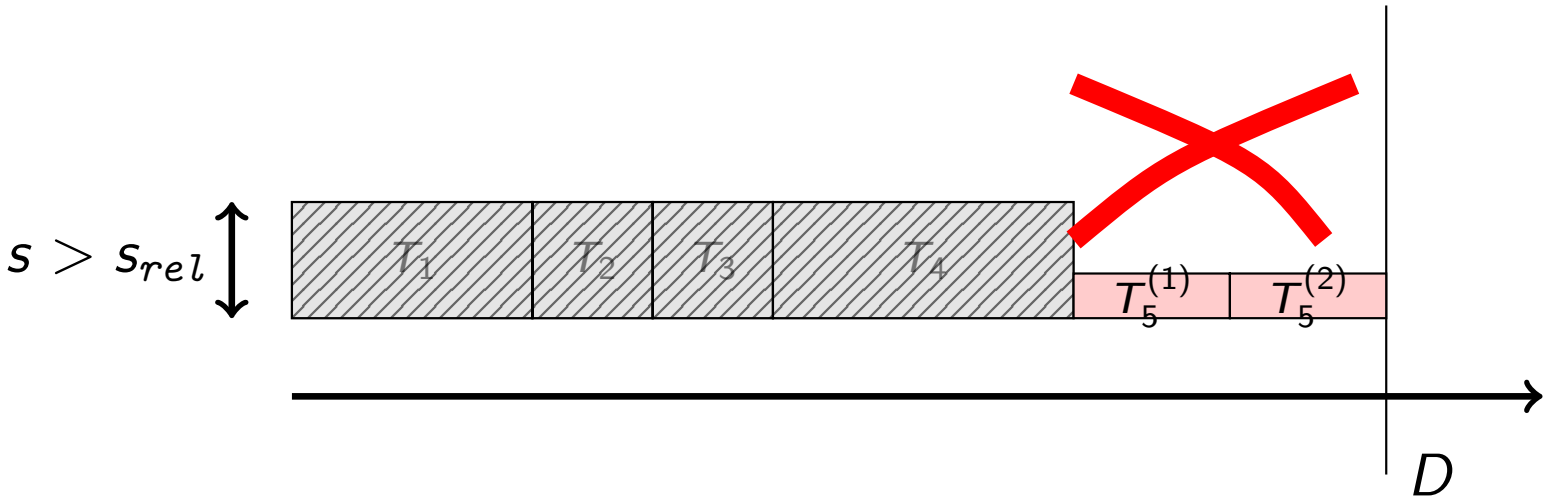
Lemme (Vitesse des tâches exécutées une seule fois)



Lemme (Vitesse des tâches exécutées une seule fois)



Lemme (Vitesse des tâches exécutées une seule fois)



Lemme (Vitesse des tâches exécutées une seule fois)

La vitesse des tâches exécutées seulement une fois est s_{rel} (quand c'est possible).

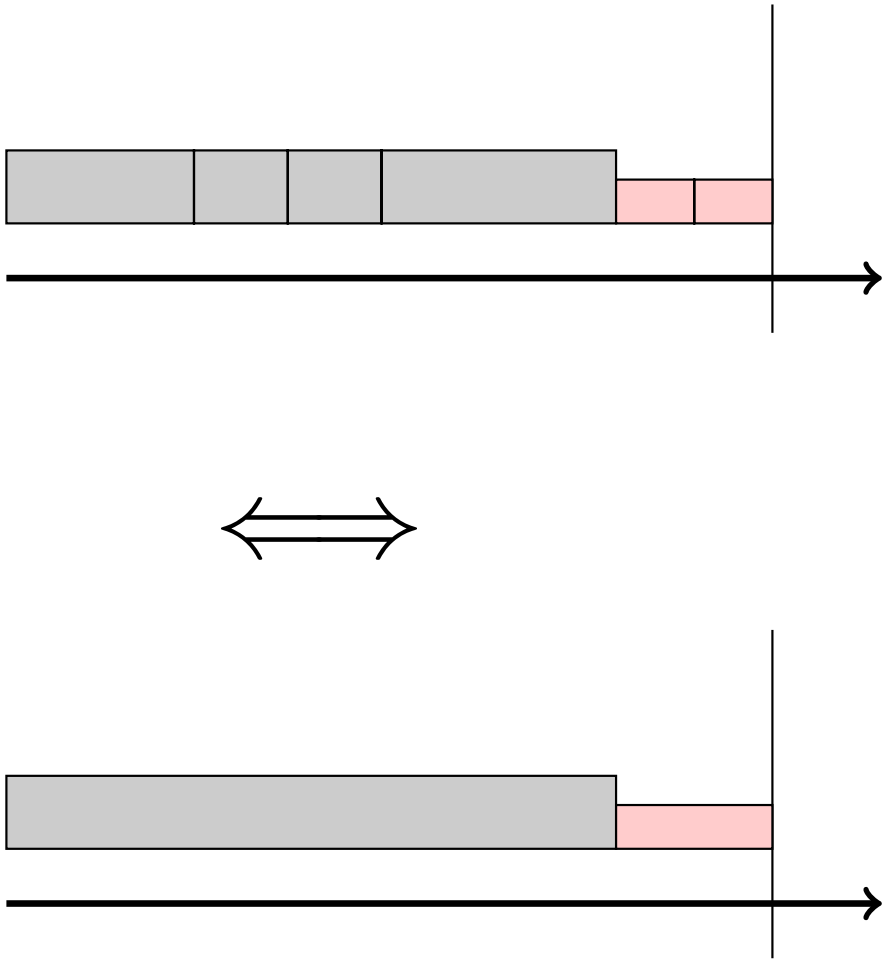
NP-complétude

On a vu :

- les tâches exécutées une fois le sont à $s_{\texttt{rel}}$;
- les tâches exécutées plusieurs fois le sont toutes à la même vitesse.

On va monter la NP-dureté de ce problème, en réduisant le problème à la sélection des tâches à re-exécuter.

NP-complétude



NP-complétude

En fait, l'ensemble des tâches à re-exécuter détermine entièrement la solution :

- l'ensemble X des tâches exécutées une seule fois l'est à $s_{\texttt{rel}}$;
- l'ensemble $\{T_1, \ldots, T_n\} \setminus X$ est re-exécuté en s'étalant dans le temps restant.

NP-complétude

La réduction se fait à partir du problème SUBSET-SUM

SUBSET-SUM

Soient $\{a_1, \ldots, a_n\}$, X. Existe-t-il un sous-ensemble I de $\{1, \ldots, n\}$ tel que $\sum_{i \in I} a_i = X$?

NP-complétude

La réduction se fait à partir du problème SUBSET-SUM

Il suffit de poser :

$$\begin{cases} \forall i,\ w_i = a_i \\ s_{\mathtt{rel}} = s_{\max} \\ D_0 = \dfrac{S}{s_{\mathtt{rel}}} + \dfrac{X}{c\ s_{\mathtt{rel}}} \\ E_0 = 2X \left(\dfrac{2c}{1+c} s_{\mathtt{rel}} \right)^2 + (S - X)\, s_{\mathtt{rel}}^2 \end{cases}$$

où c est la racine réelle positive du polynôme $7y^3 + 21y^2 - 3y - 1$ et on peut obtenir le résultat

Notions

Algorithme d'approximation : un algorithme tel que le résultat soit garanti pas trop loin de l'optimal.
Formellement A est une λ-approximation si

$$\forall I, Opt(I) \leq A(I) \leq \lambda \cdot Opt(I)$$

FPTAS : pour un problème *NP*-complet, on peut trouver une $(1+\varepsilon)$-approximation pour tout $\varepsilon > 0$, en temps polynomial en la taille du problème et en $\frac{1}{\varepsilon}$.

Notions

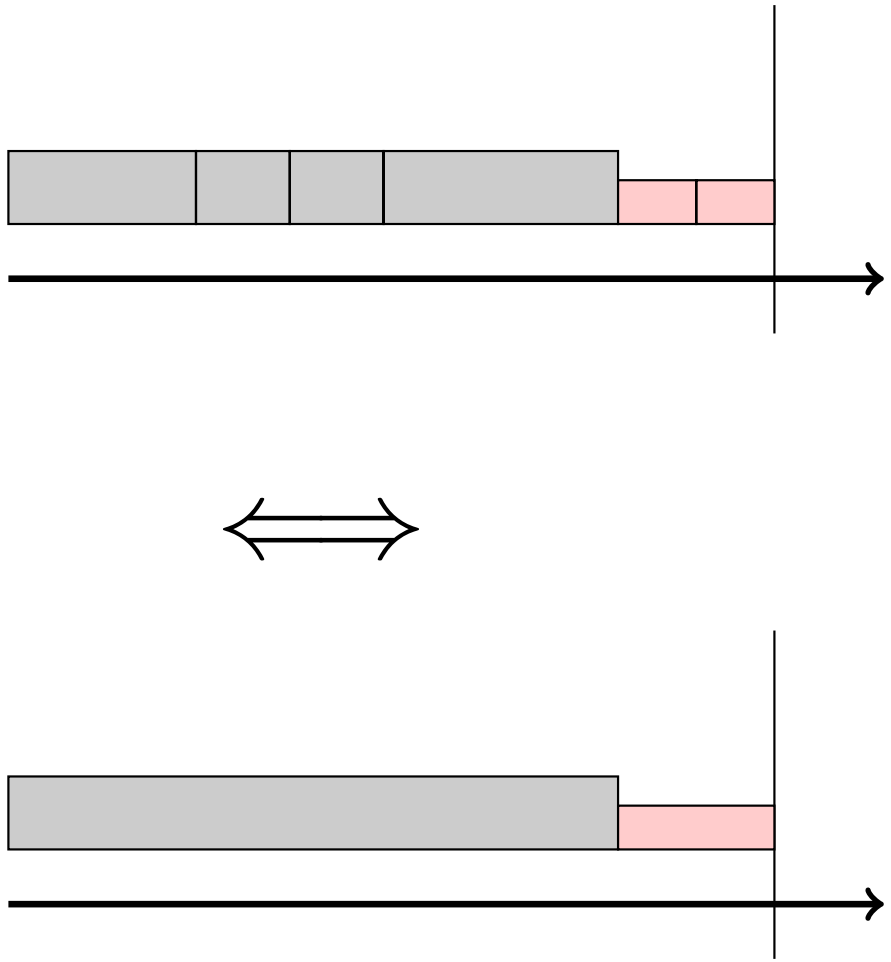
Algorithme d'approximation : un algorithme tel que le résultat soit garanti pas trop loin de l'optimal.
Formellement A est une λ-approximation si

$$\forall I, Opt(I) \leq A(I) \leq \lambda \cdot Opt(I)$$

FPTAS : pour un problème *NP*-complet, on peut trouver une $(1+\varepsilon)$-approximation pour tout $\varepsilon > 0$, en temps polynomial en la taille du problème et en $\frac{1}{\varepsilon}$.

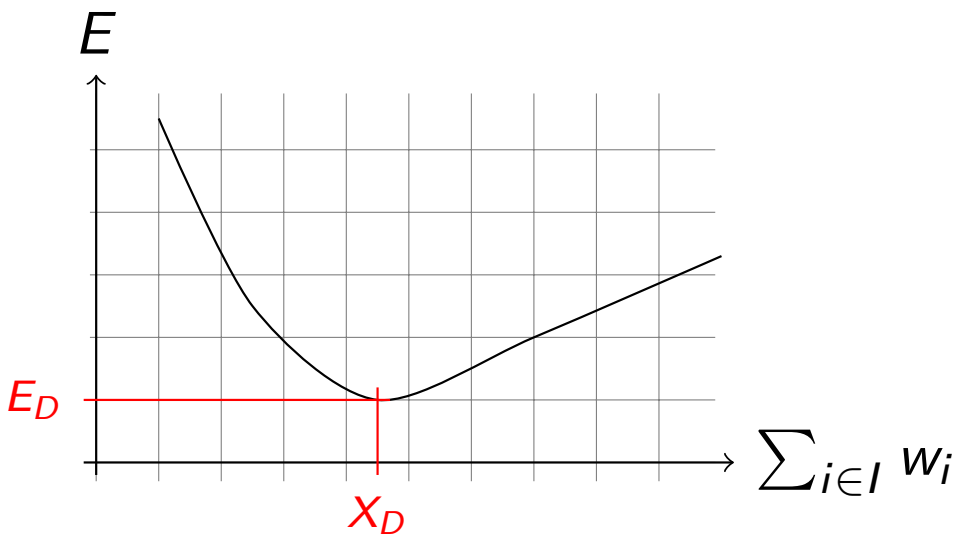
FPTAS pour le problème des chaînes

Rappel :



FPTAS pour le problème des chaînes

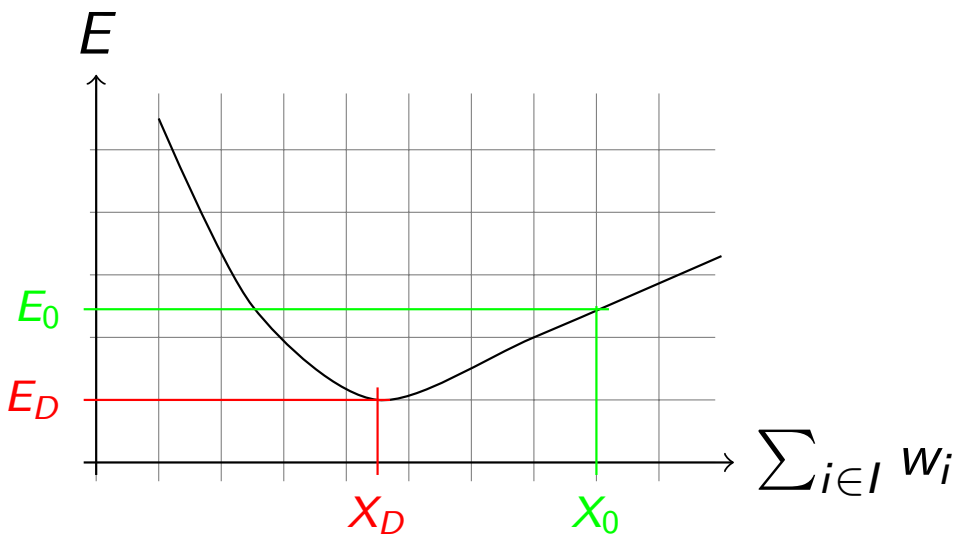
On cherche l'ensemble I des tâches à exécuter deux fois. Ce qu'on sait :



On sait calculer E_D une borne inférieure sur l'énergie, mais on ne sait pas où se situe E_0 le minimum atteignable.

FPTAS pour le problème des chaînes

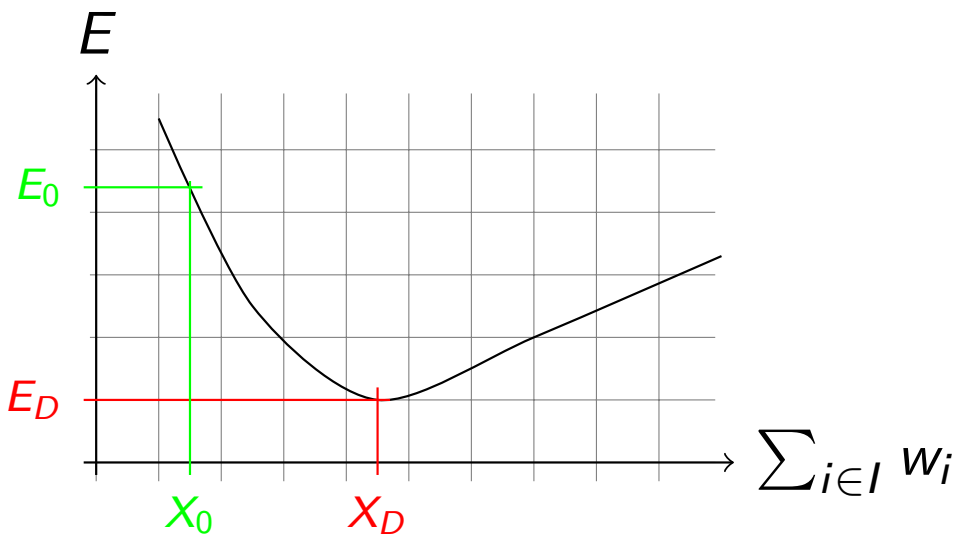
On cherche l'ensemble I des tâches à exécuter deux fois. Ce qu'on sait :



On sait calculer E_D une borne inférieure sur l'énergie, mais on ne sait pas où se situe E_0 le minimum atteignable.

FPTAS pour le problème des chaînes

On cherche l'ensemble I des tâches à exécuter deux fois. Ce qu'on sait :



On sait calculer E_D une borne inférieure sur l'énergie, mais on ne sait pas où se situe E_0 le minimum atteignable.

FPTAS pour le problème des chaînes

Par un algorithme très similaire à la FPTAS pour résoudre SubsetSum, appelé en X_D, on montre qu'on peut trouver un ensemble I_a de tâches satisfaisant

$$\sum_{i \in I_a} w_i = X_a \leq X_0 \leq X_a(1+\varepsilon)$$

FPTAS pour le problème des chaînes

Puis, on montre que

$$E_a \leq (1 + 28\varepsilon)E_0$$

$p \geq 2$

Dans le cas où $p \geq 2$, on a les mêmes résultats de manière similaire.

Tâches indépendantes

Soit n tâches $T_1, \ldots, T_n$, de poids respectifs $w_1, \ldots, w_n$ à exécuter. On suppose qu'il n'y a pas de contraintes entre les tâches (elles doivent toutes finir avant D).

NP-complétude

Ce problème se réduit au problème des chaînes lorsqu'il n'y a qu'un seul processeur. Il est donc NP-dur aussi.

Inapproximation

Théorème

Le problème avec des tâches indépendantes sur p processeurs n'est pas approximable en temps polynomial, sauf si P=NP.

Inapproximation

Théorème

Le problème avec des tâches indépendantes sur p processeurs n'est pas approximable en temps polynomial, sauf si P=NP.

- p processeurs
- $s_{\min} = s_{\mathtt{rel}} = s_{\max} = 1$

Alors, une solution quelconque pour l'énergie est la solution optimale.

Obtenir une approximation de l'énergie revient à résoudre le problème d'ordonnancement pour D.

Relaxation

On va supposer maintenant qu'on relâche la contrainte sur le temps d'exécution.

Relaxation

On va supposer maintenant qu'on relâche la contrainte sur le temps d'exécution.

Un algorithme de (α, β)-approximation est tel que, si E_{opt} est l'énergie optimale pour le problème avec deadline D,

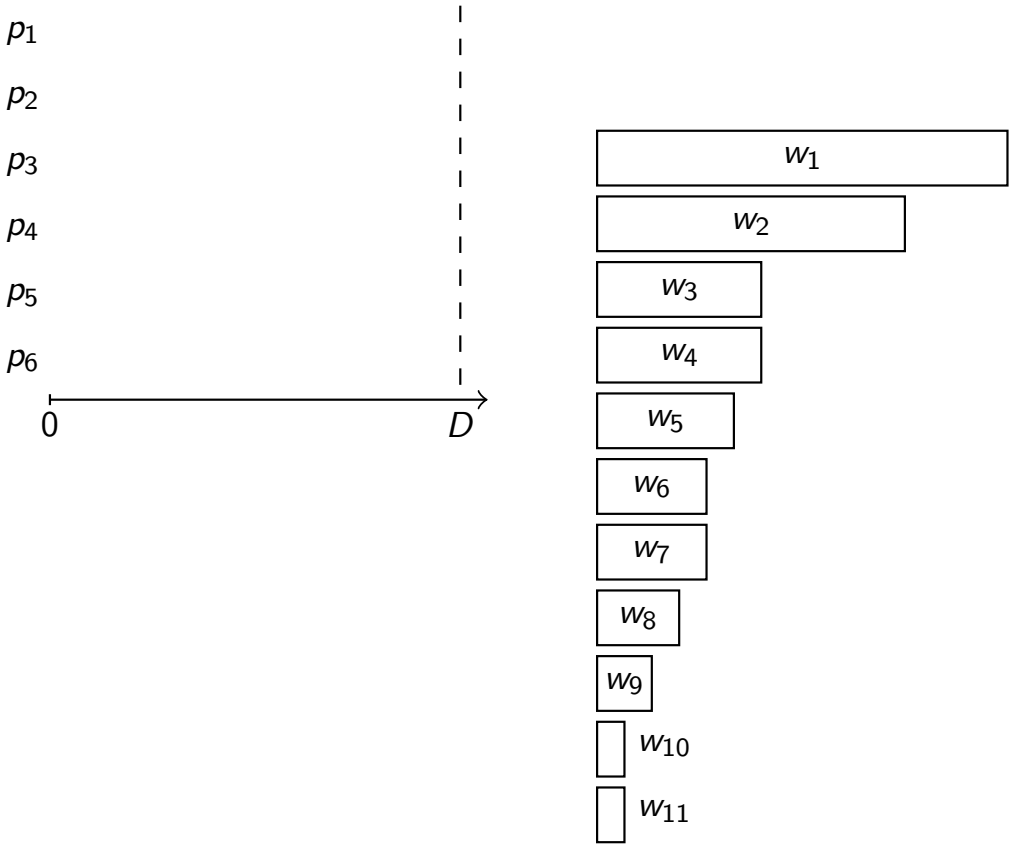
- $E_{algo} \leq \alpha \times E_{opt}$ (E_{algo} l'énergie de l'algo)
- $T_{algo} \leq \beta \times D$ (T_{algo} le temps d'exécution de l'algo)

Relaxation

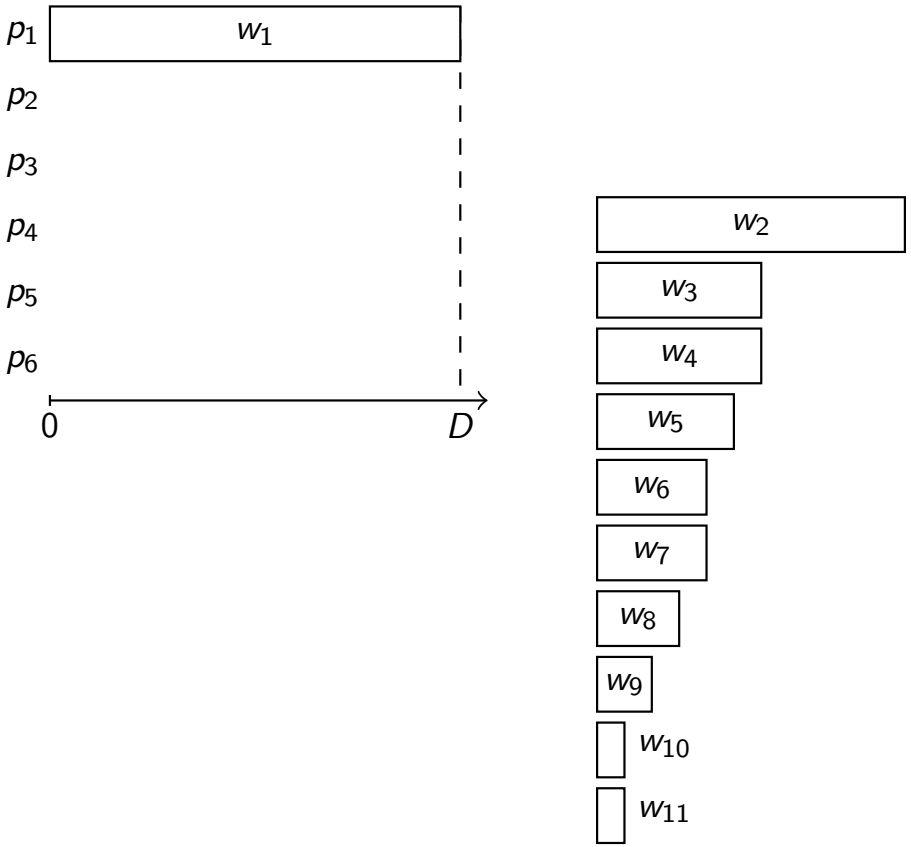
On va supposer maintenant qu'on relâche la contrainte sur le temps d'exécution.

Théorème

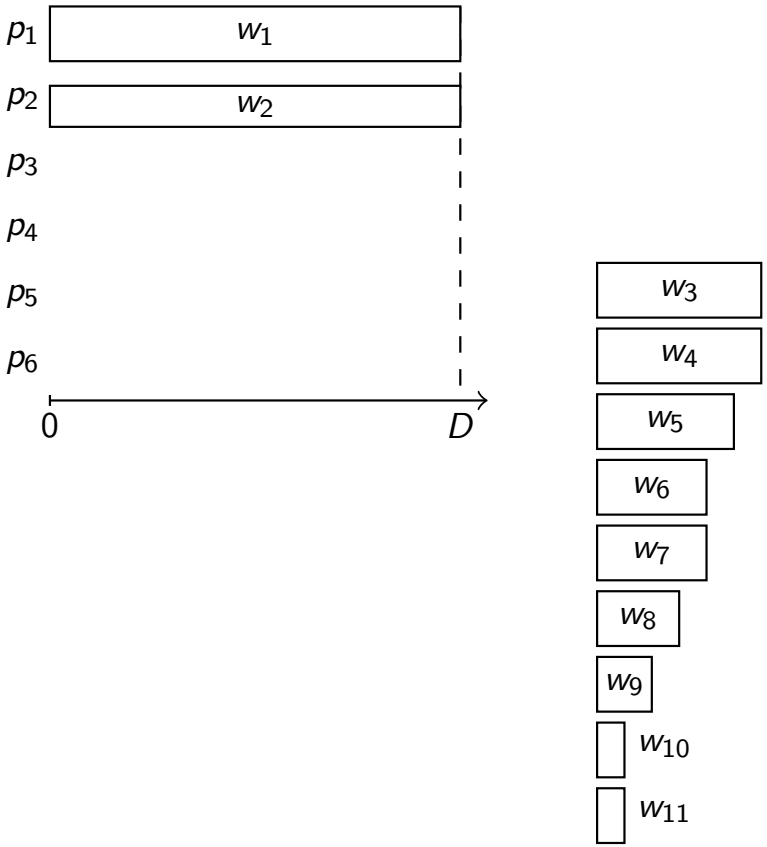
Il existe une $\left(1+\frac{1}{\beta^2},\beta\right)$*-approximation, pour tout* $\beta \geq 2-\Theta(\frac{1}{p})$.



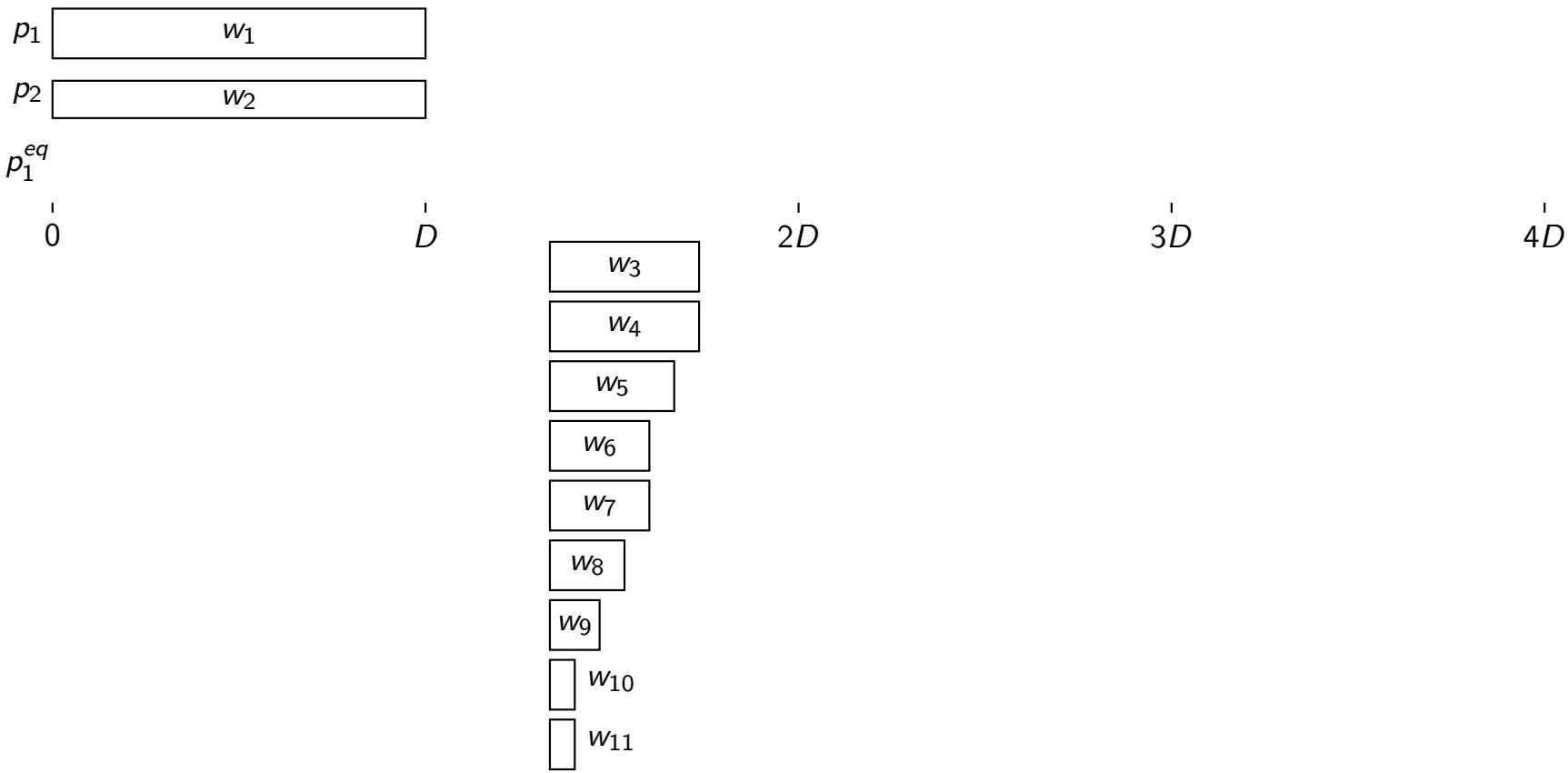
1. Au début les tâches sont triées par ordre croissant.



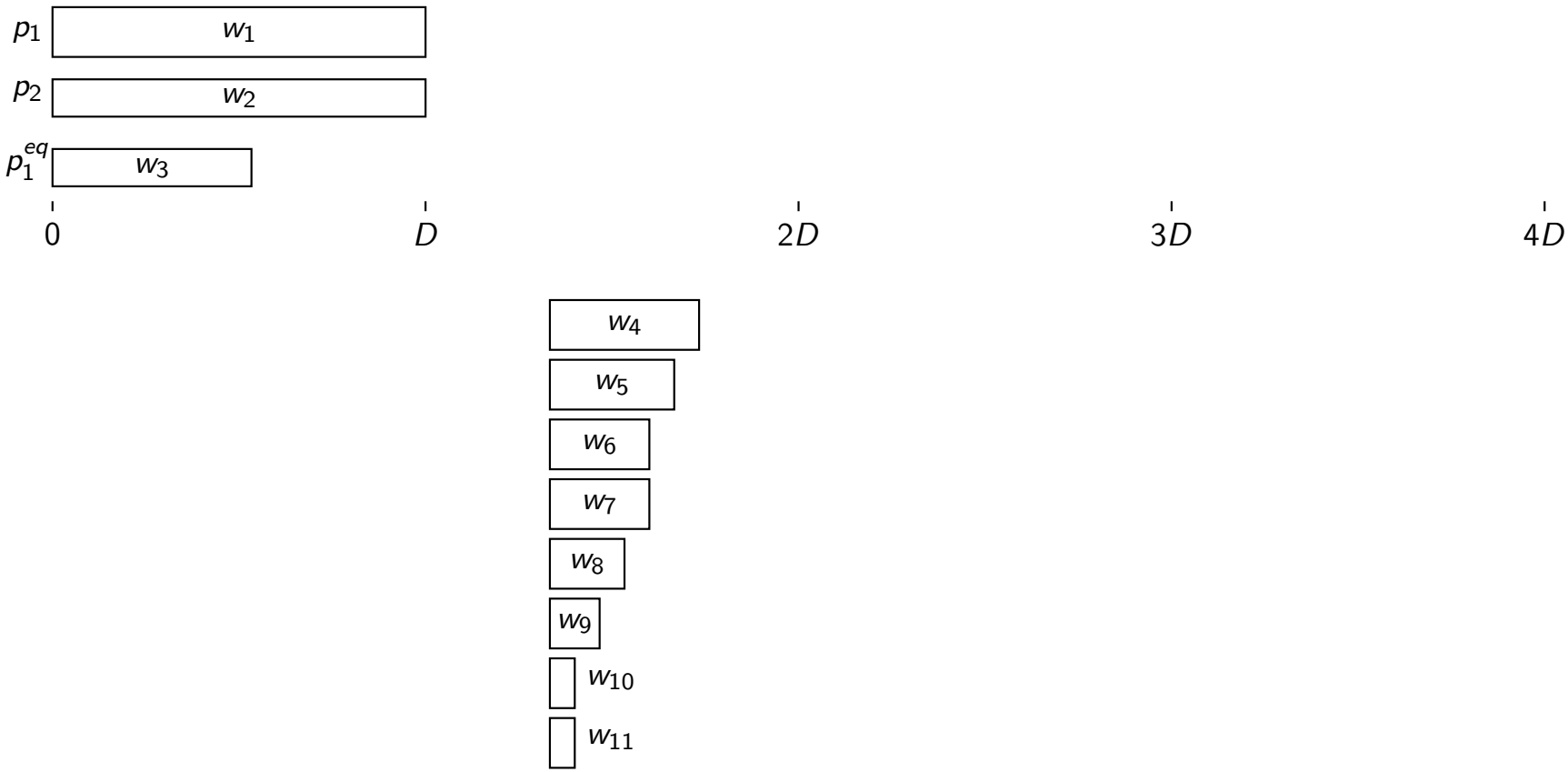
2. Les "grosses" tâches sont ordonnées sur un processeur chacunes



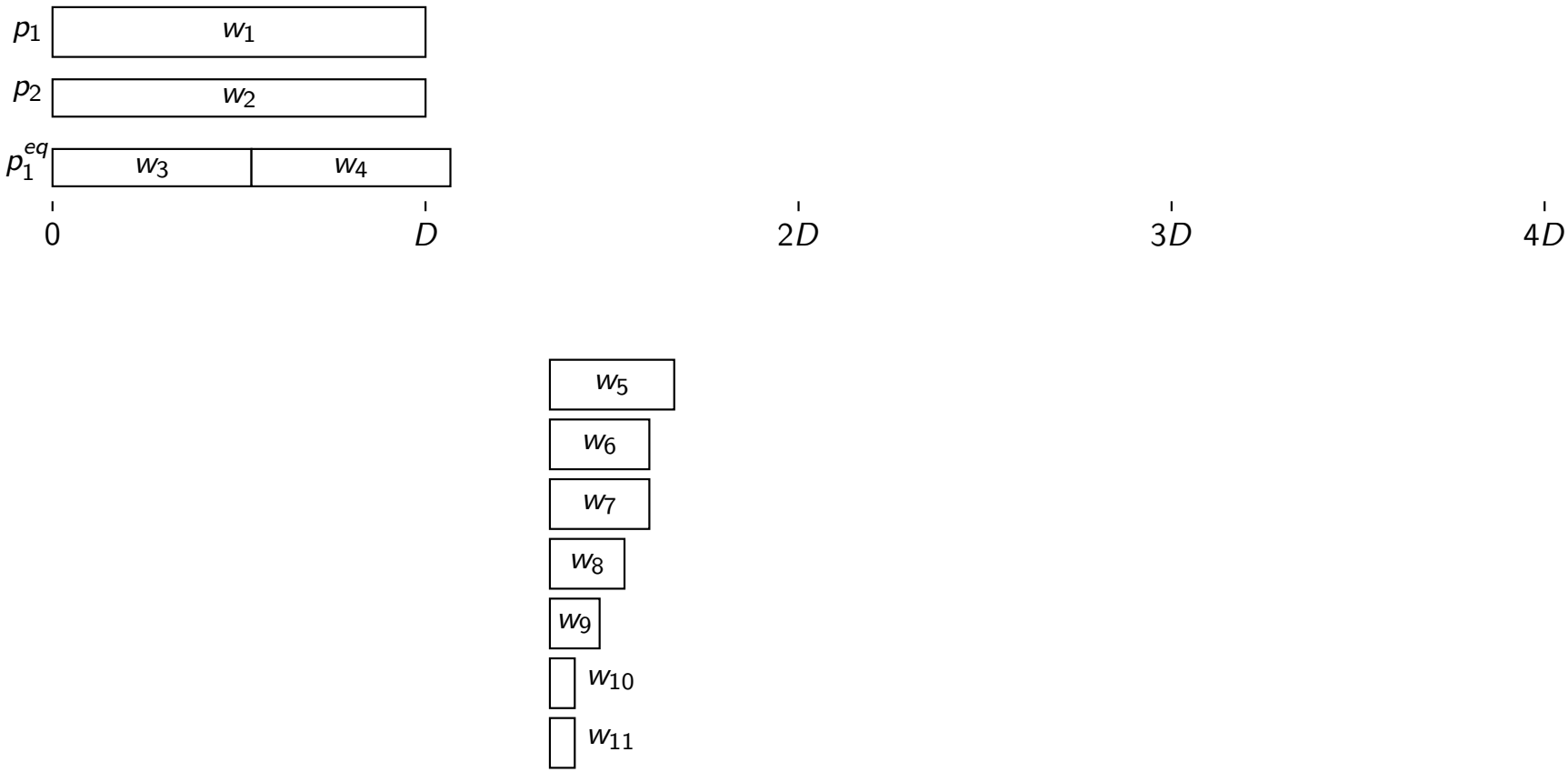
2. Les "grosses" tâches sont ordonnées sur un processeur chacunes



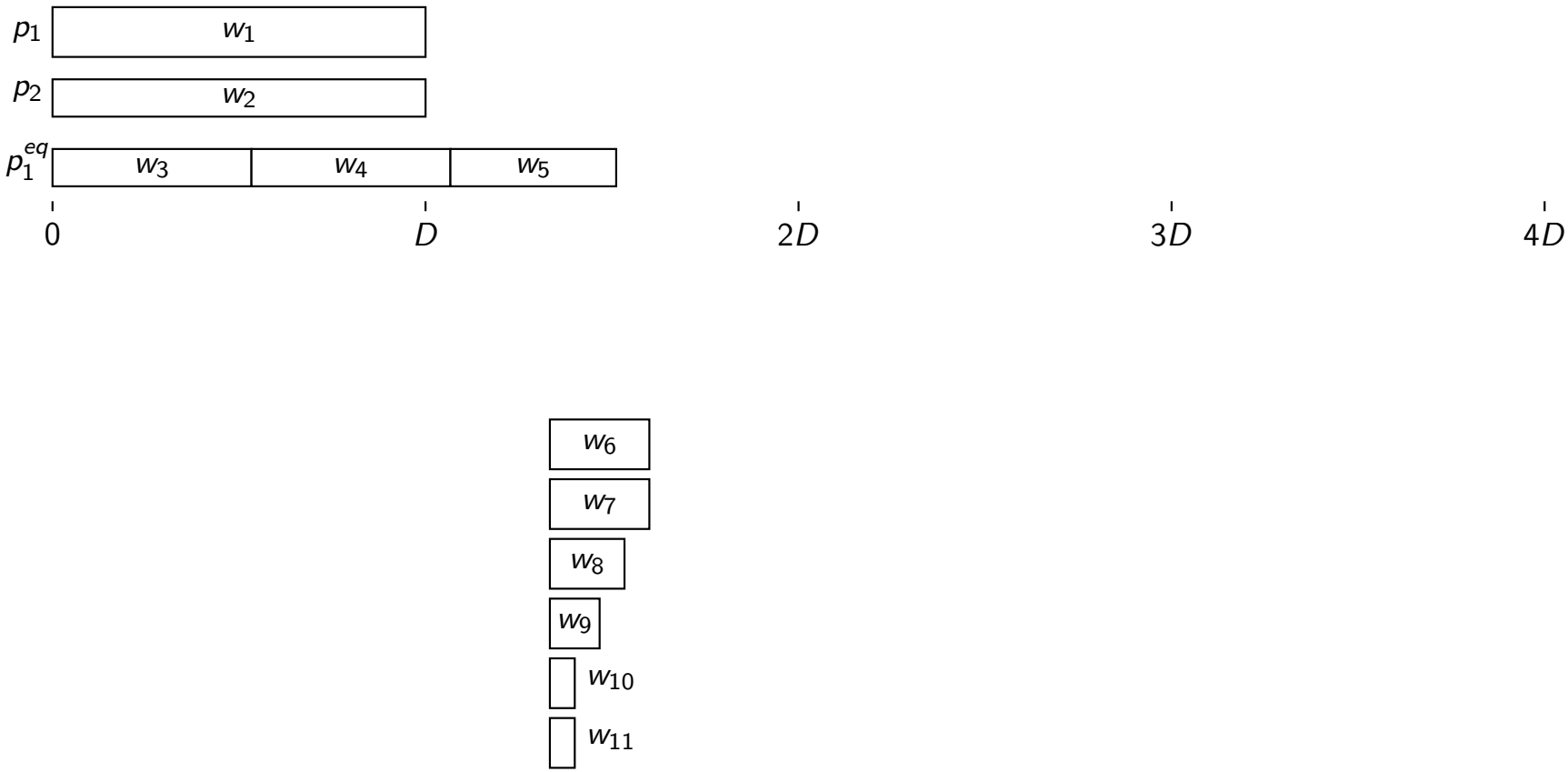
3. On créé un processeur "équivalent" aux processeurs restants.



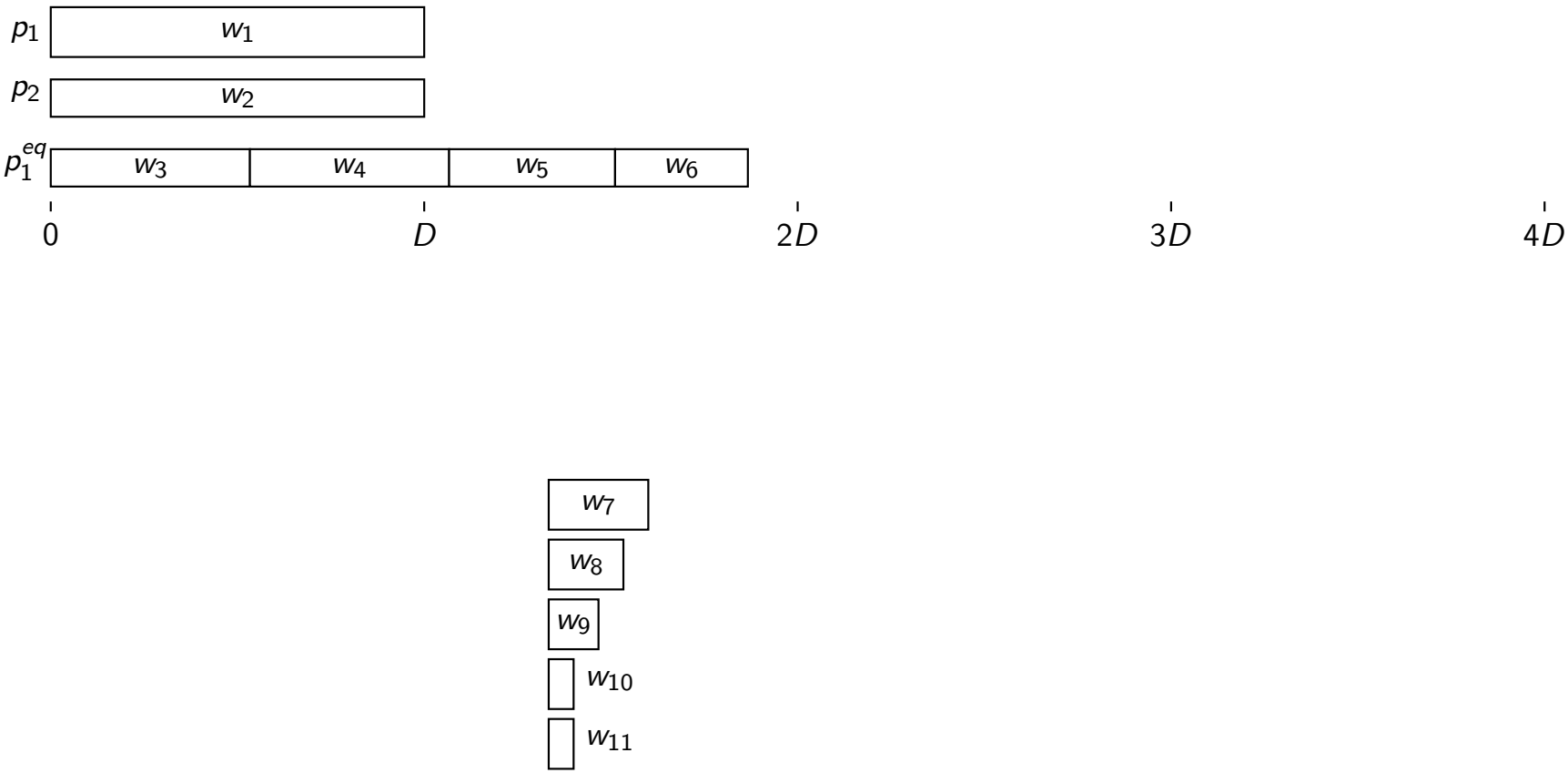
4. On applique la FPTAS des chaînes linéaires sur ce processeur...



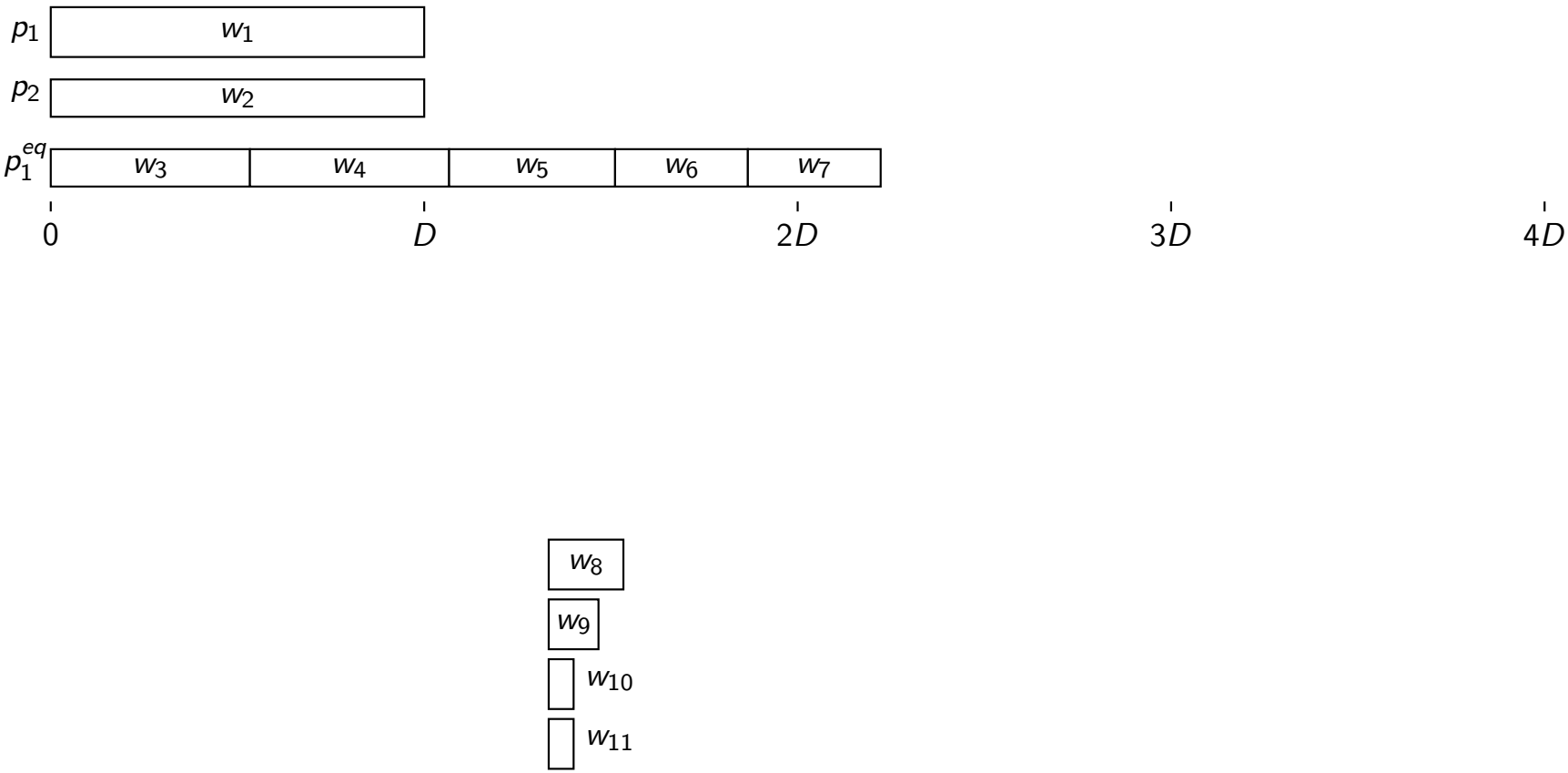
4. On applique la FPTAS des chaînes linéaires sur ce processeur...



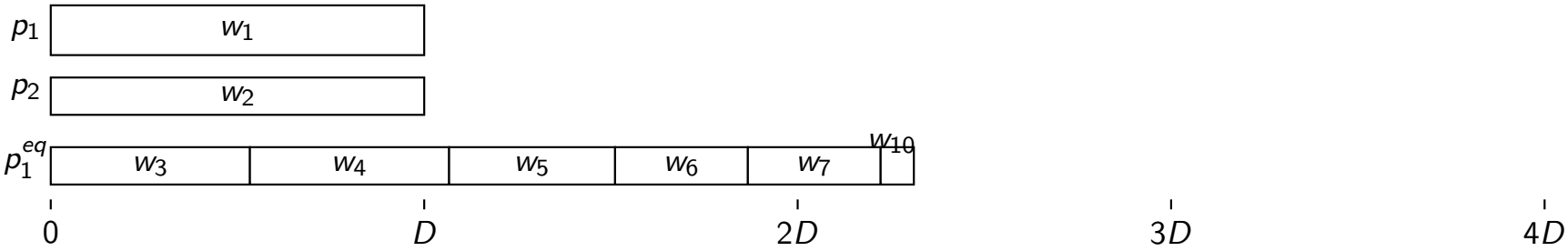
4. On applique la FPTAS des chaînes linéaires sur ce processeur...

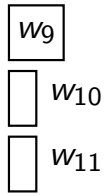


4. On applique la FPTAS des chaînes linéaires sur ce processeur...

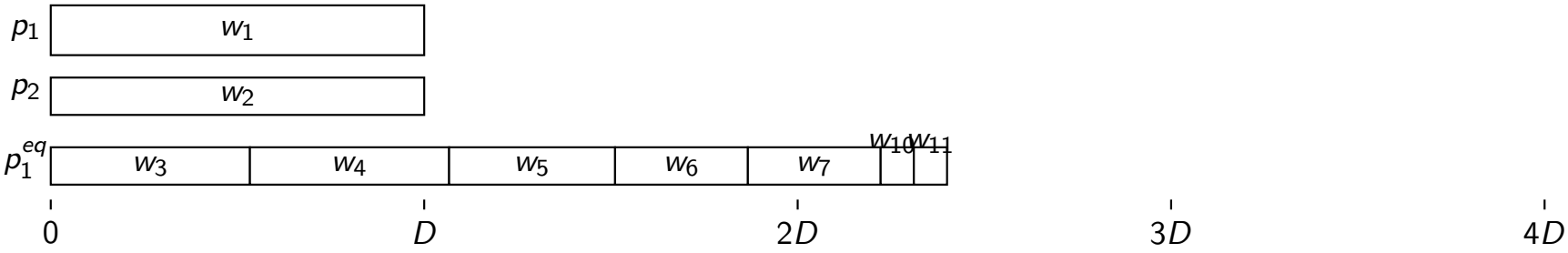


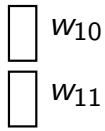
4. On applique la FPTAS des chaînes linéaires sur ce processeur...



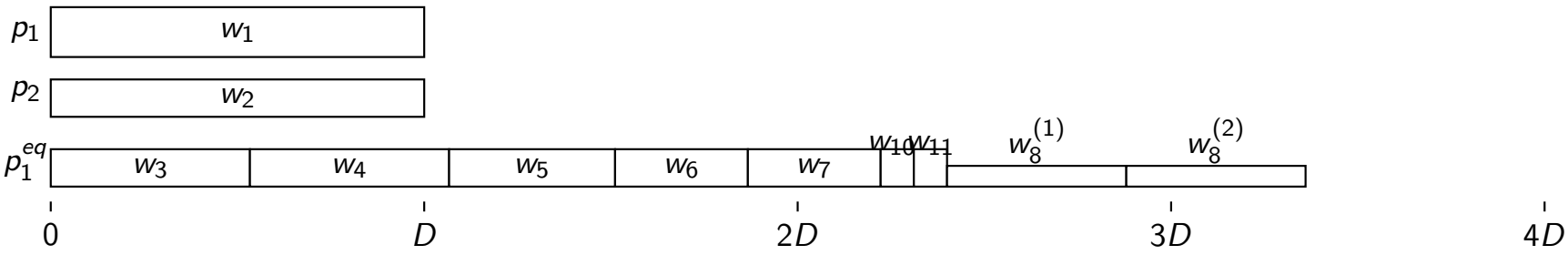


4. On applique la FPTAS des chaînes linéaires sur ce processeur...



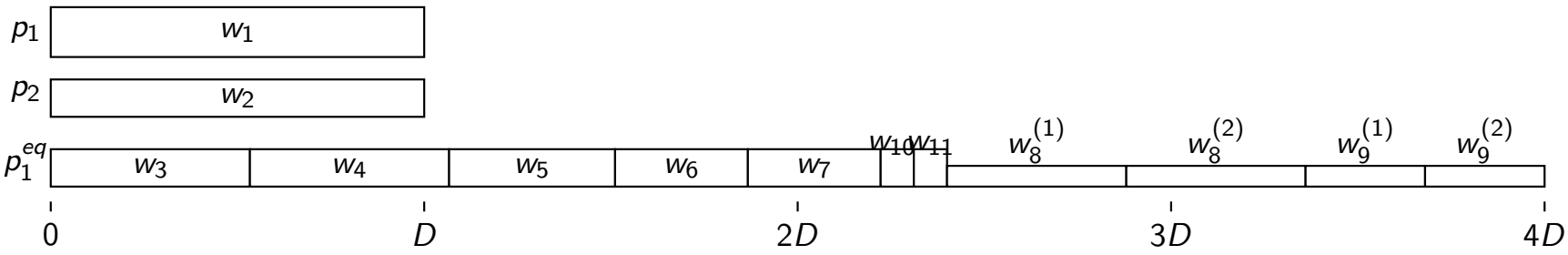


4. On applique la FPTAS des chaînes linéaires sur ce processeur...

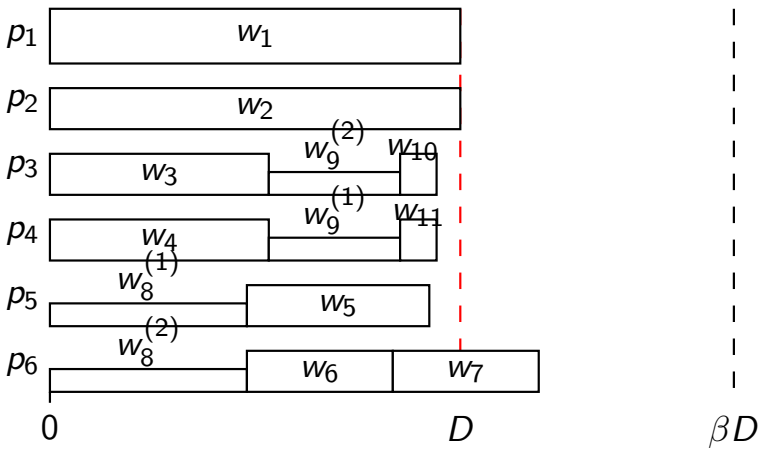


w_{11}

4. On applique la FPTAS des chaînes linéaires sur ce processeur...



4. On applique la FPTAS des chaînes linéaires sur ce processeur...
... ce qui nous donne un "nouvel" ensemble de tâches.



5. Enfin, on ordonne de façon gloutonne les "nouvelles" tâches sur les processeurs restants.

1 Introduction

2 Modèle
Fiabilité
Temps d'exécution
Énergie

3 Approximation pour les chaînes
NP-complétude
FPTAS

4 Approximation pour les tâches indépendantes
Inapproximabilité
Approximation

5 Conclusion

Conclusion

Ce qu'on a fait :

- FPTAS pour les chaînes de tâches ;
- Inapproximabilité pour les tâches indépendantes ;
- Avec une relaxation de la contrainte de temps, on peut avoir une approximation (améliorée pour les grands p en $(1 + \Theta(\frac{1}{p}), 2 - \Theta(\frac{1}{p}))$-approximation).

Conclusion

Dans le futur, il reste encore de nombreux problèmes intéressants :

- Cas des graphes généraux restant à explorer (commencer par des forks, arbres etc) ;
- On peut s'intéresser à d'autres modèles de fiabilité (fiabilité globale, checkpoints...).