# Un exemple d'ordonnancement sous le modèle BSP

Alfredo Goldman* , Grégory Mounié

LMC-IMAG
BP 53, Domaine universitaire
38041 Grenoble Cedex 9
$*$ Financé par le CNPq (Proc $200590/95-2$), détaché du DCC-IME-USP, Brésil

**Résumé**
Nous présentons tout d'abord le modèle *BSP* et le problème de l'ordonnancement de chaînes indépendantes. Puis nous nous attacherons à l'étude de la complexité du problème et à la présentation d'un algorithme approché.

**Mots-clés :** Ordonnancement, chaînes, BSP

## 1. Introduction

De nouveaux modèles d'exécution apparaissent, plus réalistes, car plus proches de la machine. Parmi ces modèles, on peut citer *BSP*, modèle relativement simple qui permet de prédire les performances d'une application. Il devient plus populaire au fur et à mesure que les outils de programmation deviennent disponibles.

Classiquement, l'ordonnancement est au coeur du processus de parallélisation d'une application. Il faut décider du lieu et de la date d'exécution des différentes tâches qui composent l'application.

Cet article relate une étude de cas pour un type de graphe de précédence particulier, les chaînes de tâches.

## 2. Préliminaires

### 2.1. Le modèle *BSP*

L'idée du modèle *BSP* est de séparer, dans le temps, les calculs et les communications. L'exécution du programme parallèle est constituée de «*supersteps*». Un *superstep* est la succession de deux phases. La première est une phase de calcul, où chaque processeur exécute un certain nombre de tâches et prépare les envois et réceptions des données qui seront utilisées lors du prochain *superstep*. La deuxième phase est composée de l'accomplissement des communications et d'une barrière de synchronisation garantissant que toutes les communications ont été achevées.

Dans ce papier, nous considérerons que le nombre de processeurs ($m$) est fixé et que la phase de communication et synchronisation dure un temps constant pour chaque *superstep*. Cette durée sera notée $C$. Cette hypothèse est réaliste si le nombre et le volume des communications impliquant chaque processeur est borné. Le volume des communications dépend de l'application mais dans les algorithmes présentés ici, chaque processeur n'émet et ne reçoit qu'un message à la fois (1-relation).

### 2.2. Le problème des chaînes

L'instance est constituée par des ensembles de tâches unitaires reliées de manière séquentielle, c'est-à-dire un ensemble de chaînes de tâches. Soit $N$ le nombre de chaînes. Notons $ch_i$ la taille de la $i$-ème chaîne. $ch_{max}$ est la longueur de la plus grande chaîne.

Le problème est de les ordonnancer de façon optimale (minimale) en temps. Il existe une borne inférieure simple de la durée de l'ordonnancement optimal : le maximum entre la taille de la plus grande chaîne et le rapport entre le nombre total de tâches et le nombre de processeurs :

$$BI = max(ch_{max}, \frac{\sum_{i=1}^{N} ch_i}{m}) \leq \text{temps de tout ordonnancement valide.}$$

Nous allons supposer maintenant que toute chaîne de taille $BI$ est ordonnancée seule, sur un processeur et qu'il faut résoudre le problème concernant les chaînes de tailles inférieures à $BI$ sur les processeurs encore libres.

### 2.3. État de l'art

Le modèle *BSP* est relativement récent [8]. Il ne s'est pas pour l'instant imposé comme modèle pratique à cause de la vision restrictive de l'algorithmique qu'il impose par rapport à une programmation de type *passage de message*. Le modèle s'est affiné et diversifié [6]. Maintenant il est possible de programmer en *BSP* en utilisant, par exemple, la bibliothèque d'Oxford [4].

L'ordonnancement des chaînes est un problème assez classique [5] et assez ancien [7], qui modélise la distribution de travaux séquentiels de tailles différentes entre plusieurs unités de production. Un problème proche avec préemption processeurs uniformes et temps de communication nul, a été résolu dans [3].

Parmi les récents développements du problème d'ordonnancement de chaînes de tâches on peut citer les tâches multi-processeurs [1].

## 3. Complexité

Nous allons donner les grandes lignes de la preuve que l'ordonnancement optimal des chaînes dans le modèle *BSP* est un problème NP-Difficile.

**Instance** : Un ensemble de chaînes $Ch_i$ de taille $ch_i$.
**Problème** : l'ordonnancement optimal en temps de ces chaînes sous *BSP* avec un temps de communication $C > 0$

Parmi les instances du problème, il existe des cas où l'ordonnancement optimal se fait sans communication. L'ajout d'une communication augmenterait inutilement le travail total de $mC$. Pour deux processeurs ($m = 2$), notre problème se réduit alors au partage de ces chaînes en deux sous-ensembles de tailles égales. Ce problème est connu sous le nom de *Partition*, et il est NP-Complet [2].

Vouloir ordonnancer toutes les instances en n'autorisant qu'un surcoût fixé est aussi un problème NP-Complet. Cela se démontre en choisissant des instances où les tailles des chaînes sont des multiples d'un facteur plus grand que le surcoût. Trouver une solution à distance fixée de l'optimal revient alors à trouver l'optimal.

Par soucis de simplicité, nous avons utilisé une réduction de *Partition*, problème pseudo-polynomial, mais le problème avec plus de deux processeurs peut être réduit de *Numerical 3 DM*, problème NP-Complet au sens fort [2].

## 4. Heuristiques

Les algorithmes heuristiques que nous allons présenter calculent des ordonnancements dont la compétitivité est garantie par rapport à la borne inférieure $BI$ à un facteur additionnel près, qui dépend du nombre, borné ou fixé, de communications utilisées par l'algorithme. Lorsque les chaînes ont une longueur voisine de $BI$, ce facteur dépend alors aussi de la longueur de la plus grande chaîne.

L'idée est d'utiliser un algorithme optimal dans un autre modèle.

### 4.1. Modèle Asynchrone

Dans ce modèle, on considère que toutes les communications peuvent avoir lieu de manière asynchrone, qu'elles sont recouvrables par du calcul et de durée unitaire.

Le placement optimal des chaînes sur processeurs identiques est relativement simple. Il suffit de placer les chaînes l'une après l'autre en utilisant les processeurs l'un après l'autre, jusqu'à $BI$. Lorsqu'un processeur ne peut recevoir l'intégralité d'une chaîne, de taille strictement inférieure à $BI$ (cf section 2.2), il suffit de placer la fin de la chaîne sur ce processeur pour l'occuper jusqu'à $BI$ et de placer le début de la chaîne sur le processeur suivant. Comme cette chaîne est plus petite que $BI$, il y a au moins une unité de temps entre le début et la fin de la chaîne qui peut être utilisée pour une communication asynchrone.

Chaque processeur initie et reçoit au plus une communication, puisque seules la première et la dernière chaîne placées sur ce processeur peuvent être coupées. Il y a au plus $m-1$ communications.

### 4.2. Algorithme à nombre de communications non fixé

L'algorithme asynchrone peut être utilisé afin d'ordonnancer les chaînes en utilisant $m-1$ communications. Il suffit pour cela de remplacer chaque communication asynchrone par une communication *BSP*.

Donc on peut aisément ordonnancer les chaînes en

$$temps = BI + (m-1)C$$

Si l'on trie et place les chaînes par tailles décroissantes, on peut montrer qu'il est toujours possible de grouper les communications deux à deux, pour presque toutes les communications sauf deux.

On ordonnance donc en

$$temps = BI + \left\lceil \frac{m+1}{2} \right\rceil C$$

### 4.3. Algorithme à nombre de communications fixé

Les communications $BSP$ sont assez chères et ne sont pas recouvertes par du calcul. Il est donc parfois souhaitable de minimiser ou fixer leur nombre. D'un autre coté, on peut aussi vouloir fixer l'écart minimum entre deux communications.

Pour cet algorithme, on fixe le nombre de communications $(n)$ et on va placer ces $n$ communications à intervalles réguliers. On les place toutes les $I_n = \lceil \frac{2}{2n+1} ch_{max} \rceil$ unités de temps.

1. Pour toute chaîne $j$ telle que $ch_j \leq \frac{2n-1}{2} I_n$, il y a au moins $I_n$ unités de temps entre le début et la fin de chaque chaîne coupée. Donc il y a toujours une communication qui est placée dans l'intervalle. On ordonnance les chaînes en :
$$temps = BI + nC$$

2. si $ch_j > \frac{2n-1}{2} I_n$, on va garantir un écart de taille $I_n$ entre le début et la fin de la chaîne de la manière suivante. On a $ch_j \leq ch_{max}$. On peut donner une borne au pire du comportement de l'algorithme en fonction de $ch_{max}$. Soit $\Delta_{ch_{max}} = ch_{max} - \frac{2n-1}{2} I_n < I_n$. Soit $\alpha(n, ch_{max}) = I_n - \Delta_{ch_{max}}$. Supposons que l'on vient de placer la chaîne $ch_p$ et que l'on veut placer la chaîne $ch_s$. Il y a problème lorsque $ch_s$ ne peut pas être placée parce que $ch_p$ utilise une portion un peu trop importante de l'intervalle où elle finit. En fonction de la taille de $ch_s$ et de la place occupée par $ch_p$, on va déplacer des tâches de $ch_s$ afin que le début de $ch_s$ finisse avant la communication du début de l'intervalle, ou retarder la fin de $ch_s$ afin qu'elle ait lieu après la fin de l'intervalle. On va choisir la solution qui retarde le moins $ch_s$. Dans les deux cas $ch_s$ termine au plus $\frac{\alpha(n, ch_{max})}{2}$ unités de temps plus tard.

   Comme on n'a qu'une réception par processeur, on ordonnance au pire en :
$$temps = BI + \frac{\alpha(n, ch_{max})}{2} + nC$$

.

### 4.4. Fixer un intervalle minimum entre deux communications

Soit $\lambda$ le temps minimum autorisé entre deux communications. Ce facteur peut être défini par des contraintes du réseau de communication de la machine ou par la volonté du programmeur de garantir le rapport entre calculs et communications.

Deux solutions peuvent être utilisées :

1. on peut décider de ne pas construire d'ordonnancement ne respectant pas la contrainte, par exemple en choisissant un nombre de communications tel que $I_n > \lambda$ ;

2. on peut appliquer un algorithme qui va supprimer ou déplacer les communications qui sont trop proches les unes des autres.

   On parcourt l'ordonnancement initial et l'ensemble des communications dans l'ordre afin de les replacer. Soit $d_s$ la date de la communication que l'on replace. Soit $d_p$ la date de la dernière communication que l'on a déjà replacée. Soit $E_s$ le processeur qui émet la communication. Soit $R_s$, le processeur qui reçoit cette communication. Si il y a plusieurs paires émetteurs-récepteurs utilisant la même communication, on les considérera comme indépendantes, utilisant chacunes leurs propres communications.

Trois cas sont possibles :

(a) $d_s - d_p \geq \lambda$ : on passe alors à la communication suivante,

(b) $\frac{\lambda}{2} \leq d_s - d_p < \lambda$ : on retarde la communication jusqu'à la date $d_p + \lambda$. Éventuellement on groupe cette communication avec celles qui sont déjà placées à cette date. Cette modification dans la date de la communication peut retarder l'exécution des chaînes sur $R_s$. Ce retard est au plus de $\frac{\lambda}{2}$ unités de temps.

(c) $d_s - d_p < \frac{\lambda}{2}$ : on avance la communication à la date $d_p$ et on la groupe avec la communication précédente. Les tâches qui s'exécutaient sur $E_s$ entre $d_p$ et $d_s$ sont déplacées sur $R_s$. Cela augmente la charge de calcul de $R_s$. Cette augmentation est au plus de $\frac{\lambda}{2}$ tâches.

En utilisant l'algorithme de la section 4.2 et l'algorithme que nous venons de présenter, il est possible d'ordonnancer les chaînes en :

$$temps = BI + \frac{\lambda}{2} + \left\lceil \frac{m+1}{2} \right\rceil C$$

## 5. Conclusion

Nous avons présenté un petit exemple montrant que le modèle *BSP* impose des restrictions par rapport aux modèles asynchrones. Un problème simple dans un modèle asynchrone devient NP-Complet dans BSP.

Pour le problème d'ordonnancement de chaînes indépendantes, il est possible d'extraire des algorithmes qui sont proches de l'optimal à un facteur additionnel près. Ce facteur est borné pour un nombre de communications qui est une fonction linéaire du nombre de processeurs, y compris si l'on décide d'imposer un délai minimal entre deux communications. Si on limite le nombre de communications, il est encore possible d'obtenir un petit facteur additionnel.

Cet article est extrait d'une étude beaucoup plus complète, réalisée avec le professeur Denis Trystram.

## Bibliographie


1. Blazewicz (Jacek) et Liu (Zhen). – Scheduling multiprocessor tasks with chain constraints. *European Journal of Operational Research*, vol. 94, 1996, pp. 231–241.
2. Garey (M. R.) et Johnson (D. S.). – *Computers and Intractability (A Guide to Theory of NP-Completeness)*. – New York, Freeman, 1979, 340p.
3. Gonzales (Teofilo) et Sahni (Sartaj). – Premptive scheduling of uniform processor systems. *Journal of the ACM*, vol. 25, n1, January 1978, pp. 92–101.
4. Hill (Jonathan M. D.), McColl (Bill), Stefanescu (Dan C.), Goudreau (Mark W.), Lang (Kevin), Rao (Satish B.), Suel (Torsten), Tsantilas (Thanasis) et Bisseling (Rob). – *BSPlib: The BSP Programming Library*. – Technical report n PRG-TR-29-9, Oxford University Computing Laboratory, May 1997. url: http://www.BSP-Worldwide.org.
5. Hoogeven (J. A.), Lenstra (J. K.) et Veltman (B.). – Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, no16, 1994, pp. 129–137.
6. McColl (W F). – Scalable computing. *In : Computer Science Today: Recent Trends and Developments. LNCS Volume 1000*, éd. par van Leeuwen (J). pp. 46–61. – Springer-Verlag.
7. McNaughton (R.). – Scheduling with deadlines and loss functions. *Management Science*, no6, 1959, pp. 1–12.
8. Valiant (Leslie G.). – A bridging model for parallel computation. *Communications of the ACM*, vol. 33, n8, august 1990, pp. 103–11.