Adaptation d'un algorithme optimal d'ordonnancement en régime permanent pour des lots bornés

Sékou Diakité, Jean-Marc Nicod, Laurent Philippe

RenPar'18
Fribourg, 11 au 13 février 2008

Adaptation d'un algorithme optimal d'ordonnancement en régime permanent pour des lots bornés

Sékou Diakité, Jean-Marc Nicod, Laurent Philippe

LIFC/ INRIA GRAAL, 16 route de Gray 25000 Besançon, France,
[diakite,nicod,philippe]@lifc.univ-fcomte.fr

Résumé
Le contexte de cet article est l'ordonnancement de lots bornés de travaux identiques sur une plate-forme d'exécution hétérogène comme la grille. Les travaux exécutés sont des graphes de tâches orientés et sans cycle (DAG), en forme d'anti-arbre. Les tâches sont de plusieurs types et les nœuds de la plate-forme ne sont pas toujours en mesure d'exécuter tous les types de tâches. Le problème de minimisation du temps d'exécution d'un lot est un problème NP-Complet. Sous l'angle du régime permanent, il est possible de décrire le problème sous la forme d'un programme linéaire donnant une solution optimale pour l'ordonnancement cyclique de lots infinis. Lorsque les lots sont bornés, les résultats restent bons bien que sous optimaux. Nous montrons ici que les phases d'initialisation et de terminaison ajoutent un sur-coût qui pénalise le temps global d'exécution. Nous montrons ensuite le lien entre la taille de ces phases et la taille de la période de l'ordonnancement cyclique et donnons un algorithme permettant le calcul de la période minimale. Des expérimentations, obtenues par simulations avec SimGrid, illustrent en fin d'article le gain apporté par le choix d'une période minimale.

Mots-clés : ordonnancement hétérogène, grille de calcul, régime permanent, lots bornés.

1. Introduction

Dans cet article, nous nous intéressons à l'ordonnancement d'un lot de travaux identiques sur une plate-forme d'exécution hétérogène. Chacun des travaux est une application exécutée indépendamment des autres, si possible en parallèle. Cette application est décrite par un graphe de tâches orienté, acyclique (DAG) et sans fourche (*intree*), dans lequel chaque nœud est une tâche et chaque arête est une dépendance. La plate-forme d'exécution est une grille de calcul formée de machines interconnectées par des liens réseaux. L'originalité de nos travaux tient au fait que chaque nœud n'est capable d'exécuter qu'un sous-ensemble des tâches, et plusieurs machines peuvent exécuter un même type de tâche. Un exemple de description du contexte est une application de type *workflow* traitant un ensemble de données en plusieurs étapes – par exemple, chaque étape peut être un filtre appliqué à une image – sur la grille où les bibliothèques utilisées ne sont pas installées sur toutes les machines – seuls certains filtres sont disponibles sur une machine donnée. Notre objectif est de minimiser la durée d'exécution d'un lot de travaux (*makespan*), or il n'existe pas de solution optimale calculable en temps raisonnable à ce problème. Ce faisant, nous nous plaçons dans le cas particulier du régime permanent. Notre problème peut alors s'exprimer sous la forme d'un ensemble de contraintes imposées par les travaux à exécuter et par les propriétés de la plate-forme. Il s'agit d'un programme linéaire dont la résolution conduit à une solution optimale, le régime permanent permettant de transformer ce problème d'optimisation du *makespan* en un problème d'ordonnancement cyclique [1]. Le résultat de ce programme linéaire permet de connaître le taux d'occupation des ressources par unité de temps (u.t.), qu'un algorithme de reconstruction permet de respecter. Nous obtenons par conséquent un ordonnancement cyclique optimal pour un lot infini de travaux.

En pratique, une phase d'initialisation, avant la phase d'ordonnancement périodique, et une phase de terminaison, après, sont nécessaires à l'exécution d'un lot de travaux. L'initialisation consiste à calculer suffisamment de tâches pour que la première période puisse s'exécuter. La phase de terminaison permet d'achever l'exécution planifiée. Ces deux phases sont décrites plus précisément dans la suite de

cet article. Notre contribution à ce domaine est d'envisager d'ordonnancer des lots de travaux de taille bornée suivant ce schéma. Ici, les phases d'initialisation et de terminaison ne peuvent plus être négligées [5]. Nous présentons alors comment optimiser la mise en œuvre d'un ordonnancement respectant les résultats du programme linéaire, par la réduction optimale de la taille de la période. Ce résultat permet de s'adapter à des séries de travaux de taille bornée et de réduire les phases d'initialisation et de terminaison. Des résultats expérimentaux confirment l'influence de la taille de la période sur le temps d'exécution d'un lot.

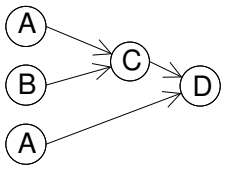
Cet article s'articule de la manière suivante. Nous introduisons d'abord le contexte de nos travaux, puis, dans la partie 3, nous les positionnons par rapport au domaine. Dans la partie suivante, nous rappelons comment un ordonnancement en régime permanent est habituellement mis en œuvre. Nous exposons ensuite notre contribution par la présentation des algorithmes utilisés pour déterminer la période optimale de cet ordonnancement cyclique. Les résultats présentés dans la partie 5 mettent en évidence l'importance de la valeur de cette période quant au temps d'exécution d'un lot de travaux de taille bornée. En fin d'article, une conclusion et des perspectives à ce travail sont données.

2. Contexte

La grille de calcul est composée de processeurs (ou hôtes) qui communiquent entre eux par des liens réseaux. Une grille G est représentée par un graphe non orienté : $G = (P, L)$. Les sommets de P sont les n processeurs p_i de la grille. Les arêtes L sont les m liens réseaux l_h entre les processeurs.

Cette grille traite un lot B (ou *Batch*) de $k = |B|$ travaux j identiques. Le lot B est donc défini comme l'ensemble des k instances j_l du travail j. Le travail $j = (T, D)$ est un graphe orienté acyclique ou DAG formé des tâches t_i de T et des dépendances D, ensemble des arcs entre les tâches.

Toutes les tâches de j ne sont pas nécessairement différentes. Dans l'exemple donné à la figure 1a, la tâche A est présente deux fois. Nous parlons alors d'une tâche de *type* A. Or, comme les bibliothèques utiles à l'exécution de toutes les tâches ne sont pas présentes sur toute la plate-forme, une tâche ne peut s'exécuter que sur un sous-ensemble de processeurs. Les conditions d'exécution de chaque type de tâche sont données par la matrice des temps d'exécution. Dans l'exemple de la figure 1b, une tâche de type A s'exécute en 20 u.t. sur p_1 et en 15 u.t. sur p_4.



(a) Graphe de tâches

		p_1	p_2	p_3	p_4
Type	A	20	∞	∞	15
	B	10	10	∞	∞
	C	∞	10	10	∞
	D	∞	∞	10	10

(b) Matrice des temps d'exécution

FIGURE 1: Graphe de tâche et plate-forme.

Le problème d'ordonnancement énoncé ci-dessus peut s'exprimer sous la forme $\alpha|\beta|\gamma$ [6] suivante : $U_r|$batch of intree$|C_{max}$. Ce problème est connu pour être NP-Complet [1, 12]. Il faut alors trouver soit des heuristiques adaptées, soit un problème proche pour lequel une solution est calculable en temps polynomial.

3. État de l'art

Les techniques d'ordonnancement en régime permanent permettent d'atteindre une utilisation optimale des ressources pour des lots infinis de travaux identiques [1]. L'ordonnancement des travaux est composé d'une phase d'initialisation (sous optimale) suivie de la phase de régime permanent (optimale). La phase de terminaison (sous optimale) permet d'achever l'exécution des travaux en cours. Le temps d'exécution du lot (*makespan*) tend vers l'optimal lorsque sa taille augmente car les phases intiale et terminale ont des coûts fixes. Elles ne dépendent que du type de graphe de tâches et de la taille de période. Par contre, pour l'ordonnancement d'un lot de taille bornée, ces coûts ne peuvent plus être négligés.

Les solutions classiques pour minimiser le *makespan* de l'exécution d'un ensemble de tâches, dépendantes ou pas, font appel à des heuristiques telles que la terminaison au plus tôt, *Earliest Finish Time* [15], ou le chemin critique, *Critical Path* [10]. Ces algorithmes pré-calculent l'ordonnancement en tenant compte de toutes les tâches. Il n'y a donc pas ici de problèmes d'initialisation ou de terminaison. Une classification des algorithmes d'ordonnancement de DAG statique a été faite par Kwok et Ahmad [11].

Hanen et Munier donnent une heuristique garantie à un facteur 4/3 de l'optimal dans le cas de ressources bornées avec communications et graphes à gros grain [7, 12].
Les solutions d'ordonnancement à la volée calculent l'ordonnancement pendant que les tâches s'exécutent. Dès qu'une machine est disponible, les tâches libres sont ordonnancées. Ces techniques faciles à implémenter donnent de bons résultats. Ici encore, cette solution peut s'appliquer à notre contexte, même si elle ne tire pas parti du fait que les travaux sont des instances d'un même graphe de tâches.
Des techniques autres que le régime permanent s'intéressent à l'ordonnancement de travaux multiples. Elles traitent pour la plupart d'ordonnancement temps réel de tâches périodiques [13, 14]. Dans l'ordonnancement temps réel, les tâches sont liées à une date butoir ou *deadline*. L'objectif est alors de maximiser le nombre de tâches qui respectent cette contrainte, ce qui n'est pas compatible avec notre problème de minimisation du temps global d'exécution d'un lot. Iverson et Özgüner [8] utilisent une technique différente où chaque graphe de tâches est en compétition avec les autres pour les ressources d'exécution de la grille. Ici, l'ordonnanceur a une connaissance limitée des processeurs de la grille et aucune information sur les prochains travaux à exécuter. Ces travaux ne sont pas adaptables à notre contexte où une entité centrale ordonnance toutes les instances des lots.
Dans une précédente étude [5], nous comparons trois solutions d'ordonnancement de notre problème. La première est un algorithme de calcul en amont utilisant la méta-heuristique génétique : GATS (*Genetic Algorithm for Task Scheduling*) [4]. Elle est basée sur un ordonnancement de liste que nous avons adapté au traitement par lots. La seconde solution est un algorithme d'ordonnancement à la volée et la troisième est l'ordonnancement en régime permanent. Cette étude met en évidence la supériorité de l'approche de l'ordonnancement cyclique, en régime permanent, pour les lots de grande taille. Pour des tailles réduites, l'approche du GATS se révèle supérieure. En effet, nous avons observé dans certains cas une dégradation importante du temps d'exécution du lot. Nous avons mis en évidence l'importance de la taille de la période dans cette dégradation. En effet, celle-ci influence directement les phases sous optimales d'initialisation et de terminaison. L'ordonnancement en régime permanent, optimal à l'infini, ne conduit alors plus toujours au meilleur ordonnancement. Pour cette raison nous proposons d'améliorer le choix de la valeur de la période en fonction des contraintes liées au problème du régime permanent.

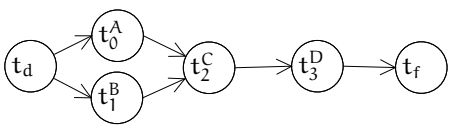
4. Adaptation du régime permanent aux lots bornés

4.1. Ordonnancement en régime permanent

L'optimisation du *makespan* pour l'ordonnancement d'un lot fini de travaux sur une plate-forme hétérogène est NP-Complet. Beaumont et al. [1] transforment le problème en un problème polynomial traitant un lot infini de travaux. Pour un lot borné, cette approche conduit au calcul d'un ordonnancement sous-optimal dont le *makespan* est une valeur approchée du calcul à l'infini : $\mathtt{makespan}_r = \mathtt{makespan}_o + \epsilon$, avec $\mathtt{makespan}_r$ le *makespan* obtenu, $\mathtt{makespan}_o$ le *makespan* optimal et ϵ une constante.

4.1.1. Construction de la période

La première étape de l'ordonnancement en régime permanent est de construire une période contenant un nombre fini de travaux. Cette période est constituée des tâches de chacun des travaux et des communications correspondantes. Le calcul de cette période peut être réalisé par le programme linéaire traduisant les contraintes du problème. Nous présentons ici un rapide survol de cette méthode au cours duquel les temps de communication sont négligés, les temps de communication étant d'un ordre de grandeur inférieur aux temps d'exécution.



(a) Job

Type	P_1	P_2	P_3	P_4
A	20	∞	∞	20
B	10	10	∞	∞
C	∞	10	10	∞
D	∞	∞	10	10

(b) Matrice de coût : c

FIGURE 2: (a) Graphe de tâches j_2, (b) Grille G_0.

Soit un exemple de graphe de tâches et un exemple de plate-forme donnés à la figure 2. L'objectif du programme linéaire est de maximiser $\rho = \sum_{i=1}^{n} cons(p_i, t_f)$, où t_f est la tâche finale de coût 0. Soit $\alpha(p_i, t_k)$ le temps que consacre le processeur p_i à la tâche t_k en une unité de temps (u.t.) et soit $cons(p_i, t_k)$ la quantité de tâches t_k traitées par le processeur p_i par u.t.. Les contraintes du système sont alors les suivantes :

– $\forall p_i \in P, \forall t_k \in T, 0 \leq \alpha(p_i, t_k) \leq 1$: chaque processeur p_i consacre entre 0 et 1 u.t. à chaque tâche t_k.
– $\forall p_i \in P, \forall t_k \in T, \alpha(p_i, t_k) = cons(p_i, t_k) \times c_{i,k}$: définition de $\alpha(p_i, t_k)$.
– $\forall p_i \in P, \sum_{t_k \in T} \alpha(p_i, t_k) \leq 1$: chaque processeur p_i consacre entre 0 et 1 u.t. sur l'ensemble des tâches.

Tâche	P_1	P_2	P_3	P_4
t_0^A	7/200	-	-	9/200
t_1^B	3/100	1/20	-	-
t_2^C	-	1/20	3/100	-
t_3^D	-	-	7/100	1/100

(a) Matrice de consommation : cons

Tâche	P_1	P_2	P_3	P_4
t_0^A	7	-	-	9
t_1^B	6	10	-	-
t_2^C	-	10	6	-
t_3^D	-	-	14	2

(b) Matrice de consommation avec un dénominateur de 200 : consInt

FIGURE 3: (a) Résultats bruts (b) Résultats avec un dénominateur commun de 200.

Les résultats de la résolution du programme linéaire sont donnés par la matrice de consommation cons (fig. 3a) et l'objectif $\rho = 2/25$, soit un débit de 2 tâches toutes les 25 u.t.. Chaque valeur $cons(p_i, t_k)$ définit la quantité de tâches t_k traitées par le processeur p_i par u.t.. Par exemple, le processeur p_2 traite une tâche t_1^B toutes les 20 u.t. dans une période. La longueur de la période est alors calculée comme le *ppcm* (plus petit commun multiple) de tous les dénominateurs de la matrice de consommation. Dans l'exemple de la figure 2, la longueur de la période est de 200 u.t. au cours de laquelle 16 travaux j_2 sont traités.

4.1.2. De la période à l'ordonnancement

La construction de l'ordonnancement en régime permanent se fait par itérations successives. À chaque itération, le graphe est parcouru en profondeur, en affectant chacune des tâches aux processeurs qui ont une valeur non nulle dans consInt (fig. 3b). Pour cela, un poids est calculé comme la consommation de la plus faible des affectations possibles, sur les processeurs éligibles. Cette valeur est alors soustraite de toutes les allocations aux processeurs choisis dans la matrice de consommation consInt. Cette opération s'arrête lorsque consInt est nulle.

À chaque fois qu'une tâche fille est affectée à un processeur différent d'une de ses tâches mères, il est nécessaire de traiter les tâches mères dans l'initialisation afin de préparer l'ordonnancement de la première période. Par exemple, dans consInt, il faudra calculer 2 graphes partiels (t_0^A, t_1^B et t_2^C) dans initialisation pour résoudre les dépendances des deux tâches t_3^D affectées au processeur p_4.

L'ordonnancement consiste alors à calculer tous les graphes partiels nécessaires aux dépendances (initialisation), puis à exécuter les allocations autant de fois que nécessaire (régime permanent) et enfin à terminer les graphes partiels présents dans la grille (terminaison).

4.2. Calcul d'une période optimale

Le paragraphe précédent montre que le régime permanent est l'exécution d'un ordonnancement reproductible (périodique), répété autant de fois que nécessaire. Pendant l'exécution d'une période, les contraintes de dépendances entre les tâches sont ignorées car chaque contrainte est résolue dans la période précédente. L'exécution de la première période nécessite alors la résolution préalable de toutes les dépendances. C'est le rôle de l'initialisation. Après l'exécution de la dernière période, la terminaison va permettre d'achever l'exécution des graphes en cours (prévus pour résoudre les dépendances d'une nouvelle période).

Comme la phase de régime permanent est optimale, améliorer le temps de calcul d'un lot borné passe par l'optimisation des phases sous-optimales d'initialisation et de terminaison. Or le temps d'exécution de ces phases dépend du nombre de tâches à effectuer pendant une période. Ainsi, réduire la période permet de diminuer le temps de préparation et de sortie du régime permanent. Pour cela, nous pro-

posons de réorganiser la période pour réduire sa longueur (et donc le nombre de travaux par période). Cette réorganisation se fait en respectant les contraintes du programme linéaire et le débit obtenu.

4.2.1. Description du problème

L'objectif du programme linéaire est la maximisation du nombre de graphes traités par u.t.. Cet objectif ne permet pas toujours d'obtenir la période la plus courte (inutile dans le cas infini). Le résultat du programme linéaire exprime, par u.t., pour chaque processeur et pour chaque tâche, la quantité de tâches traitées. Afin de calculer un nombre entier de graphes à chaque période, ces fractions doivent être mises au même dénominateur, soit le *ppcm* de tous les dénominateurs des fractions de $cons$. Pour l'ordonnancement de j_2 sur G_0 (fig. 3), on trouve 200, soit l'exécution de 16 graphes toutes les 200 u.t. (débit de $16/200 = 2/25$). Or en changeant ces fractions entre processeurs sachant résoudre des tâches de même type, il est possible de réduire cette période. Sur cet exemple, il est en effet possible de diviser par 4 la taille de la période, après réorganisation, tout en conservant un débit optimal. Les matrices figure 4 montrent une réorganisation conduisant la nouvelle période.

		P_1	P_2	P_3	P_4
Tâche	t_0^A	2	-	-	2
	t_1^B	1	3	-	-
	t_2^C	-	2	2	-
	t_3^D	-	-	3	1

(a) Matrice avec dénom. de 50

		P_1	P_2	P_3	P_4
Tâche	t_0^A	1/25	-	-	1/25
	t_1^B	1/50	3/50	-	-
	t_2^C	-	1/25	1/25	-
	t_3^D	-	-	3/50	1/50

(b) Matrice de consommation fractionnelle

FIGURE 4: (a) Résultats avec un dénominateur commun.

4.2.2. Algorithme de calcul de la période minimale

Pour réduire la taille de la période, il faut réduire le *ppcm* des dénominateurs des consommations des tâches par processeur. Ce problème est équivalent à maximiser le *pgcd* (plus grand commun diviseur) des consommations exprimées sous le même dénominateur. Pour maximiser ce *pgcd*, nous pouvons redistribuer des consommations de tâches entre les processeurs, tant que ces redistributions respectent les contraintes du programme linéaire.

Maximiser ce *pgcd* est un problème d'optimisation en nombres entiers. La programmation par contraintes en domaines finis est adaptée à la résolution de ce type de problème. Le solveur Prolog *swi-prolog* est utilisé à cet effet. Le prédicat principal `reduitPeriode` (algo. 1) est une simple boucle qui teste tous les diviseurs possibles jusqu'à 2. Ce prédicat appelle le prédicat secondaire `reorganise` (algo. 2) qui teste si un diviseur "`pgcd`" permet une réorganisation de la période.

La première tâche du prédicat `reorganise` est de définir le domaine des variables ; la borne inférieure est 0 et la borne supérieure $borneSup = \lfloor consPgcd/c[p_i][t_l] \rfloor$ (ou 0 si $c[p_i][t_k]$ est nul). La borne supérieure est définie comme telle, car il est impossible d'exécuter plus de $borneSup$ tâches t_k sur le processeur p_i sans dépasser la capacité du processeur. Le prédicat `reorganise` contraint alors le débit : la somme des consommations d'une tâche doit être égale au débit calculé par le programme linéaire. Puis `reorganise` contraint la charge du processeur : la somme des consommations d'un processeur,

Algorithme 1 : `reduitPeriode`(cons : Matrice, c : Matrice, consPgcd : entier) : Matrice

```
pgcd ← longueurPeriode/denominateurDebit;
tant que pgcd > 1 faire
    newCons : Matrice;
    si newCons ←
    reorganise(cons, c, consPgcd, pgcd) alors
        retourner newCons;
    sinon
        pgcd ← pgcd − 1;
    fin
fin
retourner cons;
```

Algorithme 2 : `reorganise`(cons : Matrice, c : Matrice, consPgcd : entier, pgcd : entier) : Matrice

```
newCons : Matrice;
definiLeDomaineDesVariables(newCons, consPgcd, c);
forceContrainteDebit(newCons, cons);
forceContrainteChargeProcesseur(newCons, cons);
forceDivisiblePar(newCons, pgcd);
si labeling(newCons) alors
    retourner newCons;
fin
retourner null;
```

pondérées par le coût associé, ne doit pas dépasser sa capacité. La dernière contrainte force chacune des valeurs à être divisible par le "pgcd" proposé.
Le solveur est exécuté ($labeling(newCons)$) et, si une valeur est trouvée pour chacune des variables de $newCons$, la nouvelle matrice est retournée. La période est alors réduite d'un facteur entier et respecte les contraintes du programme linéaire et le débit précédemment trouvé. Le résultat de l'exécution de cet algorithme sur l'exemple de la figure 3 est présenté dans la figure 4.

4.2.3. Complexité

La programmation par contraintes en domaines finis est théoriquement NP-Complet [9], cependant sur l'ensemble du jeu de tests, le temps d'exécution de notre algorithme est inférieur à 1 s sur un Core 2 Duo cadencé à 1667 MHz.

5. Expérimentations

Pour évaluer les améliorations de l'algorithme décrit ci-avant, nous utilisons un simulateur de grille. Cette méthode permet d'avoir des résultats reproductibles, ce qui ne serait pas possible sur une grille réelle. Le simulateur utilisé est SimGrid par l'intermédiaire de l'API (*Application Programming Interface*) MSG [2, 3]. Les algorithmes s'exécutent sur un nœud maître qui distribue les tâches aux nœuds esclaves, selon l'ordonnancement calculé.

5.1. Jeux de test

		G_0				G_1			
		p_1	p_2	p_3	p_4	p_1	p_2	p_3	p_4
Type	A	20	∞	∞	20	20	∞	20	20
	B	10	10	∞	∞	10	10	∞	10
	C	∞	10	10	∞	10	10	10	∞
	D	∞	∞	10	10	∞	10	10	10

FIGURE 5: Jeux de grilles G_0 et G_1

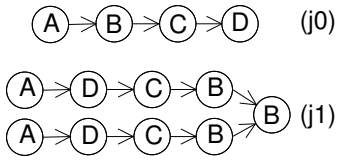


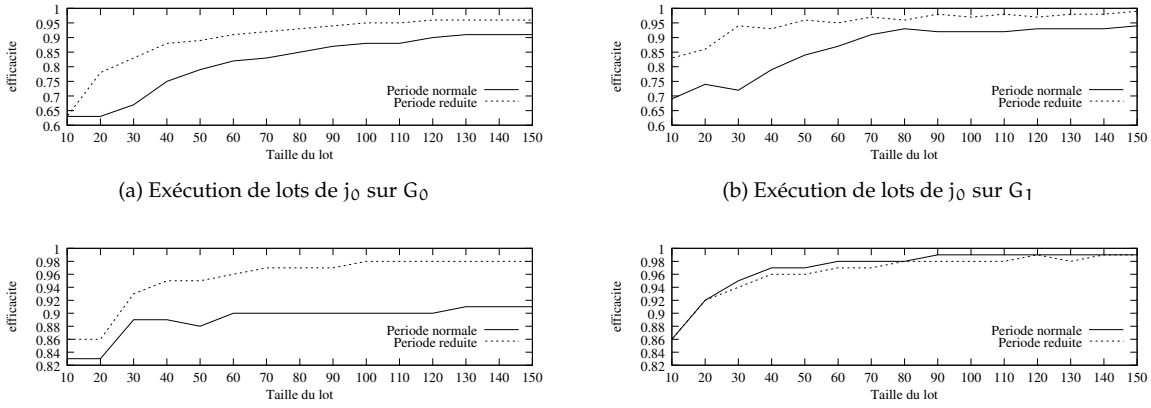
FIGURE 6: Jeux de graphes (job j_0, j_1)

Les figures 5 et 6 présentent respectivement les différentes grilles et les différents graphes de tâches (jobs) utilisés au cours de ces simulations. Nous avons choisi deux types de grilles, avec respectivement deux et trois bibliothèques présentes par processeur afin de modifier les possibilités de réduction de la période. Le job j_0 est considéré comme *simple* car il est séquentiel et ne possède que 3 dépendances. Ces 3 dépendances conduisent au maximum à 3 instances partielles du job j_0 en initialisation/terminaison pour une instance en régime permanent. Le job j_1 est considéré comme plus complexe car la tâche B finale dépend de deux tâches différentes. De plus, ce job possède 8 dépendances. De ce fait, il y un maximum de 8 instances partielles du job j_1 en initialisation/terminaison pour une instance en régime permanent. Ces choix ont été faits pour illustrer différents comportements des phases d'initialisation/terminaison avec des possibilités plus ou moins nombreuses de redistribution des tâches sur les processeurs.

5.2. Réduction de la période et performances

Les résultats présentés et discutés ici traduisent la performance des algorithmes. Ces résultats sont obtenus par la simulation de l'exécution des travaux j_0 et j_1 sur les grilles G_0 et G_1. Nous choisissons comme temps de référence pour le calcul d'une instance de job, l'inverse du débit en régime permanent. Pour une grille donnée, ce temps, multiplié par la taille du lot à exécuter, est une borne inférieure du temps de calcul car en régime permanent, le débit est optimal. Les performances présentées mesurent l'efficacité des algorithmes testés : $efficacité = makespan_o/makespan_r = taille\ du\ lot/(débit \times makespan_r)$.

Le job j_0 (figures 7a et 7b) : sur la grille G_0, la période du régime permanent est de 200 u.t. pendant lesquelles 16 instances de j_0 sont produites (débit de $16/200 = 2/25$). Après réduction, la période descend à 50 u.t. pour 4 instances. L'algorithme de réduction de la période permet donc de diviser le



(a) Exécution de lots de j_0 sur G_0

(b) Exécution de lots de j_0 sur G_1

(c) Exécution de lots de j_1 sur G_0

(d) Exécution de lots de j_1 sur G_1

FIGURE 7: Simulations de j_0 et j_1 sur G_0 et G_1 avec les deux versions du régime permanent

nombre d'instances par 4. Sur G_1, la période du régime permanent passe de 1200 u.t., avec 96 instances de j_0, à 50 u.t. pour 4 instances. Cette fois, le facteur de réduction de la période est de 24.
Pour la configuration j_0 sur G_0 (fig. 7a), les efficacités des deux régimes permanents sont égales pour un lot de 10 instances. Cependant, le régime permanent avec réduction de la période tend beaucoup plus vite vers l'optimal ($efficacité = 1$). Dans les résultats de l'exécution d'un lot de jobs j_0 sur la grille G_1 (fig. 7b), nous pouvons observer que la réduction de la période permet une nette amélioration de l'efficacité, dès 10 instances. Le meilleur gain d'efficacité par rapport à une période non réduite est atteint pour 30 instances.
Ici, la construction de la phase d'initialisation est facile dans tous les cas. Ceci permet de tirer avantage de la réduction du nombre d'instances à traiter au cours de cette phase, d'où le gain important apporté par la réduction de la taille de la période.

Le job j_1 (figures 7c et 7d) : sur la grille G_0, l'algorithme de réduction de la période permet de passer d'une période de 1320 u.t. pour 48 instances (débit de 2/55), à une période de 110 u.t. pour 4 instances. La taille de la période est donc réduite par un facteur 12. Sur la grille G_1, la réduction permet de diviser la taille de la période par 3. Avant réduction la période est de 330 u.t. pour 12 instances, après réduction, elle est de 110 u.t. pour 4 instances.
Les résultats de l'exécution d'un lot de jobs j_1 sur la grille G_0 sont représentés par la figure 7c. La période sans réduction est de 1320 u.t. pour 48 instances. L'initialisation consiste en l'exécution de 194 instances partielles car les dépendances entre tâches mères et filles exécutées sur le même processeur n'ont pas besoin d'être résolues en initialisation. De ce fait, étant donné le nombre d'instances par lot, nous n'arrivons pas au régime permanent. La comparaison se fait donc entre le régime permanent, avec période réduite, et l'algorithme de liste utilisé pour les phases d'initialisation et de terminaison. Dans ces expériences, bien que l'algorithme de liste ait de bonnes performances, la réduction de la période conduit à une meilleure efficacité.
Pour un lot de jobs j_1 sur la grille G_1, les résultats obtenus sont représentés figure 7d. Ici, le régime permanent exécute 38 instances partielles en initialisation. En dessous d'un lot de 38 instances, le régime permanent n'est jamais atteint. C'est seulement à partir de 86 instances qu'une période entière est exécutée. Nous pouvons donc considérer que jusqu'au lot de 150 instances (deux périodes entières), la comparaison se fait, là encore, entre le régime permanent avec période réduite et l'algorithme de liste utilisé pour les phases d'initialisation et de terminaison. Dans ces résultats, nous pouvons voir que la réduction de la période ne permet pas de dépasser les bonnes performances de l'algorithme de liste. Mais la différence entre les deux exécutions ne dépasse pas 1%.

Les résultats présentés ici montrent que pour des lots de petite taille, les phases de préparation et de terminaison du régime permanent conditionnent le *makespan*. Ces résultats corroborent le lien entre la

taille de période du régime permanent et le temps passé dans les phases d'initialisation et de terminaison. Or, réduire la période diminue le nombre d'instances à traiter avant l'entrée en régime permanent. Ainsi, par notre approche, nous réduisons l'influence des phases sous-optimales et descendons le seuil d'utilisation du régime permanent. L'importance de la réduction du *makespan* n'est pas toujours prévisible car elle dépend du comportement de l'algorithme de liste et du nombre de périodes permettant de traiter le lot. De plus, cette réduction n'est pas toujours possible car le résultat du programme linéaire peut conduire directement à la période minimale (c.a.d. j_0 sur une grille avec une fonction différente sur 4 processeurs). Nous avons montré dans [5] le bon comportement de l'approche par régime permanent pour les lots de taille moyenne à grande. Nous étendons ici son domaine d'application aux lots de taille plus modeste.

6. Conclusion

Les expériences menées ont permis de mettre en évidence le lien entre le nombre de travaux dans une période et le coût d'initialisation et de terminaison. La réduction de la période permet de réduire ce coût, tout en conservant un débit optimal, car il y a moins de travail à effectuer lors des phases d'initialisation et de terminaison. Cependant, cette réduction n'est pas toujours possible.
Des travaux à venir permettront de réduire encore le coût d'initialisation et de terminaison. Deux axes peuvent être explorés. Le premier est une résolution des dépendances à l'intérieur d'une période. En effet, lors de la construction de la période du régime permanent, il peut être possible de résoudre des dépendances directement à l'intérieur d'une période. Il n'est plus nécessaire alors de les résoudre pendant l'initialisation. Le second axe est une réduction sous-optimale de la période. Il est en effet probable que la dégradation de la période du régime permanent, en supprimant des fonctions lentes sur un processeur, soit comblée par la réduction des coûts d'initialisation et de terminaison sur des petits lots.

Bibliographie

1. Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. *Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms*. IEEE Computer Society Press, 2004.
2. Henri Casanova. Simgrid : A toolkit for the simulation of application scheduling. *ccgrid*, 00 :430, 2001.
3. Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling distributed applications : the simgrid simulation framework. In *In Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid*, 2003.
4. M. Daoud and N. Kharma. Gats 1.0 : A novel ga-based scheduling algorithm for task scheduling on heterogeneous processor nets. In *Genetic And Evolutionary Computation Conference*, 2005.
5. Sékou Diakité, Jean-Marc Nicod, and Laurent Philippe. Comparison of batch scheduling for identical multitasks jobs on heterogeneous platforms. In *Proceedings of PDP 2008, 16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, 2008. To appear.
6. R.L. Graham and al. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Ann. Discrete Math.*, 4 :287–326, 1979.
7. C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. *DAMATH : Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 108, 2001.
8. M. Iverson and F. Özgüner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, pages 70 – 78, 1998.
9. Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and H. C. Yap. Beyond Finite Domains. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 86–94, 1994.
10. Y. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multi-processors. In *IEEE Transactions on Parallel and Distributed Systems*, pages 506 – 521, 1996.
11. Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4) :406–471, 1999.
12. Arnaud Legrand and Yves Robert. *Algorithmique Parallèle*. Sciences Sup. Dunod, jan 2003.
13. Y. Li and W. Wolf. Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors. In *Proceedings of the 1997 European conference on Design and Test*, pages 134 – 139, 1997.
14. Xiao Qin and Hong Jiang. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 65(8) :885–900, 2005.

15. H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. In *IEEE Transactions on Parallel and Distributed Systems*, pages 260 – 274, 2002.