RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Ordonnancement sur plates-formes hétérogènes de tâches partageant des données

Arnaud Giersch

ICPS/LSIIT
Pôle API, Boulevard Sébastien Brant, BP 10413
67412 Illkirch-Graffenstaden Cedex, France
Arnaud.Giersch@icps.u-strasbg.fr

Résumé
Cet article est consacré à l'ordonnancement d'un grand ensemble de tâches indépendantes sur des plates-formes hétérogènes distribuées. Les tâches dépendent de données (en entrée) qui sont initialement réparties sur les différents nœuds de la plate-forme. Une certaine donnée peut être partagée par plusieurs tâches. Pour chaque tâche, notre problème est de décider sur quel nœud de la plate-forme l'exécuter, et de transférer les données nécessaires (celles dont dépend la tâche) vers ce nœud. L'objectif est de trouver une allocation des tâches, et un ordonnancement des communications induites, qui minimisent le temps total d'exécution. Sur le plan théorique, nous exposons différents résultats qui caractérisent la difficulté du problème. Sur le plan pratique, nous proposons plusieurs nouvelles heuristiques, que nous comparons à des heuristiques classiques comme *min-min* ou *sufferage* grâce à des simulations.

Mots-clés : ordonnancement, hétérogène, heuristiques, simulations, tâches indépendantes, données partagées

1. Introduction

Dans cet article, nous nous intéressons à l'ordonnancement de tâches indépendantes sur des plates-formes hétérogènes de type grille de calcul. Ces tâches indépendantes dépendent de données d'entrée, et la difficulté réside dans le fait que certaines données peuvent être *partagées* par plusieurs tâches, c'est-à-dire que plusieurs tâches peuvent utiliser une même donnée en entrée. Une tâche peut également utiliser plusieurs données. Les données sont initialement réparties sur les différents nœuds de la plate-forme, et elles peuvent être répliquées sur plusieurs nœuds. Avant de pouvoir exécuter une tâche, il faut rassembler les données dont elle dépend sur son lieu d'exécution. Pour chaque tâche, il nous faut décider du lieu où elle sera exécutée, puis ordonnancer les transferts de données de manière à minimiser le temps total d'exécution.

Ce travail est motivé par celui de Casanova, Legrand, Zagorodnov et Berman [6, 5] qui vise à ordonnancer des tâches dans le cadre d'*AppLeS Parameter Sweep Template* (APST) [3]. APST est un environnement de grille dont le but est de faciliter l'exécution d'applications sur des plates-formes hétérogènes. Une application pour APST consiste typiquement en un *grand* nombre de tâches indépendantes avec un partage éventuel des données en entrée. Les simulations par Monte-Carlo multiples [4, 14] sont, par exemple, des applications entrant dans ce cadre.

Nous disons que le nombre de tâches dans une application pour APST est *grand* car il est habituellement au moins d'un ordre de grandeur plus grand que le nombre de ressources de calcul disponibles. L'idée intuitive en déployant ce type d'applications est de placer sur un même processeur des tâches qui dépendent de mêmes données. Casanova *et al.* [6, 5] ont évalué trois heuristiques initialement conçues pour des tâches complètement indépendantes (sans partage de données) qui avaient été présentées par Maheswaran, Ali, Siegel, Hensgen et Freund dans [15]. Ils ont modifié ces trois heuristiques (originellement appelées *min-min*, *max-min* et *sufferage* par Maheswaran *et al.*) pour les adapter à la contrainte

supplémentaire que les données d'entrée peuvent être partagées entre plusieurs tâches. Étant donné que le nombre de tâches à ordonnancer peut être très grand, il faut faire particulièrement attention à ce que le coût des heuristiques d'ordonnancement reste relativement faible.
Nous allons donc, dans cet article, traiter le même problème d'ordonnancement, mais en proposant de nouvelles heuristiques qui se révèlent aussi performantes que les meilleures heuristiques dans [6, 5] mais pour un coût beaucoup plus faible. Dans la section 2, nous commençons par nous intéresser à des plates-formes centralisées de type maître-esclave. Le modèle de plate-forme sera ensuite généralisé, en section 3, à un ensemble décentralisé de serveurs reliés entre eux par un réseau d'interconnexion quelconque. Dans les deux parties, nous commençons par présenter des résultats théoriques sur la complexité du problème, avant d'exposer nos nouvelles heuristiques. Les différentes heuristiques sont ensuite évaluées en les comparant entre elles à l'aide de nombreuses simulations. Dans la section 4, nous présentons différents travaux s'attaquant à des problèmes similaires, avant de conclure en section 5.
À cause du manque de place, nous renvoyons à [9, 10, 11, 12] pour les détails manquants.

2. Avec un seul serveur

Dans cette section, nous nous limitons à des plates-formes centralisées de type maître-esclave : l'ensemble des données est initialement détenu par un nœud maître, et les tâches sont à exécuter sur un ensemble de processeurs esclaves directement reliés au maître.

2.1. Modèles

Notre problème est d'ordonnancer un ensemble de n tâches $\mathcal{T} = \{T_1, \ldots, T_n\}$. Ces tâches ont des tailles différentes : le poids de la tâche T_j est t_j, $1 \leq j \leq n$. Il n'y a pas de contraintes de dépendance entre les tâches, elle peuvent donc être vues comme indépendantes. L'exécution de chaque tâche dépend cependant de une ou plusieurs données, et une certaine donnée peut être partagée par plusieurs tâches. En tout, il y a m données dans l'ensemble $\mathcal{D} = \{D_1, \ldots, D_m\}$. La taille de la donnée D_i est d_i, $1 \leq i \leq m$. Nous utilisons un graphe biparti (cf. fig. 1 pour un exemple) pour représenter les relations entre les tâches et les données : c'est le graphe d'application. Intuitivement, une donnée D_i reliée par une arête à une tâche T_j correspond à une donnée qui est nécessaire pour pouvoir commencer l'exécution de T_j.

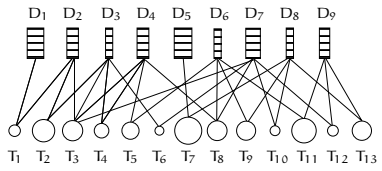

FIG. 1 – Des tâches et des données.

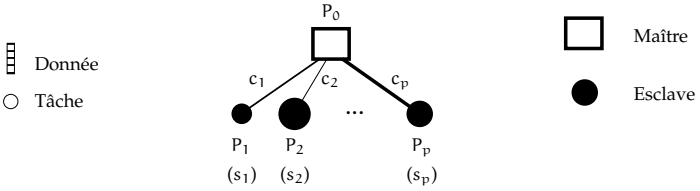

FIG. 2 – Une plate-forme maître-esclave hétérogène.

Les tâches sont à ordonnancer et à exécuter sur une plate-forme de type maître-esclave hétérogène avec un processeur maître P_0 et p processeurs esclaves P_i, $1 \leq i \leq p$ (cf. fig. 2). Chaque processeur esclave P_q a une vitesse s_q : il faut t_j/s_q unités de temps pour exécuter la tâche T_j sur le processeur P_q. Le processeur P_0 détient initialement chacune des m données de $\mathcal{D}$. Les esclaves sont chargés d'exécuter les n tâches de $\mathcal{T}$. Avant de pouvoir exécuter une tâche T_j, un esclave doit avoir reçu, de la part du maître, toutes les données dont dépend T_j. Pour les communications, nous utilisons le modèle un-port : le maître peut seulement communiquer avec un seul esclave à la fois. Nous notons c_q l'inverse de la bande passante du lien entre P_0 et P_q. Il y a donc besoin de $d_i \cdot c_q$ unités de temps pour transférer la donnée D_i du maître P_0 à l'esclave P_q. Nous supposons que les communications peuvent recouvrir les calculs sur les esclaves : un esclave peut traiter une tâche tout en recevant les données nécessaires à l'exécution d'une autre tâche.

L'objectif est de minimiser le temps total d'exécution (ou *makespan*). L'exécution est terminée quand la dernière tâche a été achevée. L'ordonnancement doit décider du placement des tâches sur les processeurs esclaves. Il doit également décider de l'ordre dans lequel le maître doit envoyer les données aux esclaves. Nous appelons TASKSSHARINGFILES le problème d'optimisation à résoudre.

Nous insistons sur deux points importants :

– des données peuvent être envoyées plusieurs fois pour que plusieurs esclaves puissent traiter des tâches (différentes) dépendant de ces données ;
– une donnée envoyée à un processeur y reste disponible pour le reste de l'ordonnancement (persistance des données). Si deux tâches dépendant d'une même donnée sont placées sur le même processeur, la donnée n'a besoin d'être envoyée qu'une seule fois.

Ainsi, une manière de réduire le temps total d'exécution va être de chercher à réduire la quantité de données répliquées en plaçant, sur un même processeur, l'ensemble des tâches dépendant d'une même donnée, car le temps de communication se trouverait ainsi réduit. On a alors le risque de placer toutes les tâches sur le même processeur. Si, au contraire, on cherche à équilibrer la charge des processeurs, on a le risque d'avoir beaucoup de communications. Il y a donc un équilibre à trouver entre ces deux positions extrêmes.

2.2. Complexité

Il est connu que la plupart des problèmes d'ordonnancement sont difficiles. Certaines instances particulières du problème d'optimisation TASKSSHARINGFILES ont néanmoins une complexité polynomiale, alors que les problèmes de décision associés à d'autres instances sont NP-complets. Différents résultats sont rassemblés dans le tableau 1. Dans ce tableau, les résultats sont indiqués en fonction de la plate-forme et de l'application.

TAB. 1 – Résultats de complexité pour notre problème d'ordonnancement de tâches, avec et sans partage des données (P = polynomial, NPC = NP-complet, ? = complexité inconnue).

Plate-forme	Application			
	sans partage des données		avec partage des données	
	homogène	hétérogène	homogène	hétérogène
Un seul esclave	P	P	?	NPC
Plusieurs esclaves homogènes	P	NPC	?	NPC
Plusieurs esclaves hétérogènes	P	NPC	NPC	NPC

Dans le cas où il y a partage des données, le problème de décision associé à TASKSSHARINGFILES est ainsi NP-complet, même dans les cas simples où (1) il n'y a qu'un seul processeur, mais les tâches et les données sont de tailles hétérogènes ; ou (2) les tâches et les données sont de taille unitaire, mais la plate-forme est composée de deux processeurs hétérogènes avec deux liens de bandes passantes différentes [9, 12]. Dans la version générale du problème, les tailles des tâches, ainsi que les tailles des données sont hétérogènes, les différents processeurs ont des vitesses différentes et leurs connexions depuis le maître ont des bandes passantes différentes. C'est pourquoi nous concevons des heuristiques polynomiales pour résoudre le problème TASKSSHARINGFILES. Nous évaluerons ensuite leurs performances grâce à de nombreuses simulations.

2.3. Heuristiques

Nous comparons nos nouvelles heuristiques aux cinq heuristiques *min-min*, *max-min*, *sufferage*, *sufferage II* et *sufferage X* présentées par Casanova, Legrand, Zagorodnov et Berman [6, 5]. Ce sont nos *heuristiques de référence*. Toutes les heuristiques de référence sont construites sur le modèle suivant : pour chaque tâche restante T_j, évaluer OBJECTIF(T_j, P_i) pour tous les processeurs ; choisir le « meilleur » couple (T_j, P_i) puis ordonnancer au plus tôt T_j sur P_i. Ici, OBJECTIF(T_j, P_i) est le temps de complétion minimum (MCT ou *Minimum Completion Time*) de la tâche T_j si elle est placée sur le processeur P_i, étant données les décisions d'ordonnancement qui ont déjà été faites. Les heuristiques diffèrent uniquement

par la définition du « meilleur » couple (T_j, P_i). Par exemple, pour *min-min*, la « meilleure » tâche T_j est celle qui minimise la fonction objectif si elle est placée sur son processeur le plus favorable. La complexité algorithmique des heuristiques de référence est au moins de $O(p \cdot n \cdot (n + |\mathcal{E}|))$, où $\mathcal{E}$ est l'ensemble des arêtes du graphe biparti d'application.

En concevant de nouvelles heuristiques, nous avons fait particulièrement attention à en réduire la complexité. Afin d'éviter la boucle sur toutes les couples de tâche et de processeur dans les heuristiques de référence, nous avons besoin d'être capables de sélectionner (plus ou moins) en temps constant la prochaine tâche à ordonnancer. Nous avons donc décidé de trier une fois pour toutes les tâches suivant une fonction objectif. Cependant, comme notre plate-forme est hétérogène, les caractéristiques d'une tâche peuvent varier d'un processeur à l'autre. C'est pourquoi nous construisons une liste triée de tâches pour chacun des processeurs. Cette liste triée est calculée initialement et n'est pas modifiée pendant l'exécution de l'heuristique. Une fois que les listes triées sont calculées, il nous reste à placer les tâches sur les processeurs et à les ordonnancer. Les tâches sont ordonnancées les unes après les autres. Quand nous voulons ordonnancer une nouvelle tâche, nous évaluons, pour chaque processeur, le temps de fin de la première tâche (suivant la liste triée) qui n'a pas encore été ordonnancée. Nous choisissons ensuite le couple de tâche et de processeur avec le temps de fin le plus petit. De cette manière, la complexité des heuristiques se trouve réduite à $O(p \cdot n \cdot \Delta T + p \cdot n \cdot \log n)$, où ΔT dénote le nombre maximal de données (plus une) dont dépend une tâche.

Nous avons ainsi conçu six nouvelles heuristiques, chacune avec une fonction objectif différente. Les nouvelles heuristiques avec leur fonction objectif sont regroupées dans le tableau 2. Les fonctions objectif sont décrites en fonction de $T_{calc}(T_j, P_i)$, le temps d'exécution de T_j sur P_j, et de $T_{comm}(T_j, P_i)$, le temps de transfert, de P_0 vers P_j, de toutes les données nécessaires à l'exécution de T_i. Chacune des nouvelles heuristiques peut être associée avec une combinaison de politiques additionnelles qui sont également résumées dans le tableau 2. Nous obtenons finalement pas moins de 44 variantes : toute combinaison des trois politiques additionnelles peut être utilisée avec les six fonctions objectifs de base, mais la politique *shared* n'a évidemment aucune influence sur la fonction de base *computation*.

TAB. 2 – Définitions des nouvelles heuristiques.

Heuristique	Fonction objectif	Ordre de tri des tâches suivant la fonction objectif
computation	$T_{calc}(T_j, P_i)$	croissant
communication	$T_{comm}(T_j, P_i)$	croissant
duration	$T_{calc}(T_j, P_i) + T_{comm}(T_j, P_i)$	croissant
payoff	$T_{calc}(T_j, P_i) / T_{comm}(T_j, P_i)$	décroissant
advance	$T_{calc}(T_j, P_i) - T_{comm}(T_j, P_i)$	décroissant
Johnson	$\min(T_{calc}(T_j, P_i), T_{comm}(T_j, P_i))$	suivant l'algorithme de Johnson pour un *flow-shop* à deux machines [13]

Politique additionnelle	Explication
shared	Dans les fonctions objectifs, des tailles pondérées pour les données sont utilisées. La taille pondérée d'une donnée est obtenue en divisant la taille de la donnée par le nombre de tâches qui en dépendent.
readiness	S'il y a une tâche prête sur le processeur P_i, elle est sélectionnée au lieu de la première tâche de la liste de P_i. Pour un processeur P_i les tâches prêtes sont celles dont toutes les données se trouvent déjà sur P_i.
locality	Le placement d'une tâche T_j sur un processeur P_i est retardé si certaines des données dont dépend T_j sont déjà présentes sur un autre processeur.

2.4. Évaluation par simulations

Aucune de nos heuristiques d'ordonnancement n'est garantie. Afin de pouvoir évaluer leur qualité, nous les avons comparées aux heuristiques de référence à l'aide de nombreuses simulations.

Génération des tests

Les plates-formes simulées sont composées de 20 processeurs. Les vitesses des processeurs et les bandes passantes des liens de communication sont tirées aléatoirement d'un ensemble de valeurs mesurées dans la réalité. Les bandes passantes des liens de communication et les vitesses des processeurs sont ensuite normalisées de manière à contrôler le ratio des coûts de communication et de calcul. Nous simulons ainsi trois principaux types de problèmes : intensif en calculs, intensif en communications et intermédiaire.

Nous avons testé les heuristiques sur quatre types de graphes d'application que nous espérons représentatifs d'une grande classe d'application. Les tailles des tâches et des données sont tirées aléatoirement. Les graphes sont schématisés par la figure 3 : *étoile* (chaque tâche dépend exactement d'une donnée partagée avec d'autres tâches), *deux-un* (chaque tâche dépend exactement de deux données : une donnée privée et une donnée partagée avec d'autres tâches), *partitionné* (le graphe est divisé en plusieurs morceaux et dans chaque morceau, les dépendances entre les tâches et les données sont tirées aléatoirement) et *aléatoire* (les dépendances entre les tâches et les données sont tirées aléatoirement). Chaque graphe contient 2 000 tâches et 2 500 données, sauf les graphes en étoile qui contiennent également 2 000 tâches mais seulement 100 données.

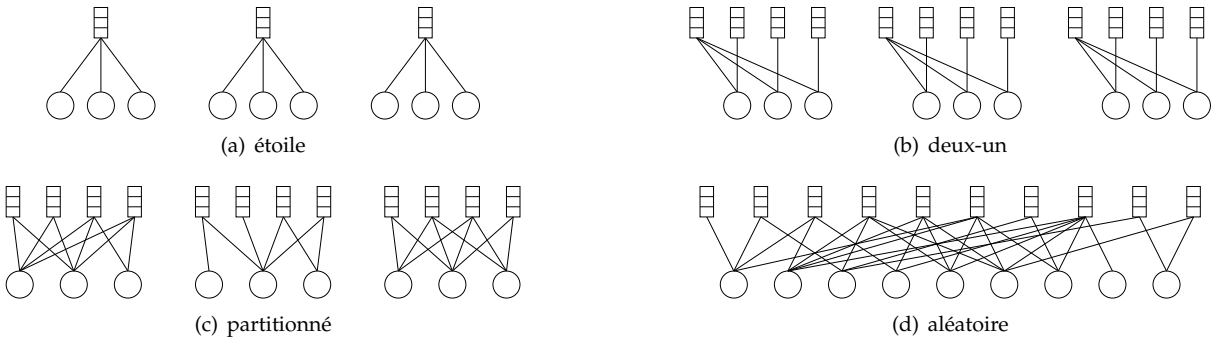

FIG. 3 – Les quatre types de graphes d'application utilisés pour les simulations.

Résultats

Le tableau 3 résume l'ensemble des expérimentations. Dans ce tableau, nous rapportons la performance et le coût des dix meilleures heuristiques. La performance d'une heuristique est calculée à partir de la durée de l'ordonnancement produit. Le coût d'une heuristique reflète le temps qu'il lui a fallu pour produire cet ordonnancement. Le tableau 3 est un résumé de 12 000 tests aléatoires (1 000 tests sur les quatre types de graphes d'application et les trois ratios des coûts de communication et de calcul). Chaque test concerne 49 heuristiques (5 heuristiques de référence et 44 combinaisons de nos nouvelles heuristiques et des politiques additionnelles). Pour chaque test, nous calculons le rapport entre la performance de chaque heuristique et celle de la meilleure heuristique, ce qui nous donne une *performance relative*. La meilleure heuristique n'est pas la même d'un test à l'autre, ce qui explique pourquoi aucune des heuristiques du tableau 3 n'a une performance relative moyenne exactement égale à 1. En d'autres termes, la meilleure heuristique n'est pas la meilleure pour chacun des tests, mais est celle qui est la plus proche, en moyenne, de la meilleure pour chaque test. Le performance relative optimale de 1 serait obtenue en prenant, pour chacun des 12 000 tests, la meilleure heuristique pour ce cas particulier.

Nous pouvons voir, dans le tableau 3, que l'heuristique *sufferage* donne les meilleurs résultats en moyenne : elle est à 11% de l'optimum relatif de 1. Les neuf heuristiques suivantes dans le classement ne sont pas très loin : elles sont entre 13% et 14,7% de l'optimum relatif de 1. Parmi ces neuf heuristiques, seule *min-min* est une heuristique de référence. Nous pouvons remarquer que les performances des nouvelles heuristiques apparaissant dans le tableau 3 suivent de très près les performances du *min-min*. De plus, les écarts types sont plus petits pour nos heuristiques que pour les heuristiques de référence.

TAB. 3 – Performance et coût relatifs des dix meilleures heuristiques.

Heuristique	Performance relative	Écart type	Coût relatif	Écart type
sufferage	1,110	0,164	377	153
min-min	1,130	0,198	419	192
computation+readiness	1,133	0,110	1,57	0,425
duration+locality+readiness	1,133	0,130	1,50	0,454
duration+readiness	1,133	0,130	1,45	0,367
payoff+shared+readiness	1,138	0,126	1,50	0,605
payoff+readiness	1,139	0,127	1,25	0,249
payoff+shared+locality+readiness	1,145	0,127	1,57	0,577
payoff+locality+readiness	1,145	0,127	1,32	0,233
computation+locality+readiness	1,147	0,123	1,62	0,475

Cela reflète une plus grande stabilité dans la qualité des résultats produits. Des résultats plus détaillés, ainsi qu'une analyse plus approfondie de ces résultats, peuvent être trouvés dans [9]. Dans le tableau 3 sont également rapportés les coûts des différentes heuristiques (temps nécessaire pour construire l'ordonnancement). L'analyse théorique est confirmée : nos nouvelles heuristiques sont au moins d'un ordre de grandeur plus rapides que les heuristiques de référence. Pour conclure, étant données leurs bonnes performances comparativement à *sufferage*, nous pensons que les huit nouvelles variantes du tableau 3 apportent une très bonne alternative aux coûteuses heuristiques de référence. De manière plus précise, on peut dire que c'est la variante *readiness* qui apporte vraiment quelque chose. Pour ce qui est des heuristiques de base, il semble que ce soit l'idée la plus simple (parmi les heuristiques que nous avons testées) qui marche le mieux, c'est-à-dire l'heuristique *duration* qui utilise une estimation du temps d'exécution des tâches sur les esclaves.

3. Avec plusieurs serveurs

Dans la section précédente, la plate-forme d'exécution était de type maître-esclave. Nous allons maintenant considérer une instance plus générale du problème d'ordonnancement : nous supposons que la plate-forme est complètement décentralisée. Cette plate-forme est constituée de plusieurs serveurs, avec des capacités de calculs différentes, et reliés entre eux par un réseau d'interconnexion.

3.1. Modèles

Notre problème est d'ordonnancer le même type d'application que dans la section précédente. Nous conservons donc les mêmes notations (cf. section 2.1).

Concernant la plate-forme d'exécution, nous généralisons le modèle de la section précédente. Les tâches sont ordonnancées et exécutées sur une plate-forme hétérogène composée d'un ensemble de s *serveurs* S_i ($1 \leq i \leq s$) reliés entre eux par des routeurs et des liens. Un lien de communication peut relier deux serveurs, deux routeurs ou un routeur et un serveur. Nous supposons qu'il y a un chemin entre chaque paire de serveurs, et que tous les liens sont bidirectionnels. Chaque serveur S_i est composé d'un entrepôt local E_i associé à une grappe (*cluster*) de calcul locale C_i. Les données nécessaires aux calculs (tâches) sont stockées dans les entrepôts. Nous supposons qu'une donnée peut être répliquée et donc être stockée simultanément dans plusieurs entrepôts. Nous supposons que chaque entrepôt est assez grand pour contenir une copie de toutes les données. La figure 4 présente un exemple de plate-forme.

Avant qu'une grappe puisse exécuter une tâche, l'entrepôt correspondant doit contenir toutes les données dont la tâche dépend. Comme précédemment, nous utilisons le modèle un-port pour les communications. Un serveur peut, à un instant donné, effectuer au plus deux communications : une émission et une réception. De plus, nous faisons l'hypothèse que le routage au sein du graphe de plate-forme est fixé : des données envoyées d'un serveur S_i à un serveur S_j emprunteront toujours le même chemin. Le coût des communications entre deux serveurs est défini en fonction du chemin entre ces serveurs, tel qu'il est fixé par le routage. Pour les communications transitant par des serveurs intermédiaires, nous utilisons un modèle *store-and-forward* entre les serveurs : les données sont transférées d'un serveur au

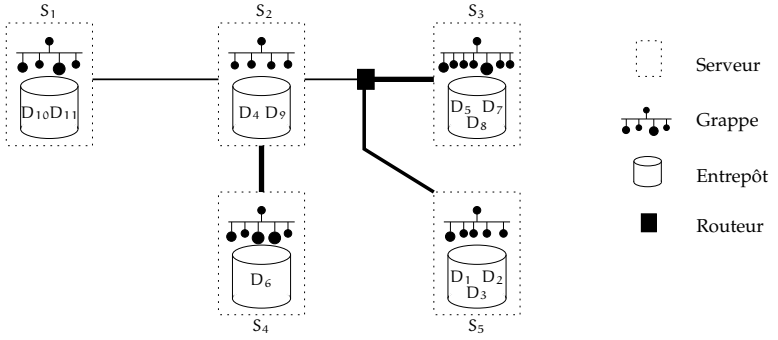

FIG. 4 – Graphe de plate-forme avec la distribution initiale des données.

serveur suivant sur le chemin, en laissant une copie de la donnée dans l'entrepôt de chaque serveur intermédiaire. Le coût total de la communication est la somme des coûts des communications adjacentes. Le fait de laisser des copies des données transférées multiplie le nombre de sources potentielles pour chaque donnée et est susceptible d'accélérer le traitement des tâches suivantes, c'est pourquoi un modèle *store-and-forward* pour les communications nous a semblé adapté à notre problème. Nous faisons finalement la supposition que les temps de communication entre une grappe et son entrepôt associé sont négligeables : il est escompté que le coût des communications intra-serveur est d'un ordre de grandeur plus petit que celui des communications inter-serveurs.

Pour ce qui est des coûts de calcul, chaque grappe C_i est composée de processeurs hétérogènes. Plus précisément, C_i regroupe p_i processeurs $P_{i,k}$ avec $1 \leq k \leq p_i$. La vitesse du processeur $P_{i,k}$ est $s_{i,k}$, ce qui signifie que $t_j/s_{i,k}$ unités de temps sont nécessaires pour exécuter la tâche T_j sur $P_{i,k}$. Une approche plus grossière est de voir la grappe C_i comme une seule ressource de calcul de vitesse cumulée $\sum_{k=1}^{p_i} s_{i,k}$.

Notre objectif est de minimiser le temps total d'exécution (ou *makespan*). L'exécution est terminée lorsque la dernière tâche est terminée. L'ordonnancement doit décider quelle tâche est à exécuter par chacun des processeurs de chaque grappe, et quand. Il doit également décider de l'ordre dans lequel les données nécessaires sont envoyées d'un entrepôt de serveur à un autre. Nous appelons TSFDR (*Tasks Sharing Files from Distributed Repositories*) le problème d'optimisation à résoudre.

Nous insistons sur trois points importants :

– Des données peuvent être envoyées plusieurs fois, pour que des grappes différentes puissent traiter indépendamment des tâches dépendant de ces données.
– Une donnée déposée dans un entrepôt y reste disponible pour le reste de l'ordonnancement. Ainsi, si deux tâches dépendant d'une même donnée sont placées sur la même grappe, la donnée n'a besoin d'être envoyée qu'une seule fois.
– Une donnée est initialement disponible sur un ou plusieurs serveurs (connus) mais, lors de l'envoi d'une donnée d'un de ces serveurs vers un autre, une copie est déposée dans l'entrepôt de chacun des serveurs intermédiaires. Ainsi tous les serveurs intermédiaires, en plus du serveur qui était la destination de la donnée, deviennent une source potentielle pour la donnée.

3.2. Complexité

Nous avons vu en section 2.2 que le problème TSFDR est NP-complet, même dans des versions simplifiées. Cette difficulté est liée à l'hétérogénéité du problème qui peut provenir de différentes sources : les tâches et les données ont des poids différents, pendant que les grappes ou les liens peuvent avoir des vitesses différentes.

En fait, une version entièrement homogène (sans poids) de TSFDR est déjà NP-complète [9]. Dans cette version, nous supposons que toutes les données ont la même taille, que tous les liens de communication ont la même bande passante et qu'il n'y a pas de routeur dans le graphe de plate-forme (tous les liens relient directement deux serveurs). Nous supposons de plus que toutes les tâches ont un poids nul ou bien (ce qui est équivalent) que tous les processeurs ont une vitesse infinie. Dans ce cas simple, nous

avons alors un problème d'allocation : nous avons à placer les tâches sur les grappes et, pour chaque tâche, à rassembler son ensemble de données requises (qui sont les entrées de la tâche) dans l'entrepôt du serveur sur lequel elle est placée. L'objectif est de minimiser la durée totale des communications. Toutes les communications durent une unité de temps, mais des communications indépendantes, c'est-à-dire avec des émetteurs et des récepteurs distincts, peuvent avoir lieu en parallèle.
Ce résultat de NP-complétude est valide pour une version non pondérée du problème original, ce qui prouve que la complexité se trouve déjà dans la combinatoire de l'allocation et de la duplication des données, en l'absence d'hétérogénéité, et dans l'application (tâches et données), et dans la plate-forme (processeurs et liens). Dans la version générale du problème, il y a des poids pour les tâches et pour les données, et la plate-forme est hétérogène. Il est alors vraisemblable qu'il n'existe pas d'algorithme d'approximation, comme c'est le cas pour le problème de partitionnement de graphe. C'est pourquoi, dans la suite, nous concevons des heuristiques polynomiales pour résoudre le problème TSFDR. Nous évaluerons leur performance grâce à de nombreuses simulations.

3.3. Heuristiques

Dans la continuité de ce qui a été fait dans la section précédente, nous commençons par adapter les heuristiques de référence *min-min* et *sufferage*, avant d'introduire plusieurs nouvelles heuristiques dont la principale caractéristique est une complexité algorithmique moindre, tout en conservant une qualité de résultat comparable.

Adaptation des heuristiques de référence

Toutes les heuristiques que nous avons étudié allouent les tâches aux processeurs les unes après les autres. En voulant ordonnancer une tâche sur un processeur donné, il faut tenir compte, d'une part des données nécessaires qui se trouvent déjà dans l'entrepôt correspondant, et d'autre part des données à transférer à travers le réseau de serveurs. Si une donnée à transférer est déjà répliquée dans les entrepôts de plusieurs serveurs, nous décidons (de manière heuristique) de l'amener depuis le serveur le plus proche parmi ceux qui en détiennent une copie. Le serveur le plus proche est celui depuis lequel la donnée serait amenée le plus rapidement si le réseau était entièrement disponible. Une fois que la source pour chaque donnée à transférer a été choisie, il reste à ordonnancer les communications.
Comme les tâches sont allouées aux processeurs les unes après les autres, nous avons à traiter le cas où toutes les communications ont la même destination (le serveur qui va exécuter la tâche). Nous avons montré que c'est un problème NP-complet, même avec notre hypothèse de routage fixé. C'est pourquoi nous avons conçu deux algorithmes d'ordonnancement gloutons pour le résoudre. Le premier algorithme mémorise pour chacun des liens la date et la durée de toutes les communications déjà ordonnancées, puis utilise un *ordonnancement par insertion* (*first-fit*) pour ordonnancer les nouvelles communications dans le premier intervalle de temps disponible. Le second algorithme, plus simple, ordonnance les nouvelles communications le plus tôt possible mais toujours *après* toutes les communications utilisant déjà le même lien. La complexité de l'algorithme d'ordonnancement des communications sera notée O_c. En notant par $\Delta\mathcal{P}$ la plus grande distance entre deux serveurs, $O_c = \Delta\mathcal{P} \cdot \Delta T \cdot m$ dans le cas avec insertion, et $O_c = \Delta\mathcal{P} \cdot \Delta T \cdot \log \Delta T$ sinon.
Grâce à ces décisions sur l'ordonnancement des communications, nous pouvons enfin adapter l'heuristique de référence *min-min*. Plutôt que de considérer directement l'ordonnancement des tâches sur les processeurs de chaque grappe, ce qui serait très coûteux, nous procédons en deux phases. Nous commençons par ordonnancer les tâches sur les serveurs en utilisant une vue simplifiée du problème où chaque grappe C_j est vue comme une seule ressource de calcul de vitesse cumulée $\sum_{k=i}^{p_j} s_{j,k}$. L'ordonnancement des tâches au sein de chaque grappe est faite de manière gloutonne dans un deuxième temps.
La complexité de l'heuristique est alors de $O(s \cdot n \cdot (n \cdot O_c + s \cdot |\mathcal{E}|) + n \cdot p)$, où $p = \max_{1 \leq i \leq s} p_i$. Nous remarquons qu'en passant d'un système avec un seul entrepôt, comme dans la section 2, à un système avec plusieurs entrepôts, la complexité de l'heuristique s'accroît d'un facteur correspondant au coût du choix et de l'ordonnancement des communications. En même temps, le nombre de processeurs a été remplacé par le nombre de serveurs.
À cause de ses bons résultats observés avec un seul serveur, nous adaptons également l'heuristique *sufferage* qui est une variante légèrement plus coûteuse du *min-min*.

Heuristiques moins coûteuses

Comme nous l'avons déjà vu dans la section précédente, l'heuristique *min-min* est séduisante par la qualité de l'ordonnancement produit, mais son coût de calcul est énorme et peut empêcher son utilisation. C'est pourquoi nous cherchons à concevoir des heuristiques qui sont d'un ordre de grandeur plus rapides, tout en essayant de conserver la qualité de l'ordonnancement produit.

Le *min-min* est particulièrement coûteux car, chaque fois qu'il tente d'ordonnancer une nouvelle tâche, il considère toutes les tâches restantes et calcule leurs MCT. Comme précédemment, nous avons plutôt travaillé sur des solutions où nous ne considérons qu'une seule tâche par serveur. Cela nous mène au schéma suivant : tant qu'il reste des tâches à ordonnancer, (1) choisir, pour chaque grappe C_i, la « meilleure » tâche candidate T_k qui reste à ordonnancer ; puis (2) choisir le « meilleur » couple (T_k, C_i) et ordonnancer T_k sur C_i. Toute l'heuristique repose sur la définition de la « meilleure » tâche candidate. Pour cela, nous concevons une *fonction de coût*. Suivant les résultats de notre étude précédente, dans le cas où il n'y a qu'un seul entrepôt, notre fonction de *coût* va représenter la durée totale du transfert des données puis de l'exécution des tâches sur les serveurs. La « meilleure » tâche candidate sera ainsi celle qui a le *coût* le plus faible et qui n'a pas encore été ordonnancée.

Nous définissons le *coût* d'une tâche T_i sur un serveur S_j comme une évaluation du MCT de T_i sur S_j. Le MCT est défini comme la somme du temps nécessaire pour envoyer les données requises vers S_j plus le temps nécessaire à la grappe C_j pour traiter la tâche, quand la grappe est vue comme une ressource de calcul simple. En réalité, nous approximons le temps de communication : au lieu de le calculer en utilisant l'algorithme d'ordonnancement des communications, nous le sur-approximons en utilisant la somme des temps de transfert de chaque donnée requise, dans le cas où tous les liens de communications sont entièrement disponibles.

Nous avons ainsi conçu notre première heuristique : *static*. L'expression de la complexité des nouvelles heuristiques devient complexe et ne peut pas être détaillée ici. Elle est de l'ordre de $n \cdot (\log n + O_c)$ alors qu'elle était de l'ordre de $s \cdot n^2 \cdot O_c$ pour les heuristiques de référence.

Dans notre heuristique statique, nous commençons par définir l'ordre dans lequel les tâches sont considérées et toutes les décisions d'ordonnancement (choix des communications et ordonnancement) sont induites par ce choix original. Cet ordre est cependant basé sur un *coût* qui n'est vraiment approprié que s'il n'y a qu'une seule tâche dans le système. La formule du *coût* ne tient pas compte du fait que d'autres tâches pourraient avoir besoin des mêmes données. Elle ne profite donc pas des possibilités offertes par des nouvelles sources pour ces données qui sont créées durant l'exécution. Pour remédier à ce défaut, nous introduisons une fonction de *coût* dynamique qui nous retourne la même estimation pour le temps de complétion minimum d'une tâche T sur un serveur S que la fonction de *coût* originale retournerait si elle tenait compte, au moment de son utilisation, (1) des données dont T dépend qui ont été répliquées, et du lieu de leur réplication ; et (2) de la quantité de tâches déjà placées sur S.

Après avoir défini le *coût* dynamique, il reste à décider de son utilisation pour sélectionner la (ou les) prochaine(s) tâche(s) à ordonnancer. Nous avons le choix entre (1) prendre une seule tâche à la fois et mettre les *coûts* à jour à chaque fois ; et (2) prendre un ensemble de k tâches et ne mettre les *coûts* à jour qu'après que ces k tâches ont été ordonnancées. La première approche est, dans l'idée, plus proche de l'heuristique *min-min* ; ce sera l'heuristique *dynamic1*. L'autre approche qui pourrait être moins coûteuse sera l'heuristique *dynamic2*. Pour les deux versions, nous visons à nouveau une faible complexité. Le surcoût dû aux mises à jour des coûts est limité en utilisant des structures de données (tas) et des algorithmes de mise-à-jour adaptés. Par exemple, le surcoût de *dynamic1* par rapport à *static* est de $s^2 \cdot (|\mathcal{E}| \cdot \log n + \log s)$.

Nous définissons également des variantes pour nos heuristiques. Nous reprenons la variante *readiness* qui commence par regarder si une tâche peut être ordonnancée sur un serveur avec un coût de communication nul. Cette variante ne s'applique qu'à l'heuristique *static* : elle est incluse d'office dans les heuristiques *dynamic*. Nous définissons également la variante *mct* qui sélectionne la « meilleure » candidate parmi les candidates locales en utilisant leurs MCT, comme le faisaient les heuristiques de la section précédente. L'effet attendu du raffinement *mct* est de mieux équilibrer la charge sur les serveurs. On peut remarquer que l'heuristique *static+mct* correspond directement à l'heuristique *duration+readiness* qui était parmi les meilleures heuristiques de la section précédente. Par rapport aux heuristiques de base, la variante *mct* accroît (grossièrement) la complexité d'un facteur s correspondant à l'évaluation du temps de fin de chaque tâche sur tous les serveurs.

3.4. Évaluation par simulations

Afin de comparer nos heuristiques, nous avons procédé comme précédemment. Nous avons exécuté nos heuristiques sur des graphes d'application et de plate-forme générés aléatoirement et simulé l'exécution des ordonnancements produits.

Génération des tests

Nous avons testé nos heuristiques sur différents types de plates-formes. Les graphes de plate-forme sont composés de 7 serveurs. Le graphe d'interconnexion entre les serveurs est un anneau, un arbre aléatoire ou une clique (nous définissons deux types de cliques, avec des politiques de routage différentes [9]). Les grappes sont composées de 8, 16 ou 32 processeurs hétérogènes. Les bandes passantes des liens de communication et les vitesses des processeurs ont été tirés aléatoirement, de la même manière que précédemment. Comme dans la section 2.4, nous gardons un contrôle sur le ratio des coûts de communication et de calcul.

Nous avons utilisés les mêmes types de graphes d'application qu'en section 2.4. Chaque graphe contient 1 500 tâches et 1 750 données, sauf pour les graphes en étoile qui contiennent également 1 500 tâches mais seulement 70 données. La distribution initiale des données est faite de manière aléatoire.

Résultats

Dans le tableau 4, nous présentons une sélection des heuristiques, en fonction de leur performance et de leur coût. Dans le tableau, nous notons par *insert* quand les communications sont ordonnancées par insertion. Nous introduisons également l'heuristique naïve *randomtask* qui choisit la tâche candidate (la même pour tous les serveurs) de manière aléatoire, mais qui utilise ensuite les mêmes optimisations et schémas d'ordonnancement que les autres heuristiques.

Nous avons commencé par comparer toutes les heuristiques sur des plates-formes composées de 7 serveurs. Pas moins de 48 000 tests aléatoires ont ainsi été effectués (1 000 tests sur les quatre types de graphes d'application, les quatre types de graphe de plate-forme et les trois ratios des coûts de communication et de calcul). Chaque test concerne 81 heuristiques. Nous avons ensuite comparé nos meilleures heuristiques sur des plates-formes plus grandes pour estimer l'impact de la taille de la plate-forme et pour étudier comment nos résultats passent à l'échelle. Comme dans la section 2.4, nous utilisons les performances et les coûts *relatifs* pour comparer les heuristiques. - Nous pouvons voir que, exceptés *min-min* et *sufferage* qui deviennent rapidement trop chères, un bon choix est d'utiliser *dynamic1+mct*

TAB. 4 – Résumé des meilleures heuristiques suivant leur performance et leur coût, pour trois tailles de plates-formes simulées. Chaque valeur est une moyenne de 48 000 configurations pour les plates-formes à 7 serveurs, de 2 400 configurations pour les plates-formes à 50 serveurs et de 1 056 configurations pour les plates-formes à 100 serveurs.

	7 serveurs		50 serveurs		100 serveurs	
Heuristique	Performance	Coût	Performance	Coût	Performance	Coût
sufferage+insert	1,06 (±10%)	21 984 (±99%)				
min-min+insert	1,07 (±7%)	18 002 (±104%)				
dynamic1 +mct+insert	1,10 (±9%)	19 (±72%)	1,05 (±9%)	873 (±117%)		
randomtask +mct+insert	1,10 (±10%)	14 (±67%)	1,05 (±8%)	1 356 (±204%)		
static+readiness +mct+insert	1,19 (±13%)	13 (±65%)	1,17 (±14%)	843 (±184%)		
dynamic1+mct	1,34 (±17%)	14 (±79%)	1,62 (±34%)	299 (±75%)	1,04 (±8%)	568 (±80%)
randomtask+mct	1,37 (±22%)	7 (±80%)	1,84 (±43%)	11 (±48%)	1,16 (±16%)	18 (±67%)
static +readiness+mct	1,48 (±24%)	7 (±92%)	1,88 (±40%)	10 (±53%)	1,19 (±21%)	17 (±63%)
static+readiness	2,19 (±33%)	3 (±104%)	4,37 (±61%)	1 (±41%)		
static	2,29 (±34%)	2 (±107%)				
randomtask	130 (±318%)	2 (±92%)	1 377 (±419%)	1 (±12%)		

(ou même *randomtask+mct*). Si on peut en supporter le supplément de coût, l'ordonnancement des communications, avec un schéma d'ordonnancement par insertion, améliore grandement la qualité des ordonnancements produits. Sur les plates-formes à 7 serveurs, pour 1 500 tâches, *min-min* prend en moyenne 38 secondes pour construire un ordonnancement, et *dynamic1+mct* seulement 0,1 seconde. Sur les plates-formes à 50 serveurs, pour 25 000 tâches, *dynamic1+mct* construit un ordonnancement en moyenne en 511 secondes, alors que cela coûterait *a priori* plus de 48 heures à *min-min*.

4. Travaux connexes

Des problèmes similaires au cas avec un serveur ont déjà été étudiés par différents auteurs. Dans le cadre de fouilles de données par des algorithmes K-means, da Silva, Carvalho et Hruschka [7] proposent de regrouper les tâches destinées à un même esclave, dans le but d'améliorer le passage à l'échelle de telles applications. Dans un autre domaine, Darling, Carey et Feng ont présenté, avec mpiBLAST [8], une parallélisation de BLAST, un outil servant à la recherche de séquences ADN dans des bases. L'allocation des tâches est faite de manière gloutonne et on retrouve des idées proches de notre variante *locality* pour limiter la duplication des bases de données. Une heuristique que l'on pourrait comparer aux nôtres a été récemment décrite par Santos-Neto, Cirne, Brasileiro et Lima [17]. Dans leur heuristique *storage affinity*, les tâches sont allouées aux esclaves en ne considérant que le volume des données déjà disponibles sur l'esclave. Une stratégie de réplication des tâches est utilisée pour équilibrer la charge à la fin de l'ordonnancement. Il réussissent ainsi à obtenir des performances similaires à celles de *sufferage X*.
Quelques auteurs ont déjà cherché à ordonnancer des tâches sur des plates-formes distribuées, en tenant compte de la répartition des données. Les heuristiques *min-min* et *max-min* ont été utilisées par Alhusaini, Prasanna et Raghavendra [1] pour ordonnancer, niveau par niveau, des DAG dont les tâches peuvent avoir besoin de données réparties dans des entrepôts. Dans le cadre du projet Nile, pour ordonnancer des applications en physique des particules, Amoroso, Marzullo et Ricciardi [2] ont procédé par recherche de flots maximums dans un graphe pour allouer les tâches aux processeurs en tenant compte de la disponibilité des données et des capacités du réseau. Ranganathan et Foster [16] proposent quant à eux d'utiliser une réplication *a priori* des données en fonction, notamment, de la popularité des données et de la charge des serveurs. Des politiques simples d'allocation des tâches sont expérimentées, en particulier *JobDataPresent* qui est dans l'idée proche de notre politique *readiness*. Plus récemment, Weng et Lu [18] ont proposé une heuristique *Qsufferage* dans le but de prendre en compte, en plus du *makespan*, le temps de réponse du système aux demandes d'ordonnancement. Contrairement à nous, pour l'ensemble de ces travaux, si une donnée peut être utilisée par plusieurs tâches, une tâche ne dépend que d'une seule donnée.

5. Conclusion

Nous avons étudié le problème de l'ordonnancement sur plates-formes hétérogènes de tâches partageant des données. Nous nous sommes, dans un premier temps, limité à des plates-formes centralisées de type maître-esclave. Nous avons ensuite généralisé le modèle de plate-forme à un ensemble décentralisé de serveurs reliés entre eux par un réseau d'interconnexion quelconque. D'un point de vue théorique, nous avons étudié la complexité de différentes instances particulières du problème, le cas général étant NP-complet. D'un point de vue pratique, nous avons conçu plusieurs nouvelles heuristiques pour résoudre le problème d'ordonnancement. De nombreuses simulations nous ont permis de montrer que ces nouvelles heuristiques réussissent à obtenir des performances aussi bonne que des heuristiques classiques comme *min-min* ou *sufferage*, tout en ayant une complexité algorithmique moindre.
Nous prévoyons de poursuivre ce travail en utilisant maintenant des approches plus globales, à base de partitionnement de graphe. Il serait également intéressant d'étudier la robustesse de nos heuristiques, face à des erreurs de prédiction.

Bibliographie

1. Alhusaini (Ammar H.), Prasanna (Viktor K.) et Raghavendra (Cauligi S.). – A unified resource scheduling framework for heterogeneous computing environments. Dans *8th Heterogeneous Computing Workshop (HCW'1999)*, p. 156–165. – IEEE Computer Society Press, avril 1999.

2. Amoroso (Alessandro), Marzullo (Keith) et Ricciardi (Aleta). – Wide-area Nile: A case study of a wide-area data-parallel application. Dans *18th International Conference on Distributed Computing Systems (ICDCS'98)*, p. 506–515. – IEEE Computer Society Press, mai 1998.
3. Berman (Francine). – High-performance schedulers. Dans *The Grid: Blueprint for a New Computing Infrastructure*, éd. par Foster (Ian) et Kesselman (Carl), chap. 12, p. 279–309. – Morgan Kaufmann Publishers, 1999.
4. Berman (Francine), Wolski (Richard), Casanova (Henri), Cirne (Walfredo), Dail (Holly), Faerman (Marcio), Figueira (Silvia), Hayes (Jim), Obertelli (Graziano), Schopf (Jennifer), Shao (Gary), Smallen (Shava), Spring (Neil T.), Su (Alan) et Zagorodnov (Dmitrii). – Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, tome 14, n° 4, p. 369–382, avril 2003.
5. Casanova (Henri), Legrand (Arnaud), Zagorodnov (Dmitrii) et Berman (Francine). – Using simulation to evaluate scheduling heuristics for a class of applications in Grid environments. – Rapport de recherche n° 1999-46, LIP, École Normale Supérieure de Lyon, France, septembre 1999.
6. Casanova (Henri), Legrand (Arnaud), Zagorodnov (Dmitrii) et Berman (Francine). – Heuristics for scheduling parameter sweep applications in Grid environments. Dans *9th Heterogeneous Computing Workshop (HCW'2000)*, p. 349–363. – IEEE Computer Society Press, mai 2000.
7. da Silva (Fabrício Alves Barbosa), Carvalho (Sílvia) et Hruschka (Eduardo Raul). – A scheduling algorithm for running bag-of-tasks data mining applications on the grid. Dans *Euro-Par 2004, Parallel Processing, 10th International Euro-Par Conference*, tome 3149 de *Lecture Notes in Computer Science*, p. 254–262. – Springer-Verlag, septembre 2004.
8. Darling (Aaron E.), Carey (Lucas) et Feng (Wu-chun). – The design, implementation, and evaluation of mpiBLAST. Dans *4th International Conference on Linux Clusters: The HPC Revolution 2003 (LCI 03)*. – Linux Cluster Institute, juin 2003.
9. Giersch (Arnaud). – *Ordonnancement sur plates-formes hétérogènes de tâches partageant des données*. – Thèse de doctorat, Université Louis Pasteur, Strasbourg, France, décembre 2004.
10. Giersch (Arnaud), Robert (Yves) et Vivien (Frédéric). – Scheduling tasks sharing files from distributed repositories. Dans *Euro-Par 2004, Parallel Processing, 10th International Euro-Par Conference*, tome 3149 de *Lecture Notes in Computer Science*, p. 246–253. – Springer-Verlag, septembre 2004.
11. Giersch (Arnaud), Robert (Yves) et Vivien (Frédéric). – Scheduling tasks sharing files on heterogeneous master-slave platforms. Dans *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, p. 364–371. – IEEE Computer Society Press, février 2004.
12. Giersch (Arnaud), Robert (Yves) et Vivien (Frédéric). – Scheduling tasks sharing files on heterogeneous master-slave platforms. *Journal of Systems Architecture*, 2005. – À paraître.
13. Johnson (S. M.). – Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, tome 1, n° 1, p. 61–68, mars 1954.
14. Li (Yaohang) et Mascagni (Michael). – Grid-based Monte Carlo application. Dans *3rd International Workshop on Grid Computing (GRID 2002)*, tome 2536 de *Lecture Notes in Computer Science*, p. 13–24. – Springer-Verlag, novembre 2002.
15. Maheswaran (Muthucumaru), Ali (Shoukat), Siegel (Howard Jay), Hensgen (Debra) et Freund (Richard F.). – Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, tome 59, n° 2, p. 107–131, novembre 1999.
16. Ranganathan (Kavitha) et Foster (Ian). – Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, tome 1, n° 1, p. 53–62, 2003.
17. Santos-Neto (Elizeu), Cirne (Walfredo), Brasileiro (Francisco) et Lima (Aliandro). – Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. Dans *10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2004)*, Lecture Notes in Computer Science. – Springer-Verlag, juin 2004.
18. Weng (Chuliang) et Lu (Xinda). – Heuristic scheduling for bag-of-tasks applications in combination with QoS in the computation grid. *Future Generation Computer Systems*, tome 21, n° 2, p. 271–280, février 2005.