

Ordonnancement sur plates-formes hétérogènes de tâches partageant des données

Arnaud Giersch

▶ **To cite this version:**

Arnaud Giersch. Ordonnancement sur plates-formes hétérogènes de tâches partageant des données. Réseaux et télécommunications [cs.NI]. Université Louis Pasteur - Strasbourg I, 2004. Français. NNT : . tel-00008222

HAL Id: tel-00008222
https://theses.hal.science/tel-00008222

Submitted on 21 Jan 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ LOUIS PASTEUR STRASBOURG

Laboratoire des Sciences
de l'Image, de l'Informatique et
de la Télédétection

UMR CNRS-ULP 7005

Équipe Image et
Calcul Parallèle Scientifique

Thèse présentée pour obtenir le grade de
Docteur de l'Université Louis Pasteur
Strasbourg I

Discipline : informatique

par Arnaud Giersch

Titre

Ordonnancement sur plates-formes hétérogènes de tâches partageant des données

Soutenue publiquement le 22 décembre 2004

Membres du jury

Directeur : M. Guy-René Perrin, professeur
Université Louis Pasteur de Strasbourg

Rapporteur interne : M. Jean-Jacques Pansiot, professeur
Université Louis Pasteur de Strasbourg

Rapporteurs externes : M. Denis Trystram, professeur
Institut national polytechnique de Grenoble
M. Henri Casanova, Research Scientist
University of California, San Diego

Examinateur : M. Hervé Guyennet, professeur
Université de Franche-Comté

Co-encadrants (membres invités) : M. Stéphane Genaud, maître de conférences
Université Robert Schuman de Strasbourg
M. Frédéric Vivien, chargé de recherche Inria
École normale supérieure de Lyon

Ce document a été composé avec LaTeX.

À Fatima,

Gabriel, Brahim et Joachim.

Remerciements

Alors Papa, tu as fini ta thèse ?
Gabriel, juillet–décembre 2004.

Eh oui, j'ai finalement terminé cette thèse. Je voudrais maintenant remercier tous ceux qui m'ont accompagné, soutenu et aidé pendant ces quatre années.

D'abord Guy-René Perrin, mon directeur de thèse, qui a su m'aider à me réorienter lorsque, au bout de la première année, je ne savais plus trop où j'allais.

Ensuite Stéphane, qui a alors accepté d'encadrer mon travail. C'est grâce à lui, et au projet TAG, que je me suis mis à jouer avec des problèmes de grilles, ce qui a finalement aboutit à cette thèse.

Puis Frédéric qui s'est joint a nous, même s'il a fallu attendre qu'il s'éloigne d'environ 486 km [1] avant de commencer à faire de la recherche ensemble ! C'est vrai que ça aurait été trop facile et moins drôle en partageant le même bureau...

Et enfin Yves, qui a accepté de consacrer de son temps pour travailler avec moi, alors qu'on ne se connaissait même pas – sauf par l'intermédiaire de Frédéric !

Merci à vous qui, par votre disponibilité, vos remarques, vos conseils et votre aide, m'ont permis de mener à bien ce travail.

Je voudrais également remercier ceux qui se sont déplacé en Alsace dans la froidure d'un 22 décembre (−10 °C), en particulier mes rapporteurs : Henri Casanova, Jean-Jacques Pansiot et Denis Trystram, ainsi que Hervé Guyennet qui a présidé mon jury de thèse.

Merci à Benjamin (Ouaf ?), Benoît, Catherine, Guillaume, Philippe, Romaric (Plop !), Vincent et – toujours par ordre alphabétique – les autres membres de l'ICPS avec qui j'ai partagé bureau, repas, cafés, canoës, discussions plus ou moins scientifiques, etc., et qui ont contribué à rendre notre cadre de travail si agréable.

Merci Fatima. Ta présence quotidienne, ton soutien et ton aide (et tout le reste aussi) me sont précieux. Merci Gabriel, Brahim et Joachim, votre présence et votre joie de vivre me sont tout autant précieux.

1. Distance approximative, par la route, entre le Pôle API à Illkirch et l'ENS à Lyon.

Je ne manquerai pas bien sûr de remercier également mes parents, famille et amis qui m'accompagnent depuis de nombreuses années.

Je tiens enfin à remercier le laboratoire LIP et le CINES qui m'ont permis d'utiliser leurs gros ordinateurs, sans lesquels je n'aurais pas pu mener toutes mes expériences et simulations. Merci aussi à tous les acteurs du logiciel libre car c'est aussi un peu grâce à eux que ce travail a pu être effectué.

Finalement, Merci à toutes celles et tous ceux (sans compter les autres) que j'aurais pu oublier de citer.

Table des matières

1 Introduction **1**

2 Maître-esclave **5**
2.1 Introduction 5
2.2 Motivation 6
2.2.1 Exemple d'application en tomographie sismique 6
2.2.2 Implémentation 8
2.2.3 Modèle de communication 8
2.3 Équilibrage statique 10
2.3.1 Modèles 10
2.3.2 Une solution exacte 11
2.3.3 Une heuristique garantie 15
2.3.4 Choix du processeur maître 17
2.4 Résolution par tâches divisibles 17
2.4.1 Durée d'exécution 18
2.4.2 Terminaisons simultanées 19
2.4.3 Politique pour l'ordre des processeurs 20
2.4.4 Conséquences pour le cas général 24
2.5 Validation expérimentale 26
2.5.1 Environnement 26
2.5.2 Résultats 27
2.6 Travaux connexes 32
2.7 Conclusion 33

3 Avec des données partagées **35**
3.1 Introduction 35
3.2 Modèles 37
3.2.1 Tâches et données 37
3.2.2 Graphe de plate-forme 38
3.2.3 Fonction objectif 39
3.3 Complexité 40

3.3.1 Avec un seul esclave . . . 40
3.3.2 Avec plusieurs esclaves . . . 45
3.4 Heuristiques . . . 50
3.4.1 Heuristiques de référence . . . 50
3.4.2 Structure des nouvelles heuristiques . . . 52
3.4.3 Les fonctions objectifs . . . 53
3.4.4 Politiques additionnelles . . . 55
3.4.5 Complexité . . . 56
3.5 Évaluation par simulations . . . 57
3.5.1 Plates-formes simulées . . . 58
3.5.2 Graphes d'application . . . 59
3.5.3 Résultats . . . 60
3.6 Conclusion . . . 66

4 Avec plusieurs serveurs 69
4.1 Introduction . . . 69
4.2 Modèles . . . 70
4.2.1 Tâches et données . . . 70
4.2.2 Graphe de plate-forme . . . 70
4.2.3 Fonction objectif . . . 72
4.2.4 Déroulement de l'exemple . . . 73
4.2.5 Discussion du modèle . . . 75
4.3 Complexité . . . 77
4.4 Adaptation du *min-min* . . . 84
4.4.1 Principe du *min-min* . . . 84
4.4.2 Ordonnancement des communications . . . 85
4.4.3 Le *min-min* adapté . . . 90
4.5 Heuristiques moins coûteuses . . . 92
4.5.1 Heuristiques statiques . . . 92
4.5.2 Variantes des heuristiques statiques . . . 94
4.5.3 Heuristiques dynamiques . . . 98
4.6 Évaluation par simulations . . . 102
4.6.1 Plates-formes simulées . . . 102
4.6.2 Graphes d'application . . . 103
4.6.3 Résultats . . . 104
4.7 Conclusion . . . 110

5 Conclusion et perspectives 113
5.1 Conclusion . . . 113
5.2 Perspectives . . . 114

Bibliographie **119**

A Notations **129**

B Résultats expérimentaux pour le chapitre 3 **131**

C Résultats expérimentaux pour le chapitre 4 **137**

D Autre résultat de complexité **155**

E Liste des publications **159**

Chapitre 1

Introduction

Les besoins en puissance de calcul informatique sont de plus en plus importants dans de nombreux domaines comme le calcul scientifique, la modélisation, la simulation, la fouille de données ou bien encore la réalité virtuelle. Le parallélisme est une solution toujours d'actualité pour répondre à ces besoins toujours croissants. C'est dans ce but qu'ont été conçus les supercalculateurs, des machines composés de plusieurs centaines, voire milliers, de processeurs reliés par un réseau d'interconnexion rapide. Ce type de machine a connu son heure de gloire jusque dans les années quatre-vingt-dix. C'est alors que sont apparues les grappes de stations (ou *cluster*), bien moins coûteuses, mais offrant des performances qui peuvent être comparables à celles des supercalculateurs. Ces grappes sont constituées d'un ensemble de stations peu chères du commerce, connectées par un réseau plus ou moins rapide. Cette évolution a été possible par la démocratisation de l'informatique et par la disponibilité de systèmes comme Linux et de bibliothèques comme PVM ou MPI. Ces grappes de calcul sont cependant plus difficiles à programmer que les supercalculateurs classiques.

Actuellement, ces différents types de machines cohabitent et le parc des universités et des entreprises est souvent très hétérogène, comprenant des supercalculateurs, des grappes de calcul et des stations de travail personnelles. Ces différentes machines ne suffisent cependant pas toujours pour résoudre des problèmes de plus en plus complexes. Les réseaux étant de plus en plus rapides, la tendance actuelle en matière de calcul distribué est de chercher à fédérer un ensemble de ces machines, réparties à l'échelle d'un continent, voire de la planète entière, afin d'en agréger les puissances de calcul. C'est la fameuse grille de calcul (*computing grid*) décrite dans le livre de Foster et Kesselman [37]. Ce nouveau type de plate-forme de calcul est de nature très hétérogène, que ce soit au niveau des ressources de calcul (processeurs) ou au niveau des capacités de communication (réseau). La prise en compte de cette hétérogénéité est donc un enjeu majeur pour l'utilisation efficace des plates-formes d'aujourd'hui et de demain.

La résolution de problèmes de plus en plus complexes implique l'utilisation de données très volumineuses. Le transfert des données nécessaires aux calculs est donc un paramètre important à prendre en compte. Dans un certains nombre de problèmes, comme par exemple la fouille de données ou les simulations par Monte-Carlo multiples, les mêmes données sont réutilisées pour différents calculs, on dit alors qu'elles sont *partagées*. Il est alors possible d'optimiser l'exécution de telles applications en prêtant une attention particulière à ce partage éventuel des données. Ainsi, c'est de cette prise en compte du partage des données, lors de l'ordonnancement d'un ensemble de calculs sur des plates-formes hétérogènes comme les grilles de calcul, dont il est question dans cette thèse.

Notre travail s'articule en trois parties, organisées par complexité croissante. Dans la première partie (chapitre 2), nous cherchons à équilibrer l'exécution d'applications de type maître-esclave où un ensemble de données de même taille est à distribuer. Une tâche (calcul) est à effectuer pour chacune des données, toutes les tâches ayant le même coût. Des algorithmes pour résoudre le problème de manière optimale, ainsi qu'une heuristique garantie sont développés. Une politique concernant l'ordre dans lequel les processeurs (esclaves) sont considérés est également proposée. Des résultats expérimentaux, avec une application scientifique réelle, viennent illustrer les gains obtenus par nos solutions.

Le partage des données est introduit dans la deuxième partie (chapitre 3). De plus, nous étendons le problème au cas où les données peuvent être de tailles différentes et où les tâches peuvent être de durées différentes. La plate-forme reste de type maître-esclave. En passant en revue la complexité du problème, nous montrons deux nouveaux résultats de NP-complétude. Nous développons ensuite plusieurs nouvelles heuristiques pour résoudre le problème de l'ordonnancement des tâches sur les processeurs. Ces nouvelles heuristiques, ainsi que des heuristiques classiques (qui seront nos heuristiques de référence) sont finalement comparées entre elles à l'aide de nombreuses simulations. Nous montrons ainsi que nos heuristiques réussissent à obtenir des performances équivalentes à celles des heuristiques de référence, tout en ayant une complexité algorithmique d'un ordre de grandeur plus faible.

Dans la dernière partie (chapitre 4), nous généralisons le modèle de plate-forme à un ensemble décentralisé de serveurs reliés entre eux par un réseau d'interconnexion quelconque. Nous montrons alors que le simple fait d'ordonnancer les transferts de données est un problème difficile. Nous présentons ensuite une méthode pour adapter les heuristiques de référence au cas des serveurs distribués. De la même manière qu'au chapitre précédent, nous développons un ensemble d'heuristiques moins coûteuses. Toutes les heuristiques sont comparées entre elles à l'aide de simulations intensives

Nous concluons finalement dans le chapitre 5 et nous exposons plusieurs perspectives de poursuite de ce travail.

Chapitre 2

Maître-esclave

2.1 Introduction

Le paradigme maître-esclave est une technique classique de parallélisation. Dans ce modèle, le travail à effectuer est divisé en un ensemble de tâches indépendantes. Ces tâches sont distribuées sur un ensemble de processeurs appelés *esclaves*, sous le contrôle d'un processeur particulier, le *maître*. Cette technique de parallélisation est simple et en général efficace en environnement homogène. En environnement hétérogène les processeurs peuvent être différents et donc avoir des puissances de calculs différentes. Les vitesses de transfert sur les liens d'interconnexion entre le maître et les esclaves peuvent, elles aussi, être hétérogènes. Cette non-uniformité de la plate-forme doit être prise en compte pour pouvoir utiliser au mieux les ressources disponibles.

C'est le problème de l'adaptation à des plates-formes hétérogènes d'une application scientifique originalement écrite pour machine parallèle homogène qui a motivé notre travail. Comme nous partions d'un code existant, nous avons cherché des solutions minimisant l'importance des réécritures dans le code original. Notre application prend un ensemble de données en entrée. À chaque donnée de cet ensemble correspond un calcul qui peut être fait indépendamment des autres données. Chacun de ces calculs constitue ainsi une tâche. L'ensemble des données est réparti entre les esclaves à l'aide d'une opération de distribution (*scatter*). L'opération de distribution est souvent implémentée dans les bibliothèques de passage de messages. La bibliothèque MPI [73], par exemple, fournit la primitive `MPI_Scatter` qui permet au programmeur de répartir les données de manière équitable (en nombre) parmi les processeurs d'un communicateur MPI. Une fois les données distribuées, les esclaves traitent les tâches qui correspondent aux données qu'ils ont reçues.

La méthode permettant d'améliorer les performances en environnement hétéro-

gène qui est la moins intrusive, est celle qui consiste à utiliser une bibliothèque de communication adaptée à l'hétérogénéité. Ainsi beaucoup de travaux ont été consacrés à ce but : pour MPI, de nombreux projets comme MagPIe [55], MPI-StarT [50] et MPICH-G2 [54] visent à améliorer les performances des communications en présence de réseaux hétérogènes. L'essentiel du gain est obtenu en modifiant les opérations de communication collectives. Par exemple, MPICH-G2 se comporte souvent mieux que MPICH pour diffuser des informations d'un processeur à plusieurs autres (*broadcast*) en choisissant un algorithme adapté suivant les latences du réseau [53]. De telles bibliothèques de communication apportent des résultats intéressants par rapport aux bibliothèques standards, mais ces améliorations ne sont souvent pas suffisantes pour obtenir des performances acceptables quand les processeurs sont également hétérogènes. Il faut alors équilibrer la charge de calcul entre les processeurs pour pouvoir réellement tirer parti de ces environnements.

Dans les codes comme celui qui nous intéresse, les données sont indépendantes et l'opération de distribution sert à démarrer une section de calcul SPMD sur les esclaves. La charge de travail sur les processeurs dépend directement de la quantité de données reçues. L'opération de distribution peut donc être utilisée pour répartir la charge de calcul. MPI fournit la primitive `MPI_Scatterv` qui permet de distribuer des parts inégales de données aux processeurs. Il est donc *a priori* possible de remplacer le `MPI_Scatter` par un `MPI_Scatterv` correctement paramétré pour accroître de façon significative les performances de l'application à faible coût. En terme de réécriture de code, une telle transformation ne demande pas de réorganisation profonde du programme. Notre problème est donc de répartir la charge entre les processeurs en calculant une distribution des données dépendant des vitesses des processeurs et des bandes passantes des liens réseau.

Dans la section 2.2, nous présentons l'application cible, une application scientifique de géophysique, dont nous cherchons à optimiser l'exécution. Nous présentons ensuite, dans la section 2.3, nos techniques d'équilibrage de charge. Dans la section 2.4, nous résolvons le problème dans le cadre des tâches divisibles. Cette étude nous permet alors de dériver une politique pour l'ordre des processeurs. Dans la section 2.5, nous présentons des résultats d'expériences mettant en œuvre les solutions proposées puis, dans la section 2.6, nous faisons un tour d'horizon des travaux préexistants avant de conclure en section 2.7.

2.2 Motivation

2.2.1 Exemple d'application en tomographie sismique

En tomographie sismique, les géophysiciens cherchent à modéliser la structure interne de la Terre à partir des informations recueillies lors d'évènements sismiques

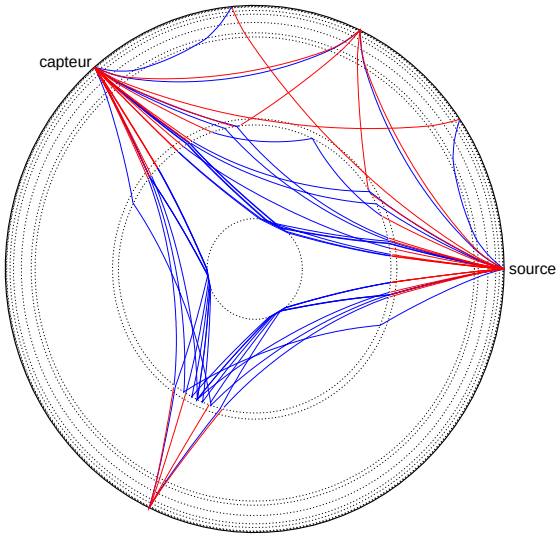


Figure 2.1 – Visualisation du tracé de 27 rais sismiques entre une source et un capteur.

(tremblements de terre, essais nucléaires, etc.). Une manière de procéder est de simuler la propagation d'ondes sismiques à travers un modèle de la Terre afin de vérifier l'adéquation du modèle avec la réalité, puis de raffiner le modèle.

L'application que nous considérons, développée par Grunberg [43], cherche à modéliser les vitesses de propagation des ondes sismiques à l'intérieur du globe terrestre. Les différentes vitesses trouvées aux différents points discrétisés par le modèle (en général un maillage) reflètent les propriétés physiques des roches à ces endroits. Les vitesses des ondes sismiques sont calculées à partir de sismogrammes enregistrés par différentes stations sismologiques couvrant la surface de la planète. Ces sismogrammes sont analysés pour déterminer les temps d'arrivée des différentes ondes sismiques, leur type, ainsi que pour calculer la localisation des foyers des séismes. Des organisations internationales, comme l'ISC (International Seismic Center) conservent l'ensemble de ces informations dans des bases de données qui peuvent contenir plusieurs millions d'éléments.

À partir de ces données, une application de tomographie reconstruit la propagation des ondes sismiques en utilisant un modèle de vitesses initial. La propagation d'une onde de son foyer à un capteur donné définit un chemin (ou *rai*) que l'application trace avec les données du modèle de vitesses initial. La figure 2.1 montre les chemins (entre une source et un capteur) pour différents types d'ondes. Le temps de propagation de l'onde le long du chemin reconstruit est alors comparé avec le temps de propagation réel. Un nouveau modèle de vitesses est ensuite calculé pour minimiser les différences entre la simulation et la réalité. Ce procédé est plus précis si le nouveau modèle est calculé à partir de nombreux chemins passant en

beaucoup d'endroits à l'intérieur de la Terre, ce qui demande des calculs intensifs.

2.2.2 Implémentation

Nous allons maintenant sommairement décrire comment l'application étudiée exploite le parallélisme potentiel des calculs, et comment les tâches sont distribuées aux processeurs. La parallélisation est expliquée en détails dans [43]. Les données d'entrées sont les caractéristiques d'un ensemble d'ondes sismiques. Chaque onde est décrite par son type et par une paire de coordonnées 3D (les coordonnées de la source du séisme et celles du capteur ayant enregistré l'onde). Avec ces caractéristiques, une onde peut être modélisée par un ensemble de rais représentant le front de propagation de l'onde. Les caractéristiques de l'onde sismique sont suffisantes pour permettre de tracer l'ensemble des rais associés et les différents rais peuvent être tracés indépendamment.

La parallélisation originale de l'application avait été pensée pour un ensemble homogène de processeurs (comme une machine parallèle par exemple) avec un processus MPI par processeur. Les différents processeurs se partagent l'ensemble des rais à tracer. Le pseudo-code suivant illustre la phase de communication et de calcul :

```
if (rank = ROOT)
    raydata ← lire n lignes du fichier de données ;
MPI_Scatter(raydata, n/P,. . .,rbuff,. . .,ROOT,MPI_COMM_WORLD) ;
calculer(rbuff) ;
```

où P est le nombre de processeurs utilisés et n le nombre de données. Un processeur *maître* (identifié par `ROOT`) commence par répartir toutes les données de `raydata` équitablement entre des *esclaves* par une opération de distribution `MPI_Scatter`. Chaque esclave effectue sa part de travail après avoir reçu ses n/P données dans `rbuff`. Notre but est d'optimiser la répartition des données de manière à terminer l'ensemble des calculs le plus tôt possible. La figure 2.2 montre une exécution possible de l'opération de communication, avec P_4 comme processeur maître.

2.2.3 Modèle de communication

La figure 2.2 montre le comportement de l'opération de distribution tel qu'il a été observé en exécutant l'application sur notre grille de test (décrite dans la section 2.5.1). Dans l'implémentation de MPICH-G2 utilisée (v1.2.2.3), pour effectuer une opération de distribution, le processeur maître (P_4 sur le dessin) doit avoir entièrement envoyé un message avant de pouvoir en envoyer un autre. De plus, il n'y a pas de recouvrement entre les calculs et les communications : un processeur doit avoir entièrement terminé ses communications avant de pouvoir commencer ses calculs. Ce comportement correspond au modèle habituellement appelé *un-port*,

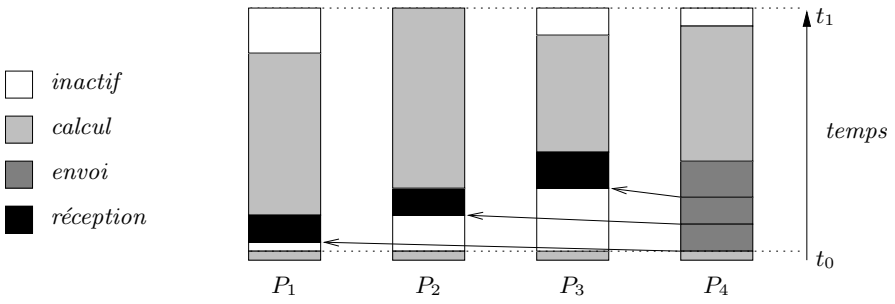


Figure 2.2 – Une opération de distribution suivie d'une phase de calcul.

c'est-à-dire qu'à un instant donné, un processeur peut être impliqué dans au plus deux communications : une émission et une réception. Comme le processeur maître envoie les données aux autres processeurs les uns après les autres, un esclave ne commence effectivement sa communication qu'après que tous les processeurs précédents ont été servis. Cela produit « l'effet d'escalier » représenté sur la figure 2.2 par les temps de fin des réceptions (rectangles noirs). Avec cette implémentation l'ordre des processeurs suit l'ordre du rang MPI, ce qui nous permettra plus tard d'avoir un contrôle sur cet ordre (cf. section 2.5).

Comme l'ont noté Yang et Casanova [95, section 6.4], le modèle un-port est un modèle courant mais, dans certains cas, le maître pourrait obtenir un meilleur débit agrégé en envoyant les données à plusieurs esclaves simultanément. L'utilisation de connexions TCP multiples est en effet une technique connue pour augmenter le débit réseau apparent [44, 2, 23]. Le modèle dans lequel un processeur peut ainsi effectuer plusieurs communications en même temps est communément appelé *multi-port*. Une solution multi-port demanderait cependant une implémentation appropriée de l'opération de distribution, et poserait un problème bien plus compliqué. Par exemple, Saif et Parashar ont observé dans [82] que les primitives de communication non bloquantes de MPI ont recours à un comportement bloquant quand les messages deviennent assez « grands ». De plus, pour exploiter au mieux une solution multi-port, un modèle plus complexe du comportement du réseau serait nécessaire afin de pouvoir prédire les performances des communications. Un tel modèle devrait être paramétré par des caractéristiques du réseau comme la topologie et les performances des différents liens. Il faudrait donc être capable de découvrir ces caractéristiques, ce qui est un problème difficile en soi [63].

Plutôt que de chercher à résoudre tous ces problèmes avant de chercher à équilibrer la charge des processeurs en autorisant des communications à avoir lieu en parallèle, nous avons préféré nous restreindre au modèle de communication un-port. Cette restriction nous permet également d'utiliser une bibliothèque de communication existante et nous pouvons ainsi optimiser l'exécution d'une application

en paramétrant simplement la distribution des données sur les processeurs. Dans toute la suite du chapitre, nous nous restreignons donc au modèle de communications un-port. Un modèle de plate-forme plus général sera proposé au chapitre 4.

2.3 Équilibrage statique

Dans cette section, nous présentons différentes solutions pour trouver une distribution optimale des données. Après avoir brièvement présenté le cadre dans lequel nous nous plaçons, nous donnons deux algorithmes par programmation dynamique, le deuxième étant plus efficace que le premier, mais nécessitant des hypothèses supplémentaires sur les fonctions de coût. Nous terminons en présentant une heuristique garantie qui peut être utilisée pour trouver rapidement, par programmation linéaire, une très bonne approximation dans le cas où les fonctions de coût sont affines.

Nous faisons l'hypothèse que les caractéristiques de la plate-forme ne changent pas durant l'exécution d'une phase de communication et de calculs. Nous ne considérons donc que l'équilibrage de charge statique. Une application effectuant plusieurs opérations de distribution pourrait cependant s'adapter à des variations des caractéristiques de la plate-forme pour chacune des distributions. Un système de mesure (comme par exemple NWS [94]) peut être interrogé juste avant de calculer la distribution des données pour obtenir les caractéristiques instantanées de la plate-forme.

2.3.1 Modèles

Nous introduisons ici quelques notations, ainsi que le modèle de coût qui sera utilisé pour déduire la distribution optimale des données.

Nous considérons un ensemble de p processeurs : $P_1, \ldots, P_p$. Un processeur P_i est caractérisé par :

- $T_{comm}(i, x)$, le temps nécessaire pour recevoir x données du processeur maître ;
- $T_{calc}(i, x)$, le temps nécessaire pour traiter x données.

Nous voulons traiter au total n données. Nous cherchons donc une distribution $n_1, \ldots, n_p$ de ces données sur les p processeurs minimisant le temps total d'exécution. Les données sont envoyées aux processeurs dans l'ordre : d'abord à P_1, puis à P_2, et ainsi de suite, jusqu'à P_p. Comme le processeur maître ne peut commencer à traiter son lot de données *qu'après* avoir envoyé les autres données à tous les autres processeurs (car il n'y a pas de recouvrement entre les calculs et les communications), il sera toujours le dernier processeur : P_p. Il faut un temps $T_{comm}(i, n_i)$ au processeur maître pour envoyer ses données au processeur P_i. Comme nous nous plaçons dans le modèle un-port, et que le processeur maître envoie les données dans

l'ordre des processeurs, le processeur P_i commence sa communication après que les processeurs $P_1, \ldots, P_{i-1}$ ont été servis, ce qui prend un temps $\sum_{j=i}^{i-1} T_{comm}(j, n_j)$. Le processeur maître prend ensuite un temps $T_{comm}(i, n_i)$ pour envoyer à P_i ses données. Le processeur P_i prend finalement un temps $T_{calc}(i, n_i)$ pour traiter son lot de données. Ainsi, P_i termine son exécution au temps :

$$T_i = \sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i). \tag{2.1}$$

Le temps T, nécessaire pour traiter complètement l'ensemble des n données est donc :

$$T = \max_{1 \leq i \leq p} T_i = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i) \right), \tag{2.2}$$

et nous cherchons une distribution $n_1, \ldots, n_p$ minimisant cette durée.

2.3.2 Une solution exacte

Dans cette section, nous présentons deux algorithmes par programmation dynamique pour calculer une distribution optimale des données. Le premier a pour seules hypothèses que les fonctions de coût sont positives. Le second présente quelques optimisations le rendant beaucoup plus rapide, mais sous l'hypothèse supplémentaire que les fonctions de coût sont croissantes.

Algorithme de base

Nous étudions maintenant l'équation (2.2). Le temps total d'exécution est le maximum entre le temps d'exécution du processeur P_1 et de celui des autres processeurs :

$$T = \max \left(T_{comm}(1, n_1) + T_{calc}(1, n_1), \max_{2 \leq i \leq p} \left(\sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i) \right) \right).$$

Nous pouvons remarquer que tous les termes de cette équation contiennent le temps nécessaire au processeur maître pour envoyer ses données à P_1. Cette équation peut donc être écrite :

$$T = T_{comm}(1, n_1) + \max \left(T_{calc}(1, n_1), \max_{2 \leq i \leq p} \left(\sum_{j=2}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i) \right) \right).$$

Nous constatons donc que le temps nécessaire pour traiter n données sur les processeurs 1 à p est égal au temps pris par le processeur maître pour envoyer n_1 données à P_1, plus le maximum

(i) du temps pris par P_1 pour traiter ses n_1 données ;

(ii) du temps nécessaire aux processeurs 2 à p pour traiter $n - n_1$ données.

Ainsi, en supposant que l'on connaisse, pour tout n_1, $0 \leq n_1 \leq n$, une distribution optimale de $n - n_1$ donnée sur les processeurs 2 à p (ainsi que les temps correspondants), on peut trouver une distribution optimale de n données sur les processeurs 1 à p en parcourant simplement toutes les valeurs possibles pour n_1 et en retenant une valeur minimisant le temps total.

Cela nous conduit à l'algorithme 2.1 (la distribution est exprimée sous forme de liste, d'où l'utilisation du constructeur de liste « cons »). Dans l'algorithme 2.1, $coût[d, i]$ dénote le coût de traitement minimal de d données par les processeurs P_i à P_p. La liste $solution[d, i]$ décrit une distribution de d données sur les processeurs P_i à P_p permettant d'obtenir le temps d'exécution minimal $coût[d, i]$.

L'algorithme 2.1 fonctionne de la manière suivante : il commence, dans la phase d'initialisation (lignes 1 à 6), par calculer le temps de traitement de 1 à n données pour un seul processeur (P_p). Ensuite, dans la boucle principale (ligne 7), il ajoute les processeurs un à un (d'abord P_{p-1}, puis P_{p-2} et ainsi de suite jusqu'à P_1) en cherchant à chaque fois le coût de traitement minimal et une distribution permettant d'atteindre ce coût. Cette recherche (ligne 12) est effectuée pour un nombre de données allant de 1 à n (ligne 10). L'algorithme retourne enfin une distribution optimale de n données sur les p processeurs P_1 à P_p, ainsi que le temps de traitement obtenu avec cette distribution (ligne 22).

L'algorithme 2.1, de par ses trois boucles imbriquées, a une complexité de $O(p \cdot n^2)$ qui peut être prohibitive, mais cet algorithme a pour seules hypothèses que les fonctions $T_{comm}(i, x)$ et $T_{calc}(i, x)$ sont positives, et nulles quand $x = 0$.

Algorithme optimisé

En faisant maintenant les hypothèses supplémentaires (mais qui restent raisonnables) que $T_{comm}(i, x)$ et $T_{calc}(i, x)$ sont croissantes en x, nous pouvons apporter quelques optimisations à l'algorithme. Ces optimisations consistent à réduire les bornes de la boucle interne (boucle en e, lignes 12 à 17 de l'algorithme 2.1). L'algorithme 2.2 (page 14) présente ces optimisations.

Expliquons ce qui a changé entre les deux algorithmes. Pour la suite, souvenons-nous des hypothèses que $T_{comm}(i, x)$ et $T_{calc}(i, x)$ sont croissantes en x. Comme $T_{comm}(i, x)$ et $T_{calc}(i, x)$ sont des fonctions positives, $coût[x, i]$ est aussi croissant, et donc $coût[d - x, i]$ est décroissant en x. Le rôle de la boucle en e est de trouver sol dans l'intervalle $[0 ; d]$ tel que

$$T_{comm}(i, sol) + \max(T_{calc}(i, sol), coût[d - sol, i + 1])$$

soit minimal. Nous cherchons à réduire la borne supérieure de cette boucle, et à augmenter sa borne inférieure.

Algorithme 2.1 – Calcul d'une distribution optimale de n données sur p processeurs.

Entrées : n, p
Sorties : distribution optimale avec son coût
▷ *initialisation : avec un seul processeur (P_p)*
1 $solution[0,p] \leftarrow \text{cons}(0, NIL)$
2 $coût[0,p] \leftarrow 0$
3 **pour** $d \leftarrow 1$ **à** n **faire**
4 $\quad solution[d,p] \leftarrow \text{cons}(d, NIL)$
5 $\quad coût[d,p] \leftarrow T_{comm}(p,d) + T_{calc}(p,d)$
6 **fin**
▷ *ajout des processeurs un à un*
7 **pour** $i \leftarrow p-1$ **à** 1 **par pas de** -1 **faire**
8 $\quad solution[0,i] \leftarrow \text{cons}(0, solution[0,i+1])$
9 $\quad coût[0,i] \leftarrow 0$
10 $\quad$ **pour** $d \leftarrow 1$ **à** n **faire**
$\quad\quad$ ▷ *recherche du minimum*
11 $\quad\quad (sol, min) \leftarrow (0, coût[d,i+1])$
12 $\quad\quad$ **pour** $e \leftarrow 1$ **à** d **faire**
13 $\quad\quad\quad m \leftarrow T_{comm}(i,e) + \max(T_{calc}(i,e), coût[d-e,i+1])$
14 $\quad\quad\quad$ **si** $m < min$ **alors**
15 $\quad\quad\quad\quad (sol, min) \leftarrow (e, m)$
16 $\quad\quad\quad$ **fin**
17 $\quad\quad$ **fin**
18 $\quad\quad solution[d,i] \leftarrow \text{cons}(sol, solution[d-sol, i+1])$
19 $\quad\quad coût[d,i] \leftarrow min$
20 $\quad$ **fin**
21 **fin**
22 **retourner** $(solution[n,1], coût[n,1])$

Soit e_{max} le plus petit entier tel que $T_{calc}(i, e_{max}) \geq coût[d - e_{max}, i+1]$. Pour tout $e \geq e_{max}$, nous avons

$$T_{calc}(i,e) \geq T_{calc}(i,e_{max}) \geq coût[d-e_{max}, i+1] \geq coût[d-e, i+1],$$

donc

$$\min_{e \geq e_{max}} (T_{comm}(i,e) + \max(T_{calc}(i,e), coût[d-e,i+1])) = \min_{e \geq e_{max}} (T_{comm}(i,e) + T_{calc}(i,e)).$$

Comme $T_{comm}(i,e)$ et $T_{calc}(i,e)$ sont toutes les deux croissantes en e,

$$\min_{e \geq e_{max}} (T_{comm}(i,e) + T_{calc}(i,e)) = T_{comm}(i,e_{max}) + T_{calc}(i,e_{max}).$$

Algorithme 2.2 – Calcul d'une distribution optimale de n données sur p processeurs (version optimisée).

```
   Entrées : n, p
   Sorties : distribution optimale avec son coût
   ▷ initialisation : avec un seul processeur (P_p)
 1 solution[0,p] ← cons(0, NIL)
 2 coût[0,p] ← 0
 3 pour d ← 1 à n faire
 4 |   solution[d,p] ← cons(d, NIL)
 5 |   coût[d,p] ← T_comm(p,d) + T_calc(p,d)
 6 fin
   ▷ ajout des processeurs un à un
 7 pour i ← p − 1 à 1 par pas de −1 faire
 8 |   solution[0,i] ← cons(0, solution[0,i+1])
 9 |   coût[0,i] ← 0
10 |   pour d ← 1 à n faire
   |   |   ▷ recherche de e_max par dichotomie
11 |   |   (e_min, e_max) ← (0,d)
12 |   |   e ← ⌊d/2⌋
13 |   |   tant que e ≠ e_min faire
14 |   |   |   si T_calc(i,e) < coût[d−e, i+1] alors
15 |   |   |   |   e_min ← e
16 |   |   |   sinon
17 |   |   |   |   e_max ← e
18 |   |   |   fin
19 |   |   |   e ← ⌊(e_min + e_max)/2⌋
20 |   |   fin
   |   |   ▷ recherche du minimum
21 |   |   (sol, min) ← (e_max, T_comm(i,e_max) + T_calc(i,e_max))
22 |   |   pour e ← sol − 1 à 0 par pas de −1 faire
23 |   |   |   m ← T_comm(i,e) + coût[d−e, i+1]
24 |   |   |   si m < min alors
25 |   |   |   |   (sol, min) ← (e, m)
26 |   |   |   sinon si coût[d−e, i+1] ≥ min alors
   |   |   |   |   ▷ on sort de la boucle
27 |   |   |   |   sortie
28 |   |   |   fin
29 |   |   fin
30 |   |   solution[d,i] ← cons(sol, solution[d−sol, i+1])
31 |   |   coût[d,i] ← min
32 |   fin
33 fin
34 retourner (solution[n,1], coût[n,1])
```

En utilisant une recherche dichotomique pour trouver e_{max} (lignes 11 à 20 de l'algorithme 2.2), nous pouvons réduire la borne supérieure de la boucle en e. Pour tirer parti de cette information, le sens de la boucle doit alors être inversé. De plus, nous savons qu'à l'intérieur de la boucle, $coût[d-e, i+1]$ est toujours plus grand que $T_{calc}(i, e)$, ce qui nous permet d'éviter l'opération max dans le calcul de m (ligne 23).

Nous ne pouvons pas procéder de la même manière pour augmenter la borne inférieure de la boucle. Nous pouvons cependant remarquer que, par l'inversion de la boucle, e est décroissant, et donc que $coût[d-e, i+1]$ est croissant. Si $coût[d-e, i+1]$ devient supérieur ou égal à min, alors pour tout $e' < e$, nous avons $coût[d-e', i+1] \geq coût[d-e, i+1] \geq min$, et comme $T_{comm}(i, x)$ est positive, $T_{comm}(i, e') + coût[d-e', i+1] \geq min$. L'itération peut donc être stoppée, d'où l'instruction de sortie de boucle à la ligne 27.

Dans le pire cas, la complexité de l'algorithme 2.2 est la même que pour l'algorithme 2.1, c'est-à-dire $O(p \cdot n^2)$. Dans le meilleur cas, la boucle de recherche du minimum s'arrête tout de suite, et la complexité de l'algorithme 2.2 est alors de $O(p \cdot n \cdot \log n)$. Nous avons implémenté les deux algorithmes et, en pratique, l'algorithme 2.2 est beaucoup plus efficace (cf. section 2.5).

En dépit de ces optimisations, l'exécution de l'algorithme 2.2 est toujours coûteuse en temps. Une autre approche possible pour réduire le temps d'exécution de ces algorithmes serait d'augmenter la granularité du problème, c'est-à-dire de grouper les données en blocs puis de considérer la répartition de ces blocs parmi les processeurs. Cela conduirait évidemment à une solution non optimale. De plus, comme nous ne faisons que très peu d'hypothèses sur les fonctions de coût, il serait impossible, dans le cas général, de borner l'erreur commise en fonction de la taille des blocs choisie. Dans le cas moins général où les fonctions de coût sont affines, une heuristique plus efficace est maintenant présentée.

2.3.3 Une heuristique garantie

Dans cette section, nous considérons le cas, moins général mais néanmoins réaliste, où les temps de calcul et de communication sont des fonctions affines. Cette nouvelle hypothèse nous permet de coder notre problème comme un programme linéaire. Nous dérivons ensuite une heuristique efficace et garantie de cette formulation en programmation linéaire.

Ainsi, nous faisons l'hypothèse que toutes les fonctions $T_{comm}(i, n)$ et $T_{calc}(i, n)$ sont affines en n, croissantes et positives (pour $n \geq 0$). L'équation (2.2) peut alors

être codée par le programme linéaire suivant :

$$\begin{cases} \text{Minimiser } T \text{ tel que} \\ \forall i \in [1\,;\,p],\ n_i \geq 0, \\ \sum_{i=1}^{p} n_i = n, \\ \forall i \in [1\,;\,p],\ T \geq \sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i). \end{cases} \tag{2.3}$$

Comme nous avons besoin d'une solution entière, nous devrions normalement résoudre en nombres entiers ce programme linéaire. La résolution en nombres entiers est cependant très coûteuse en temps.

Heureusement, il existe une manière de contourner le problème qui produit une bonne approximation : nous pouvons résoudre le système en rationnels pour obtenir une solution rationnelle $n_1, \ldots, n_p$, que nous arrondissons pour obtenir une solution entière $n'_1, \ldots, n'_p$ avec $\sum_i n'_i = n$. Soit T' le temps d'exécution de cette solution entière, T le temps de la solution rationnelle, et T_{opt} le temps de la solution entière optimale. Si pour tout i, $|n_i - n'_i| \leq 1$ (ce qui est aisément accompli par le mode d'arrondi décrit plus bas), alors T' peut être encadré de la manière suivante :

$$T_{opt} \leq T' \leq T_{opt} + \sum_{j=1}^{p} T_{comm}(j, 1) + \max_{1 \leq i \leq p} T_{calc}(i, 1). \tag{2.4}$$

Démonstration. Par l'équation (2.2), on a :

$$T' = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{comm}(j, n'_j) + T_{calc}(i, n'_i) \right). \tag{2.5}$$

Par les hypothèses, $T_{comm}(j, x)$ et $T_{calc}(j, x)$ sont des fonctions affine positives et croissantes. Ainsi,

$$\begin{aligned} T_{comm}(j, n'_j) &= T_{comm}(j, n_j + (n'_j - n_j)) \\ &\leq T_{comm}(j, n_j + |n'_j - n_j|) \\ &\leq T_{comm}(j, n_j) + T_{comm}(j, |n'_j - n_j|) \\ &\leq T_{comm}(j, n_j) + T_{comm}(j, 1) \end{aligned}$$

et nous avons une borne supérieure équivalente pour $T_{calc}(j, n'_j)$. En utilisant ces bornes supérieures pour surestimer l'expression de T' donnée par l'équation (2.5) nous obtenons :

$$T' \leq \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} (T_{comm}(j, n_j) + T_{comm}(j, 1)) + T_{calc}(i, n_i) + T_{calc}(i, 1) \right) \tag{2.6}$$

qui implique l'équation (2.4), sachant que $T_{opt} \leq T'$, $T \leq T_{opt}$, et finalement que $T = \max_{1 \leq i \leq p}(\sum_{j=1}^{i} T_{comm}(j, n_j) + T_{calc}(i, n_i))$. □

Mode d'arrondi

Notre mode d'arrondi est simple : nous commençons par arrondir à l'entier le plus proche le nombre non entier n_i qui est le plus proche d'un entier. Nous obtenons ainsi n'_i et nous faisons une erreur d'approximation de $e = n'_i - n_i$ (avec $|e| < 1$). Si l'erreur e est négative (respectivement positive), alors n_i est sous-estimé (respectivement surestimé) par l'approximation. Ensuite, nous arrondissons à son entier supérieur (respectivement inférieur) le plus proche, un des n_j restants qui est le plus proche de son entier immédiatement supérieur $\lceil n_j \rceil$ (respectivement inférieur $\lfloor n_j \rfloor$). Nous obtenons une nouvelle erreur d'approximation de $e = e + n'_j - n_j$ (avec $|e| < 1$). Nous continuons ainsi jusqu'à ce qu'il ne reste plus qu'un seul des n_i à arrondir : n_k. Nous posons enfin $n'_k = n_k + e$. La distribution $n'_1, \ldots, n'_p$ est donc entière, $\sum_{1 \leq i \leq p} n'_i = d$, et chacun des n'_i diffère de n_i par moins de un.

Remarque. La même méthode d'arrondi a été récemment publiée par Barlas et Veeravalli [10]. Ils se placent dans un cadre un peu plus général : l'architecture de la plate-forme cible peut être de type bus ou arbre, en un-port ou en multi-port. Ils cherchent à arrondir en entiers une solution – en une ou plusieurs tournées – obtenue dans le cadre des tâches divisibles. Ils donnent une majoration de l'erreur commise dans certains cas, et en particulier dans le cas qui nous intéresse : une solution en une tournée avec un arbre à un niveau en un-port. On retrouve alors la même borne que celle qui a été donnée ci-dessus.

2.3.4 Choix du processeur maître

Nous faisons l'hypothèse qu'au départ, les n données à traiter sont stockées sur un ordinateur appelé $\mathcal{C}$. Ce n'est pas forcément un processeur de $\mathcal{C}$ qui est utilisé comme processeur maître. Si le processeur maître n'est pas sur $\mathcal{C}$, alors le temps total d'exécution est égal au temps nécessaire pour transférer les données de $\mathcal{C}$ au processeur maître, plus le temps d'exécution suivant la distribution des données calculée par une des méthodes présentées ci-dessus. Le meilleur processeur maître est alors le processeur minimisant ce temps total d'exécution, s'il est choisi comme maître. Cela revient donc à trouver le minimum parmi les p candidats.

2.4 Résolution par tâches divisibles

Dans cette section, nous étudions une version simplifiée du problème où les fonctions de coût sont linéaires. Cette étude, que nous menons dans le cadre des *tâches divisibles* (*divisible load theory*) [17, 18] va nous permettre de définir une politique concernant l'ordre des processeurs. Une tâche divisible est une tâche qui

peut être arbitrairement répartie sur un nombre quelconque de processeurs. Ce modèle nous permet de résoudre notre problème de manière analytique.

Ainsi, nous faisons l'hypothèse que toutes les fonction $T_{comm}(i,n)$ et $T_{calc}(i,n)$ sont linéaires en n, c'est-à-dire qu'il existe des constantes $\lambda_i \geq 0$ et $\mu_i > 0$ telles que $T_{comm}(i,n) = \lambda_i \cdot n$ et $T_{calc}(i,n) = \mu_i \cdot n$. Notons que par nos hypothèses, $\lambda_p = 0$ (pour le processeur maître, le coût d'envoi des données vers lui-même est négligeable). De plus, nous ne cherchons qu'une solution rationnelle et non entière comme nous le devrions.

Nous commençons par donner, dans la section 2.4.1, une expression de la durée d'exécution quand tous les processeurs reçoivent une partie du travail total à traiter, et terminent en même temps. Nous montrons ensuite, dans la section 2.4.2, qu'il existe toujours une solution optimale (rationnelle) dans laquelle tous les processeurs recevant des données terminent en même temps. Nous montrons également la condition pour qu'un processeur reçoive une partie du travail. Ces résultats intermédiaires nous permettent ensuite de montrer un résultat nouveau : dans le cas simple qui nous intéresse, le fait de ranger les processeurs par valeurs décroissantes de leurs bandes passantes depuis le processeur maître permet d'atteindre le temps d'exécution optimal. C'est ce que faisons dans la section 2.4.3. Nous déduisons finalement, en section 2.4.4, une heuristique garantie pour le cas général.

2.4.1 Durée d'exécution

Theorème 2.1 (Durée d'exécution). *Dans une solution rationnelle, si chaque processeur P_i reçoit une part n_i non nulle de l'ensemble des n données, et si tous les processeurs terminent leurs calculs à une même date t, alors la durée de l'exécution est*

$$t = \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_i+\mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j+\mu_j}} \tag{2.7}$$

et le processeur P_i reçoit

$$n_i = \frac{1}{\lambda_i+\mu_i} \cdot \left(\prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j+\mu_j} \right) \cdot t \tag{2.8}$$

données à traiter.

Démonstration. Nous voulons exprimer t, la durée de l'exécution, et n_i, le nombre de données que le processeur P_i doit traiter, en fonction de n. L'équation (2.2) nous dit que le processeur P_i termine son calcul au temps : $T_i = \sum_{j=1}^{i} T_{comm}(j,n_j) + T_{calc}(i,n_i)$. Ainsi, avec nos hypothèses actuelles : $T_i = \sum_{j=1}^{i} \lambda_j \cdot n_j + \mu_i \cdot n_i$. Donc, $n_1 = t/(\lambda_1+\mu_1)$ et, pour $i \in [2\,;p]$,

$$T_i = T_{i-1} - \mu_{i-1} \cdot n_{i-1} + (\lambda_i+\mu_i) \cdot n_i.$$

Comme, par hypothèse, tous les processeurs terminent leur calcul en même temps, alors $T_i = T_{i-1} = t$, $n_i = n_{i-1} \cdot \mu_{i-1}/(\lambda_i + \mu_i)$, et nous trouvons l'équation (2.8).

Pour exprimer la durée de l'exécution t en fonction de n il suffit de sommer l'équation (2.8) pour toutes les valeurs de i dans l'intervalle $[1\ ;\ p]$:

$$n = \sum_{i=1}^{p} n_i = \sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \left(\prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} \right) \cdot t$$

ce qui est équivalent à l'équation (2.7). □

Dans la suite, nous noterons :

$$D(P_1, \ldots, P_p) = \frac{1}{\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}}.$$

Nous avons donc, avec les hypothèses du théorème 2.1, $t = n \cdot D(P_1, \ldots, P_p)$.

2.4.2 Terminaisons simultanées

Nous présentons maintenant une condition nécessaire et suffisante sur les fonctions de coût $T_{comm}(i, n)$ et $T_{calc}(i, n)$ pour qu'il existe une solution rationnelle optimale où tous les processeurs reçoivent une part non nulle des données, et où tous les processeurs terminent en même temps. Lorsque cette condition est vérifiée, le théorème 2.1 peut être utilisé pour trouver une solution rationnelle à notre système.

Theorème 2.2 (Terminaisons simultanées). *Soient p processeurs, $P_1, \ldots, P_i, \ldots, P_p$, dont les fonctions exprimant la durée des communications et des calculs, $T_{comm}(i, n)$ et $T_{calc}(i, n)$, sont linéaires en n. Il existe une solution rationnelle optimale où tous les processeurs reçoivent une part non nulle de l'ensemble des données si et seulement si*

$$\forall i \in [1\ ;\ p-1], \quad \lambda_i \leq D(P_{i+1}, \ldots, P_p)$$

et, dans ce cas, tous les processeurs terminent leurs calculs en même temps.

Démonstration. La démonstration est faite par récurrence sur le nombre de processeurs. S'il n'y a qu'un seul processeur, alors le théorème est évidemment vrai. Nous devons ensuite montrer que si le théorème est vrai pour p processeurs, alors il l'est aussi pour $p + 1$ processeurs.

Supposons que nous avons $p+1$ processeurs $P_1, \ldots, P_{p+1}$. Une solution optimale pour $P_1, \ldots, P_{p+1}$ calculant n données est obtenue en donnant $\alpha \cdot n$ données à P_1

et $(1-\alpha)\cdot n$ données à $P_2, \ldots, P_{p+1}$ avec α dans $[0\ ;\ 1]$. La date de fin pour le processeur P_1 est alors $t_1(\alpha) = (\lambda_1 + \mu_1)\cdot n \cdot \alpha$.

Comme le théorème est supposé vrai pour p processeurs, nous savons qu'il existe une solution rationnelle optimale où tous les processeurs P_2 à P_{p+1} travaillent et terminent leurs travaux simultanément, si et seulement si $\lambda_i \leq D(P_{i+1}, \ldots, P_{p+1})$ pour tout $i \in [2\ ;\ p]$. Dans ce cas, par le théorème 2.1, le temps pris par $P_2, \ldots, P_{p+1}$ pour calculer $(1-\alpha)\cdot n$ données est $(1-\alpha)\cdot n \cdot D(P_2, \ldots, P_{p+1})$. Ainsi, les processeurs $P_2, \ldots, P_{p+1}$ terminent tous à la même date $t_2(\alpha) = \lambda_1 \cdot n \cdot \alpha + k \cdot n \cdot (1-\alpha) = k \cdot n + (\lambda_1 - k)\cdot n \cdot \alpha$ avec $k = D(P_2, \ldots, P_{p+1})$.

Si $\lambda_1 \leq k$, alors $t_1(\alpha)$ est strictement croissante, et $t_2(\alpha)$ est décroissante. De plus, nous avons $t_1(0) < t_2(0)$ et $t_1(1) > t_2(1)$, donc la date de l'ensemble $\max(t_1(\alpha), t_2(\alpha))$ est minimisée pour un unique α dans $]0\ ;\ 1[$, quand $t_1(\alpha) = t_2(\alpha)$. Dans ce cas, chaque processeur a des données à calculer, et ils terminent tous à la même date.

À l'inverse, si $\lambda_1 > k$, alors $t_1(\alpha)$ et $t_2(\alpha)$ sont toutes les deux strictement croissantes, donc la date de fin de l'ensemble $\max(t_1(\alpha), t_2(\alpha))$ est minimisée pour $\alpha = 0$. Dans ce cas, le processeur P_1 ne reçoit rien à calculer et sa date de fin est 0, alors que les processeurs P_2 à P_{p+1} terminent tous à une même date $k \cdot n$.

Donc, il existe une solution rationnelle optimale où chacun des $p+1$ processeurs $P_1, \ldots, P_{p+1}$ reçoit une part non nulle de l'ensemble des données, et où tous les processeurs terminent leurs calculs à la même date, si et seulement si $\forall i \in [1\ ;\ p]$, $\lambda_i \leq D(P_{i+1}, \ldots, P_{p+1})$. □

La démonstration du théorème 2.2 montre que, pour un ordre des processeurs fixé, tout processeur P_i ne satisfaisant pas la condition $\lambda_i \leq D(P_{i+1}, \ldots, P_p)$ n'est pas intéressant pour notre problème : l'utiliser ne ferait qu'augmenter le temps total de traitement. Intuitivement, il s'agit des processeurs pour lesquels le simple temps d'envoi d'une quantité de données est plus long que le temps d'envoi et de traitement de la même quantité de données par l'ensemble des processeurs suivants. Par conséquent, nous ne prenons pas ces processeurs en compte, et le théorème 2.2 nous dit qu'il y a une solution rationnelle optimale où tous les processeurs travaillent et ont la même date de fin. Comme le critère de participation, pour un processeur donné, ne dépend que des processeurs suivants, il suffit de vérifier ce critère dans l'ordre inverse des processeurs $(P_{p-1}, P_{p-2}, \ldots, P_1)$, en éliminant au fur et à mesure les processeurs inintéressants.

2.4.3 Politique pour l'ordre des processeurs

Comme nous l'avons dit en section 2.2.3, le processeur maître envoie les données à un processeur après l'autre, et un processeur ne commence effectivement sa communication qu'après que tous les processeurs le précédant ont reçu leur part des

données. L'équation (2.2) montre que, dans notre cas, le temps total d'exécution n'est pas symétrique par rapport aux processeurs, mais dépend de leur ordre. Nous devons donc correctement définir cet ordre afin d'accélérer l'exécution. Il apparaît que le meilleur ordre peut facilement être produit.

Theorème 2.3 (Politique pour l'ordre des processeurs). *Quand les fonctions $T_{comm}(i,n)$ et $T_{calc}(i,n)$ sont toutes linéaires en n, et si nous ne cherchons qu'une solution rationnelle, alors le temps d'exécution le plus court est obtenu quand les processeurs (excepté le processeur maître) sont ordonnés par ordre décroissant de leurs bandes passantes (de P_1, le processeur connecté au processeur maître avec la plus grande bande passante, à P_{p-1}, le processeur connecté au processeur maître avec la plus petite bande passante), le dernier processeur étant le processeur maître. Les vitesses de calcul des processeurs n'ont pas d'influence.*

Démonstration. Considérons n'importe quel ordre $P_1, \ldots, P_p$ des processeurs, excepté que P_p est le processeur maître (comme nous l'avons expliqué en section 2.3.1). Considérons n'importe quelle transposition π qui permute deux voisins, mais laisse inchangé le processeur maître. En d'autre termes, considérons n'importe quel ordre $P_{\pi(1)}, \ldots, P_{\pi(p)}$ des processeurs tel qu'il existe $k \in [1\ ;\ p-2]$, $\pi(k) = k+1$, $\pi(k+1) = k$, et $\forall j \in [1\ ;\ p] \setminus \{k, k+1\}$, $\pi(j) = j$ (notons que $\pi(p) = p$).

Sans aucune perte de généralité, nous pouvons supposer que, dans les deux ordres ($P_1, \ldots, P_p$ et $P_{\pi(1)}, \ldots, P_{\pi(p)}$), tous les processeurs dont le rang est compris entre $k+2$ et p reçoivent une part non nulle de l'ensemble des données. Cela vient du fait que pour tout $i \in [k+2\ ;\ p]$, $P_i = P_{\pi(i)}$. Par conséquent, en vertu du théorème 2.2, la condition pour que P_i reçoive une part non nulle de l'ensemble des données dans l'ordre original est la même que la condition pour que $P_{\pi(i)}$ reçoive une part non nulle de l'ensemble des données dans le nouvel ordre. Ainsi nous n'avons pas besoin de considérer les processeurs de rang supérieur ou égal à $k+2$ qui ne reçoivent aucune donnée dans l'ordre original.

Nous notons par $\Delta(i,p)$ le temps nécessaire à l'ensemble des processeurs P_i, $P_{i+1}, \ldots, P_p$ pour traiter de manière optimale une donnée de taille unitaire. Symétriquement, nous notons par $\Delta_\pi(i,p)$ le temps nécessaire à l'ensemble des processeurs $P_{\pi(i)}, P_{\pi(i+1)}, \ldots, P_{\pi(p)}$ pour traiter de manière optimale une donnée de taille unitaire. Par les remarques précédentes, nous savons que pour toute valeur de $i \in [k+2\ ;\ p]$, $\Delta(i,p) = \Delta_\pi(i,p) = D(P_i, \ldots, P_p)$.

Nous voulons finalement montrer qu'une solution optimale, quand les processeurs sont ordonnés par bandes passantes décroissantes donne le plus petit temps d'exécution possible. Prouver que $\lambda_{k+1} < \lambda_k$ implique $\Delta(1,p) \geq \Delta_\pi(1,p)$ aboutira à ce résultat. Nous supposons donc que $\lambda_{k+1} < \lambda_k$. Nous commençons par montrer que $\Delta(k,p) \geq \Delta_\pi(k,p)$. Nous avons deux cas à étudier :

1. *Dans n'importe quelle solution optimale pour l'ordre $P_k, P_{k+1}, P_{k+2}, \ldots, P_p$,*

au plus un des deux processeurs P_k et P_{k+1} reçoit une part de données non nulle.

N'importe laquelle de ces solutions optimales est également faisable avec l'ordre $P_{k+1}, P_k, P_{k+2}, \ldots, P_p$. Donc, $\Delta_\pi(k,p) \leq \Delta(k,p)$.

2. *Il y a au moins une solution optimale pour l'ordre $P_k, P_{k+1}, P_{k+2}, \ldots, P_p$ dans laquelle les deux processeurs P_k et P_{k+1} reçoivent une part de données non nulle.*

   Dans ce cas, $\Delta(k,p) = D(P_k, \ldots, P_p)$. C'est un peu plus compliqué pour $\Delta_\pi(k,p)$. Si, pour tout i dans $[k\ ;\ p-1]$, $\lambda_{\pi(i)} \leq D(P_{\pi(i+1)}, \ldots, P_{\pi(p)})$, les théorème 2.2 et 2.1 s'appliquent et donc :

$$\Delta_\pi(k,p) = \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)}+\mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)}+\mu_{\pi(j)}}}. \tag{2.9}$$

   À l'opposé, s'il existe au moins une valeur i dans $[k\ ;\ p-1]$ telle que $\lambda_{\pi(i)} > D(P_{\pi(i+1)}, \ldots, P_{\pi(p)})$, alors le théorème 2.2 dit que le temps d'exécution optimal ne peut pas être obtenu par une solution où tous les processeurs reçoivent une part non nulle de l'ensemble des données. Par conséquent, toute solution où chacun des processeurs reçoit une part non nulle de l'ensemble des données et où tous les processeurs terminent en même temps donne un temps d'exécution strictement supérieur à $\Delta_\pi(k,p)$ et :

$$\Delta_\pi(k,p) < \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)}+\mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)}+\mu_{\pi(j)}}}. \tag{2.10}$$

   Les équations (2.9) et (2.10) sont résumées par :

$$\Delta_\pi(k,p) \leq \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)}+\mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)}+\mu_{\pi(j)}}} \tag{2.11}$$

   et la preuve de l'implication suivante :

$$\lambda_{k+1} < \lambda_k \Rightarrow \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)}+\mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)}+\mu_{\pi(j)}}} < \Delta(k,p) \tag{2.12}$$

   prouvera qu'en fait $\Delta_\pi(k,p) < \Delta(k,p)$. Ainsi, nous étudions le signe de

$$\sigma = \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)}+\mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)}+\mu_{\pi(j)}}} - \frac{1}{\sum_{i=k}^{p} \frac{1}{\lambda_i+\mu_i} \cdot \prod_{j=k}^{i-1} \frac{\mu_j}{\lambda_j+\mu_j}}$$

   et nous cherchons à montrer qu'il est négatif. Comme, dans l'expression ci-dessus, les deux dénominateurs sont évidemment (strictement) positifs, le

signe de σ est le signe de :

$$\sum_{i=k}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=k}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} - \sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}. \tag{2.13}$$

Nous souhaitons simplifier la seconde somme dans l'équation (2.13). Pour cela nous remarquons que pour toute valeur de $i \in [k+2\,;\,p]$ nous avons :

$$\prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}} = \prod_{j=k}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}. \tag{2.14}$$

Afin de bénéficier de la simplification proposée par l'équation (2.14), nous décomposons la seconde somme de l'équation (2.13) en trois termes : les termes pour k et $k+1$, et ensuite la somme de $k+2$ à p :

$$\begin{aligned}\sum_{i=k}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=k}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}} &= \frac{1}{\lambda_{k+1} + \mu_{k+1}} \\ &+ \frac{1}{\lambda_k + \mu_k} \cdot \frac{\mu_{k+1}}{\lambda_{k+1} + \mu_{k+1}} + \sum_{i=k+2}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=k}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}. \end{aligned} \tag{2.15}$$

Nous reportons enfin le résultat de l'équation (2.15) dans l'équation (2.13) et nous supprimons les termes apparaissant des deux côtés du signe moins. De cette manière, nous obtenons que σ est du même signe que :

$$\frac{1}{\lambda_k + \mu_k} + \frac{1}{\lambda_{k+1} + \mu_{k+1}} \cdot \frac{\mu_k}{\lambda_k + \mu_k} - \frac{1}{\lambda_{k+1} + \mu_{k+1}} - \frac{1}{\lambda_k + \mu_k} \cdot \frac{\mu_{k+1}}{\lambda_{k+1} + \mu_{k+1}}$$

qui est équivalent à :

$$\frac{\lambda_{k+1} - \lambda_k}{(\lambda_k + \mu_k) \cdot (\lambda_{k+1} + \mu_{k+1})}.$$

Donc, si $\lambda_{k+1} < \lambda_k$, alors $\sigma < 0$, l'équation (2.12) s'applique et dans ce cas $\Delta_\pi(k,p) < \Delta(k,p)$. Remarquons que si $\lambda_{k+1} = \lambda_k$, $\sigma = 0$ et alors $\Delta_\pi(k,p) \leq \Delta(k,p)$. Comme nous n'avons fait aucune hypothèse sur les puissances de calcul de P_k et P_{k+1}, cela montre de manière symétrique que $\Delta_\pi(k,p) \geq \Delta(k,p)$ et ainsi que $\Delta_\pi(k,p) = \Delta(k,p)$. Donc, des processeurs qui sont connectés avec une même bande passante peuvent être placés dans n'importe quel ordre.

Nous venons de montrer que $\Delta_\pi(k,p) \leq \Delta(k,p)$. Nous allons maintenant montrer par récurrence que pour toute valeur de $i \in [1\ ;\ k]$, $\Delta_\pi(i,p) \leq \Delta(i,p)$. Une fois ce résultat établi, nous saurons que $\Delta_\pi(1,p) \leq \Delta(1,p)$, ce qui est exactement notre but.

Supposons que l'hypothèse de récurrence est prouvée de i à k, et concentrons-nous sur le cas $i-1$. Nous avons trois cas à considérer :

1. $\lambda_{i-1} \leq \Delta_\pi(i,p) \leq \Delta(i,p)$.
   Suivant la démonstration du théorème 2.2, nous avons :
   $$\Delta_\pi(i-1,p) = \frac{1}{\frac{1}{\lambda_{i-1}+\mu_{i-1}} + \frac{\mu_{i-1}}{(\lambda_{i-1}+\mu_{i-1})\cdot\Delta_\pi(i,p)}}$$
   et une formule équivalente pour $\Delta(i-1,p)$. Ainsi, $\Delta_\pi(i,p) \leq \Delta(i,p)$ implique $\Delta_\pi(i-1,p) \leq \Delta(i-1,p)$.
2. $\Delta_\pi(i,p) \leq \lambda_{i-1} \leq \Delta(i,p)$.
   Alors, $\Delta_\pi(i-1,p) = \Delta_\pi(i,p)$, $\Delta(i-1,p) = \frac{(\lambda_{i-1}+\mu_{i-1})\cdot\Delta(i,p)}{\Delta(i,p)+\mu_{i-1}}$ et
   $$\begin{aligned}\Delta(i-1,p) - \Delta_\pi(i-1,p) \\ = \frac{\mu_{i-1}\cdot(\Delta(i,p)-\Delta_\pi(i,p)) + (\lambda_{i-1}-\Delta_\pi(i,p))\cdot\Delta(i,p)}{\Delta(i,p)+\mu_{i-1}}.\end{aligned}$$
   Comme $\Delta_\pi(i,p) \leq \Delta(i,p)$ (hypothèse de récurrence) et $\Delta_\pi(i,p) \leq \lambda_{i-1}$ (hypothèse sur λ_{i-1}), alors $\Delta(i-1,p) - \Delta_\pi(i-1,p)$ est positif.
3. $\Delta_\pi(i,p) \leq \Delta(i,p) < \lambda_{i-1}$.
   Dans ce dernier cas le résultat est évident car $\Delta_\pi(i-1,p) = \Delta_\pi(i,p)$ et $\Delta(i,p) = \Delta(i-1,p)$.

Donc, l'inversion des processeurs P_k et P_{k+1} est bénéfique si la bande passante du processeur maître au processeur P_{k+1} est plus grande que celle du processeur maître au processeur P_k. □

2.4.4 Conséquences pour le cas général

Nous avons défini une politique pour l'ordre des processeurs, quand les fonctions de coût sont linéaires en n. Maintenant, comment allons-nous ordonner nos processeurs dans le cas général où les fonctions de coût sont quelconques ? Connaissant les caractéristiques des communications et des calculs de chacun des processeurs, une étude exacte serait possible même dans le cas général. Il peut en effet être envisagé de considérer tous les ordres possibles pour les p processeurs, d'utiliser l'algorithme 2.1 pour calculer le temps d'exécution théorique, et de choisir le

meilleur résultat. Cette approche est théoriquement possible, mais serait en pratique irréaliste pour des grandes valeurs de p. De plus, dans le cas général une étude analytique est bien sûr impossible (nous ne pouvons pas traiter *n'importe quelle* fonction $T_{comm}(i,n)$ ou $T_{calc}(i,n)$).

Nous nous appuyons donc sur le résultat précédent et nous ordonnons les processeurs par ordre décroissant des bandes passantes avec lesquelles ils sont connectés au processeur maître, sauf pour le processeur maître lui-même qui est placé en dernier. Même sans l'étude précédente, une telle politique ne devrait pas être surprenante. Effectivement, le temps passé à envoyer ses données au processeur P_i est payé en temps d'attente par tous les processeurs de P_i à P_p. Donc le premier processeur doit être celui pour qui l'envoi des données est le moins coûteux, et ainsi de suite. Bien sûr, en pratique, les choses se compliquent un peu car nous travaillons en nombres entiers. Néanmoins, l'idée principale est grossièrement la même, comme nous allons le montrer.

Nous supposons uniquement que toutes les fonctions de calcul et de communication sont linéaires. Nous notons alors par :

- T_{opt}^{rat} : le meilleur temps d'exécution qui peut être obtenu pour une distribution *rationnelle* des n données, quel que soit l'ordre des processeurs.
- T_{opt}^{int} : le meilleur temps d'exécution qui peut être obtenu pour une distribution *entière* des n données, quel que soit l'ordre des processeurs.

Remarquons que T_{opt}^{rat} et T_{opt}^{int} peuvent être obtenus pour des ordres différents sur les processeurs. Nous prenons une distribution rationnelle aboutissant au temps d'exécution T_{opt}^{rat}, puis nous l'arrondissons pour obtenir une distribution entière en suivant le mode d'arrondi décrit dans la section 2.3.3. De cette manière, nous obtenons une distribution entière avec un temps d'exécution T' satisfaisant l'équation :

$$T' \leq T_{opt}^{rat} + \sum_{j=1}^{p} T_{comm}(j,1) + \max_{1 \leq i \leq p} T_{calc}(i,1)$$

(la démonstration étant la même que pour l'équation (2.4)). Cependant, comme c'est une solution entière, son temps d'exécution est évidemment au moins égal à T_{opt}^{int}. De plus, comme une solution entière est aussi rationnelle, T_{opt}^{int} est au moins égal à T_{opt}^{rat}. D'où les bornes.

$$T_{opt}^{int} \leq T' \leq T_{opt}^{int} + \sum_{j=1}^{p} T_{comm}(j,1) + \max_{1 \leq i \leq p} T_{calc}(i,1)$$

où T' est le temps d'exécution de la distribution obtenue en arrondissant, suivant le schéma de la section 2.3.3, la meilleure solution rationnelle (où tous les processeurs sont ordonnés par ordre décroissant des bandes passantes de leurs connexions avec

le processeur maître, sauf pour le maître qui est ordonné en dernier). Par conséquent, quand les fonctions de calcul et de communication sont linéaires, notre politique pour l'ordre des processeurs est même garantie !

2.5 Validation expérimentale

Dans cette section, nous présentons les expériences qui ont été menées afin de valider les résultats précédents. Notre expérimentation consiste au calcul de 817 101 rais correspondant à l'ensemble des événements sismiques de l'année 1999. Nous commençons par exposer l'environnement logiciel et matériel utilisé (section 2.5.1) avant de présenter les résultats obtenus (section 2.5.2).

2.5.1 Environnement

Nos expériences ont été menées sur la grille de calcul du projet TAG qui avait été mise en place peu de temps auparavant. Cette grille fédère les ressources de différentes machines réparties entre Strasbourg (ULP) et Montpellier (CINES). Elle nous permet d'avoir accès à un ensemble de systèmes hétérogènes avec des performances (réseau et processeur) hétérogènes. Toutes les machines font tourner Globus [36] et MPICH-G2 [54] est utilisé comme bibliothèque de communication. Les versions des logiciels qui étaient en place lors de nos expérimentations étaient, selon les machines, les versions 1.1.4 ou 2.0 pour Globus, et la version 1.2.2.3 pour MPICH-G2. La mise en place de cette grille a demandé un travail non négligeable. En plus de la complexité de mise en place d'un système comme Globus (que nous découvrions), nous avons eu à faire face à un certain nombre de problèmes, que ce soit pour passer à travers les pare-feux (*firewall*) ou pour communiquer entre des systèmes hétérogènes. Nous avons eu à diagnostiquer et à corriger quelques *bugs*, y compris dans MPICH-G2, et à apporter des modifications à Globus pour permettre un contrôle plus fin des ports utilisés.

Toutes nos expériences ont été réalisées avec un ensemble de 16 processeurs. Le tableau 2.1 montre les ressources utilisées pour l'expérience. L'ensemble des données d'entrée est localisé sur le PC nommé *dinadan*, à la fin de la liste, qui sert de processeur maître. Excepté pour le maître, les ressources sont ordonnées par ordre décroissant de leurs bandes passantes depuis le maître.

Les ordinateurs sont répartis sur deux sites géographiquement éloignés. Les processeurs 1 à 3 et 14 à 16 (PCs standards avec Intel PIII et AMD Athlon XP), ainsi que 4, 5 (deux processeurs Mips d'une SGI Origin 2000) sont situés à Strasbourg, alors que les processeurs 6 à 13 proviennent de *leda*, une SGI Origin 3800 (processeurs Mips) du CINES à Montpellier. Le réseau public est utilisé entre les différentes machines, et le système n'est pas dédié : les différents processeurs

peuvent être partagées avec d'autres utilisateurs. Un système de réservation (LSF) nous permet cependant d'avoir un usage exclusif des processeurs sur les machines parallèles. Les performances des processeurs et des liens réseau sont des valeurs calculées à partir de séries de tests que nous avons effectués avec notre application. Remarquons que *merlin*, avec les processeurs 14 et 15, bien que géographiquement proche du processeur maître, a la plus petite bande passante car il était connecté à un *hub* à 10 Mbits/s pendant les expérimentations, alors que tous les autres étaient connectés à des *switch fast-ethernet*.

La colonne μ indique le nombre de millisecondes nécessaires pour calculer un rai (le plus petit est le meilleur). La note associée est simplement une indication plus intuitive des vitesses des processeurs (la plus grande est la meilleure) : c'est l'inverse de μ, normalisé par rapport à une note de 1 choisie arbitrairement pour le Pentium III/933. Comme il s'agit de valeurs mesurées, il y avait de légères différences entre les valeurs obtenues pour des processeurs identiques sur un même ordinateur (pour *seven*, *leda* et *merlin*). Dans ce cas, la moyenne des performances mesurées pour chacun des processeurs est rapportée.

Les débits des liens réseau entre le processeur maître et les autres nœuds sont reportés dans la colonne λ, en supposant un temps de communication linéaire. Cela indique le temps en microsecondes nécessaire pour recevoir un élément de données du processeur maître. Il y a 65 octets de données à transférer par rai, ce qui fait un total d'environ 51 Mo pour l'ensemble des 817 101 rais. Le fait de considérer de coûts de communication linéaires est suffisamment précis dans notre cas car la latence réseau est négligeable comparée au temps d'envoi des blocs de données.

2.5.2 Résultats

Les résultats expérimentaux de cette section évaluent deux aspects de notre étude. La première expérience compare une exécution non équilibrée (le programme original sans aucune modification du code source) à ce que nous prévoyons comme

Tableau 2.1 – Processeurs utilisés comme nœuds de calcul durant les expérimentations.

Machine	CPU	Type	μ (ms/rai)	Note	λ (μs/rai)
caseb	1	XP1800	4,63	2,00	10,0
pellinore	2	PIII/800	9,37	0,99	11,2
sekhmet	3	XP1800	4,89	1,90	17,0
seven	4, 5	R12K/300	16,16	0,57	21,0
leda	6–13	R14K/500	9,68	0,95	35,3
merlin	14, 15	XP2000	3,98	2,33	81,5
dinadan	16	PIII/933	9,29	1,00	0,0

étant la meilleure exécution équilibrée. La seconde expérience évalue les performances des exécutions par rapport à notre politique pour l'ordre des processeurs (les processeurs sont ordonnés par ordre décroissant de leurs bandes passantes) en comparant cette politique avec son opposée (les processeurs sont ordonnés par ordre croissant de leurs bandes passantes). Dans l'implémentation de MPICH-G2 que nous avons utilisée, l'ordre des processeurs dans l'opération de distribution suit le numéro de rang MPI, qui lui-même est déterminé par la requête Globus servant à lancer l'application. Le programmeur peut donc contrôler l'ordre des processeurs. Notons que, quel que soit l'ordre choisi, le processeur maître est toujours placé à la fin de la liste (en dépit de sa bande passante infinie vers lui-même) car il reçoit ses propres données en dernier (cf. section 2.3.1). Pour chacun des cas, nous présentons le résultat d'une seule exécution, représentative de ce que nous avons pu observer lors de différentes expériences.

Application originale

La figure 2.3 reporte les performances obtenues avec le programme original, dans lequel tous les processeurs reçoivent une même quantité de données. Nous avions à choisir l'ordre des processeurs et, à partir des conclusions données dans la section 2.4.4, nous avons ordonné les processeurs par bandes passantes décroissantes.

Sans surprise, les temps de fin des processeurs diffèrent largement, montrant un gros déséquilibre entre le premier processeur terminant après 241 s et le dernier après 835 s.

Application équilibrée

Dans la seconde expérience, nous évaluons notre stratégie d'équilibrage de charge. L'adaptation du code de l'application a simplement consisté à remplacer l'opération `MPI_Scatter` par un `MPI_Scatterv` permettant d'avoir une distribution arbitraire des données. Pour calculer la distribution, nous avons fait la supposition que les fonctions de coût pour les calculs et les communications étaient affines et croissantes. Cette supposition nous a permis d'utiliser notre heuristique garantie. Avec un si grand nombre de rais, l'algorithme 2.1 nécessite plus de deux jours de calcul (nous l'avons en fait interrompu avant la fin) et l'algorithme 2.2 prend environ 6 minutes pour s'exécuter sur un Pentium III/933 alors que l'exécution de l'heuristique, en utilisant `pipMP` [35, 76], est instantanée et a une erreur relative par rapport à la solution optimale qui est inférieure à 6×10^{-6} !

Les résultats de cette expérience sont présentés sur la figure 2.4. L'exécution apparaît bien équilibrée : les premiers et derniers temps de fin sont respectivement 388 s et 412 s, ce qui représente une différence maximale entre les temps de fin

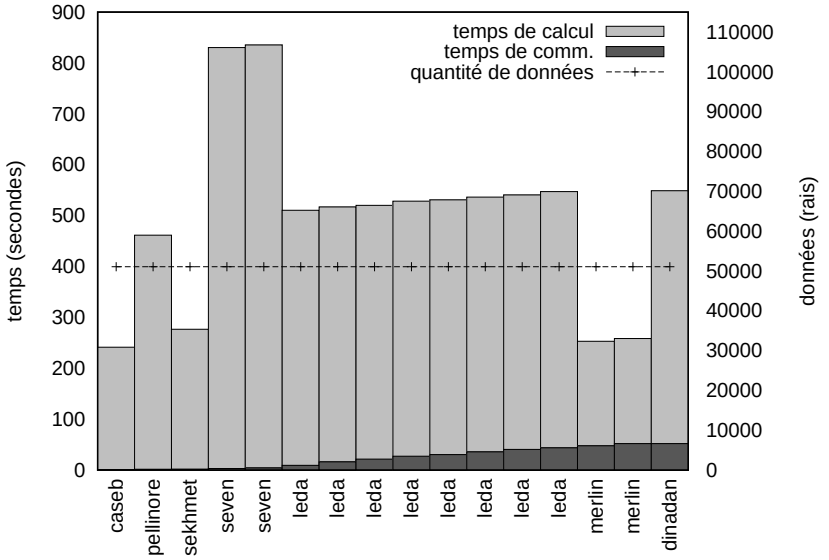


Figure 2.3 – Exécution du programme original (distribution uniforme des données).

de 6% de la durée totale. Par comparaison avec les performances de l'application originale, le gain est significatif : la durée totale d'exécution est approximativement la moitié de la durée de la première expérience.

Politique pour l'ordre des processeurs

Nous comparons maintenant les effets de la politique pour l'ordre des processeurs. Les résultats présentés sur la figure 2.4 ont été obtenus avec l'ordre par bandes passantes décroissantes. La même exécution avec les processeurs rangés par ordre de bandes passantes croissantes est présentée sur la figure 2.5.

L'équilibrage de charge de cette exécution est acceptable avec une différence maximale entre les temps de fin de 12% de la durée totale (les processeurs terminent au plus tôt et au plus tard après 420 s et 469 s). Comme prédit, la durée totale est plus longue (de 57 s) qu'avec les processeurs dans l'ordre inverse. La charge était légèrement moins bien équilibrée que dans la première expérience (à cause d'un pic de charge sur *sekhmet* durant l'expérience [1]). Si on considère plutôt le processeur finissant avant-dernier, l'équilibrage de charge est meilleur : ce processeur termine après 447 s soit une différence de 8% par rapport au processeur le plus rapide.

1. Nous aurions pu recommencer l'expérience pour avoir des chiffres plus « propres » mais, le temps qu'on se rende compte de cette anomalie, les performances du réseau avaient changé (la grille était de plus restée indisponible pendant plusieurs semaines). Il nous aurait alors fallu re-mesurer les bandes passantes et recommencer toutes les expériences pour pouvoir comparer les différents temps d'exécution.

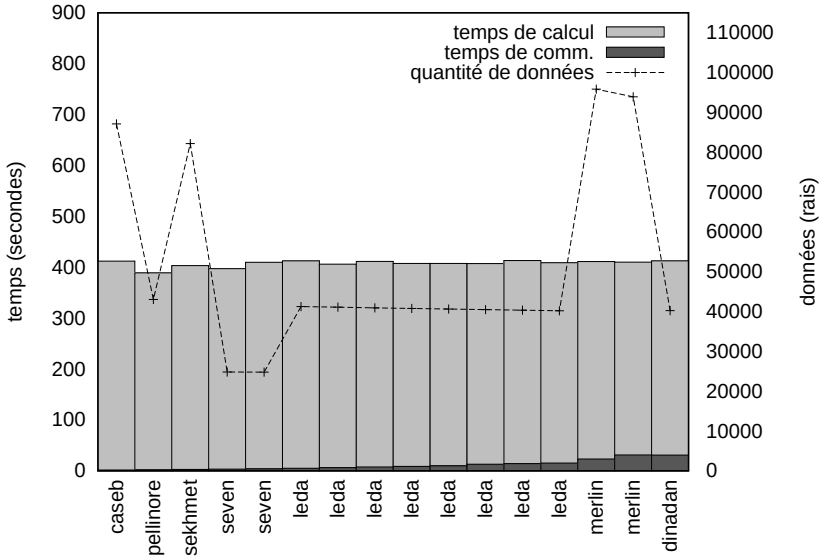


Figure 2.4 – Exécution équilibrée avec les nœuds ordonnés par bandes passantes décroissantes.

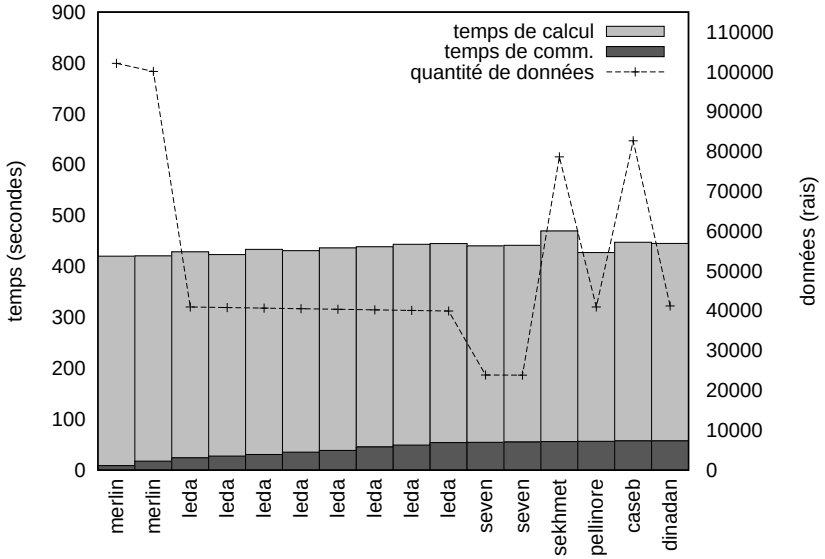


Figure 2.5 – Exécution équilibrée avec les nœuds ordonnés par bandes passantes croissantes.

Cela reste toujours plus long (de 35 s) qu'avec les processeurs dans l'ordre inverse. L'essentiel de la différence vient du temps passé par les processeurs à attendre le début effectif des communications. Cela apparaît clairement sur la figure 2.5 : l'aire de la zone en gris foncé (l'effet d'escalier) est nettement plus grande que sur la figure 2.4.

Qualité des prédictions

Si nous comparons maintenant les temps prédits en utilisant notre modèle (cf. section 2.3) avec les temps de fin obtenus en réalité, nous pouvons voir que la qualité des prédictions dépend des caractéristiques des processeurs, et plus précisément des interférences pouvant avoir lieu durant l'exécution.

Concernant les temps de communication, les prédictions ont été plus précises pour les processeurs géographiquement proches du processeur maître, avec une erreur de prédiction inférieure à 0,7 s, alors qu'elle peut atteindre 5,5 s pour les processeurs situés à l'autre bout du pays. Cela provient certainement du fait que les performances du réseau sont plus stables (et donc plus facilement prédictibles) pour un réseau local que pour un réseau d'envergure plus large. En les comparant aux durées totales des communications, ces erreurs paraissent grandes : respectivement 8,1% et 79,6%. Nous pouvons cependant voir, à partir des figures 2.4 et 2.5 que, en dépit des erreurs de prédiction possibles, il est important de prendre les communications en compte pour choisir le bon ordre des processeurs et pour calculer une distribution des données équilibrée. Concernant les temps de calcul, l'erreur relative est inférieure 2,3% sur les Origin (*leda* et *seven*) alors qu'elle pouvait parfois atteindre 9,2% sur les PC. La différence ici est due au fait que les processeurs des PC pouvaient être partagés avec d'autres utilisateurs alors qu'un système de réservation nous permettait un usage exclusif des processeurs des machines parallèles. Pour le temps total d'exécution des expériences, les prédictions ont été plutôt bonnes pour les deux premières expériences : les temps de fins ont été sous-estimés de moins de 1,9%. Comme cela a déjà été vu pour l'équilibrage de charge, des interférences ont eu lieu pendant la troisième expérience, induisant ici une sous-estimation de 11,9%.

Avec une approche statique comme celle qui a été présentée dans ce chapitre, la qualité des prédictions dépend clairement de la stabilité des bandes passantes des liens réseau et de la puissance processeur disponible. Ces expériences montrent cependant qu'un bon équilibrage de charge, ainsi qu'une bonne prédiction des temps de fin, peuvent être obtenus en pratique.

2.6 Travaux connexes

De nombreux travaux de recherche s'occupent du problème d'équilibrage de charge en environnement hétérogène, mais la plupart d'entre eux considèrent un équilibrage dynamique. Par exemple, le travail de George [41] est fortement lié à notre problème. Dans ce travail, une bibliothèque permet au programmeur de produire, durant l'exécution, des statistiques de charge par processeur et cette information peut alors être utilisée pour redistribuer des données d'une itération à l'autre. Une approche dynamique est également utilisée par Heymann, Senar, Luque et Livny [49] pour optimiser, au fur et à mesure de l'exécution, le nombre de processeurs esclaves à utiliser. Leur solution est implémentée à l'aide de MW, de Goux, Kulkarni, Yoder et Linderoth [42], qui est une bibliothèque facilitant l'écriture d'applications de type maître-esclave. Cependant, l'équilibrage de charge dynamique ajoute à l'exécution une surcharge qui peut être évitée avec une approche statique.

L'approche statique est utilisée dans des contextes variés. Cela va du partitionnement de données pour un traitement parallèle de vidéo par Altilar et Paker [5] à la recherche du nombre optimal de processeurs pour des algorithmes d'algèbre linéaire par Barbosa, Tavares et Padilha [9]. Une approche par flot peut également être utilisée comme dans le travail de Shao [87, 86]. Plus généralement, de nombreux résultats ont été produits en utilisant la théorie des tâches divisibles (*divisible load theory*) pour des topologies de réseaux variées (cf. Bharadwaj, Ghose, Mani et Robertazzi [17, 18] pour un aperçu). Notre modèle correspond au *single-level tree network without front-end processors* de [17] : le processeur maître doit attendre que tous les esclaves ont reçu leur part de données respective avant de pouvoir commencer à traiter sa propre part du travail. Pour ces réseaux, quand l'ordre des processeurs est fixé, Robertazzi [81, 17] a établi une condition pour qu'un processeur participe à une solution optimale, et il a montré qu'il y avait une solution optimale où tous les processeurs participant terminent leurs calculs à la même date. Ces résultats sont donnés par nos théorèmes 2.1 et 2.2. Des résultats similaires existent pour des *single-level tree networks with front-end processors* où le processeur maître peut calculer pendant qu'il envoie des données aux esclaves. Dans ce cadre, Kim, Jee, et Lee ont prétendu avoir montré que, dans une solution optimale, les processeurs sont aussi ordonnés par bandes passantes décroissantes [56]. Leur démonstration repose sur l'hypothèse non prouvée que dans une solution optimale, tous les processeurs participent. Dans le même contexte, on retrouve le même défaut dans la preuve de Błażewicz et Drozdowski [19]. Beaumont, Legrand, et Robert [13, 14] ont démontré, pour les *single-level tree networks with front-end processors*, que dans toute solution optimale, les processeurs sont ordonnés par bandes passantes décroissantes, ils participent tous au calcul et terminent leurs exécutions en même temps. Ils ont également proposé des algorithmes en

plusieurs tournées asymptotiquement optimaux.

Dans [3], Almeida, González, et Moreno présentent un algorithme de programmation dynamique pour résoudre le problème du maître-esclave dans des systèmes hétérogènes. La principale différence avec notre travail est qu'ils considèrent que les esclaves doivent renvoyer des résultats au maître. Dans leur modèle, le maître ne calcule rien, et il ne peut pas envoyer et recevoir de données simultanément. Le principal inconvénient de leur travail est qu'ils supposent simplement que les processeurs doivent être ordonnés par ordre décroissant de leur puissance de calcul (les processeurs les plus rapides en premier). Ils ne justifient pas cette approche alors que différents auteurs et nous-mêmes avions précédemment montré que bien souvent, le critère de tri devait être les valeurs des bandes passantes des liens réseaux [56, 11, 13, 40]. Avec une formulation du problème proche de la nôtre, Beaumont, Legrand, et Robert donnent, dans [12], une solution en temps polynomial en utilisant un algorithme glouton complexe. Il se restreignent cependant à des fonctions de coût linéaires, alors que notre solution présentée en section 2.3.2 est valide pour toute fonction de coût positive. Une autre différence est dans l'ordonnancement produit : nous contraignons que pour chaque esclave, toutes les données doivent être envoyées avant de commencer le calcul, alors qu'ils autorisent les communications avec différents esclaves à être entrelacées. Les mêmes auteurs, en association avec Banino, Carter et Ferrante, ont étudié dans [8] des réseaux avec une structure plus générale en forme d'arbres. Plutôt que de chercher une solution optimale au problème, ils caractérisent le meilleur régime permanent pour différents modèles d'opération.

Un problème apparenté est la distribution de boucles pour des processeurs hétérogènes afin d'équilibrer la charge de travail. Ce problème est étudié par Cierniak, Zaki et Li dans [29], dans le cas particulier d'itérations indépendantes, ce qui est équivalent à une opération de distribution. Cependant, les fonctions de coût pour les communications et les calculs sont affines. Une solution d'équilibrage de charge est d'abord présentée pour des processeurs hétérogènes, seulement s'il n'y a pas de contention réseau. Ensuite, la contention est prise en compte, mais seulement pour des processeurs homogènes.

2.7 Conclusion

Nous avons, dans ce chapitre, étudié une approche d'équilibrage de charge statique pour optimiser l'exécution d'applications de type maître-esclave en environnement hétérogène. Cette approche repose sur le calcul d'une répartition des données adaptée aux paramètres de la plate-forme.

Nous avons présenté deux solutions pour calculer une distribution équilibrée. Nous avons d'abord donné un algorithme général qui trouve une solution exacte.

Nous avons ensuite présenté une optimisation de cet algorithme, valide si les fonctions de coût sont croissantes en le nombre de données. Dans le cas où les fonctions de coût sont affines, nous avons proposé une heuristique qui est beaucoup plus rapide tout en étant garantie. Nous avons également proposé une politique pour ordonner les processeurs : ils doivent être ordonnés par bandes passantes (depuis le processeur maître) décroissantes. Des résultats expérimentaux avec notre application cible illustrent les gains obtenus.

Dans le modèle maître-esclave que nous venons d'étudier, un ensemble de tâches est à effectuer, chacune des tâches utilisant une donnée distincte. Les tailles des données et les durées des tâches sont homogènes. Nous allons, dans le chapitre suivant, étendre ce modèle au cas où les tâches et les données peuvent avoir des tailles différentes. Nous allons également généraliser les relations entre les tâches et les données : une donnée pourra être utilisée par plusieurs tâches et chaque tâche pourra utiliser plusieurs données.

Chapitre 3

Avec des données partagées

3.1 Introduction

L'énoncé du problème traité dans le chapitre précédent comporte un certain nombre d'hypothèses qui limitent le champ d'application des solutions proposées. Il s'agissait de traiter un ensemble de tâches indépendantes, chaque tâche utilisant une donnée distincte. Les tailles des tâches étaient homogènes, et les tailles des données aussi. Pour ce qui est de la plate-forme, nous nous restreignions à des plates-formes de type maître-esclave, avec un modèle de communication de type un-port. Ces plates-formes pouvaient toutefois être hétérogènes. Nous n'autorisions pas de recouvrement entre les calculs et les communications et des solutions en une seule tournée ont été proposées : chaque esclave attend d'avoir reçu toutes ses données avant de commencer à effectuer les tâches correspondantes.

Nous allons maintenant chercher à lever un certain nombre de ces limitations. La plate-forme cible reste la même : une plate-forme de type maître-esclave hétérogène avec un modèle de communication un-port. Nous permettons toutefois aux processeurs d'effectuer des calculs et des communications en parallèle (recouvrement). Les principaux changements interviennent dans le modèle d'application. Pour commencer, nous allons autoriser des tâches de tailles différentes ainsi que des données de tailles différentes. Les tâches sont toujours indépendantes, mais nous généralisons les relations de dépendances possibles entre les tâches et les données : une donnée peut être *partagée* par plusieurs tâches, c'est-à-dire que plusieurs tâches peuvent utiliser une même donnée en entrée. Une tâche peut également utiliser plusieurs données. C'est principalement ce partage potentiel des données qui introduit une difficulté supplémentaire.

Ces extensions au modèle initial sont motivées par le travail de Casanova, Legrand, Zagorodnov et Berman [25, 24] qui vise à ordonnancer des tâches dans le cadre d'*AppLeS Parameter Sweep Template* (APST) [15]. APST est un envi-

ronnement 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. MCell [89] est un exemple d'application entrant dans ce cadre. Cette application sert à simuler les interactions biochimiques des molécules dans les cellules vivantes. Une exécution de MCell est composée de multiples simulations par Monte-Carlo, chacune des simulations utilisant les mêmes données en entrée. Ces données décrivent les cellules concernées par les simulations ainsi que les positions initiales des différentes molécules. Plusieurs exécutions de MCell peuvent ainsi réutiliser les mêmes données. MCell est représentative d'une vaste classe de simulations par Monte-Carlo multiples [16, 66].

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.* [25, 24] 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 [68, 69]. 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 ce chapitre, traiter le même problème d'ordonnancement où un maître sert d'entrepôt pour toutes les données. Son rôle est de distribuer les données aux esclaves pour qu'ils puissent exécuter les tâches. L'objectif est donc d'allouer les tâches aux esclaves, puis de sélectionner quelles données envoyer à quel esclave, et dans quel ordre, afin de minimiser le temps total d'exécution de l'application. Ce modèle maître-esclave comporte un inconvénient qui est que les communications du maître vers les esclaves peuvent devenir un goulot d'étranglement. Il est cependant important de commencer par se concentrer sur ce modèle simple pour bien comprendre le problème. Une généralisation de la plate-forme d'exécution en autorisant plusieurs entrepôts pour les données, et un réseau d'interconnexion distribué (et non plus centralisé) sera l'objet du chapitre suivant.

Ce chapitre a deux volets. D'un point de vue théorique, nous établissons deux résultats de complexité montrant la difficulté du problème. Le premier de ces résultats montre la NP-complétude du problème d'ordonnancement avec un seul esclave. Nous montrons également qu'avec deux esclaves, dans le cas très particulier où toutes les tâches et toutes les données ont une taille unitaire, le problème

est aussi NP-complet. Cela justifie donc l'approche par heuristiques pour trouver un ordonnancement. D'un point de vue pratique, nous proposons plusieurs nouvelles heuristiques qui se révèlent aussi performantes que les meilleures heuristiques dans [25, 24] mais pour un coût beaucoup plus faible.

Dans la section 3.2, nous exposons précisément le problème avant de présenter, en section 3.3, des résultats de complexité. Les nouvelles heuristiques sont ensuite présentées en section 3.4 avant de les comparer par de nombreuses simulations dans la section 3.5 puis de conclure en section 3.6.

3.2 Modèles

Nous commençons, dans cette section, par poser formellement le problème d'optimisation à résoudre.

3.2.1 Tâches et données

Notre problème est d'ordonnancer un ensemble constitué de n tâches indépendantes $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Ces tâches ont des tailles différentes : le poids de la tâche T_j est t_j, avec $1 \leq j \leq n$. Le poids d'une tâche correspond à son temps de traitement sur un processeur de vitesse unitaire. Les tâches sont indépendantes, il n'y a donc pas de contraintes de dépendance entre elles.

L'exécution de chaque tâche dépend toutefois 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 $\mathcal{D} = \{D_1, D_2, \ldots, D_m\}$. Le taille de la donnée D_i est d_i, avec $1 \leq i \leq m$. Nous utilisons un graphe biparti $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ pour représenter les relations entre les tâches et les données : c'est le graphe d'application. L'ensemble des nœuds du graphe $\mathcal{G}$ est $\mathcal{V} = \mathcal{D} \cup \mathcal{T}$, et chaque nœud est pondéré par d_i ou t_j selon qu'il est dans $\mathcal{D}$ ou bien dans $\mathcal{T}$. Il y a une arête $e_{i,j} : D_i \to T_j$ dans $\mathcal{E}$ si et seulement si la tâche T_j dépend de la donnée D_i. Toutes les données D_i telles que $e_{i,j} \in \mathcal{E}$ sont nécessaires pour pouvoir commencer l'exécution de T_j. Elles peuvent par exemple correspondre à des fichiers pouvant contenir le code (binaire) de la tâche T_j ou bien des données d'entrée pour cette tâche. Le processeur qui aura à exécuter la tâche T_j aura besoin de recevoir toutes les données D_i telles que $e_{i,j} \in \mathcal{E}$ avant de pouvoir commencer l'exécution de T_j. La figure 3.1 montre un petit exemple avec $m = 9$ données et $n = 13$ tâches. Dans l'exemple, la tâche T_1 dépend des données D_1 et D_2, la tâche T_3 dépend des données D_2, D_3, D_4 et D_7, et ainsi de suite.

Pour résumer, le graphe biparti $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, où chaque nœud dans $\mathcal{V} = \mathcal{D} \cup \mathcal{T}$ est pondéré par d_j ou t_j, et où les arêtes de $\mathcal{E}$ représentent les relations entre les tâches et les données, contient toutes les informations modélisant l'application.

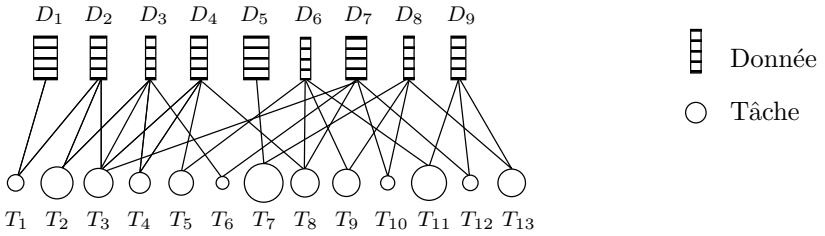


Figure 3.1 – Graphe biparti rassemblant les relations entres les tâches et les données.

3.2.2 Graphe de plate-forme

Les tâches sont à ordonnancer et à exécuter sur une plate-forme de type maître-esclave hétérogène. Nous notons $\mathcal{P}$ le graphe de plate-forme, qui est un graphe en étoile (cf. figure 3.2) avec un processeur maître P_0 et p processeurs esclaves : $P_1, P_2, \ldots, P_p$. Nous représentons la puissance du processeur P_q par son temps de cycle (relatif) w_q : il faut $t_j \cdot w_q$ unités de temps pour exécuter la tâche T_j sur le processeur P_q. Remarquons que tous les résultats de ce chapitre peuvent sans problème être étendus au cas plus général de temps d'exécution *inconsistants*, avec la terminologie de [21]. Dans cette situation, chaque tâche T_j a un temps d'exécution $w_{j,q}$ qui est différent sur chaque processeur P_q, et il n'y a aucune relation entre ces différents temps d'exécution. Il suffit alors de remplacer toutes les expressions $t_j \cdot w_q$ par $w_{j,q}$.

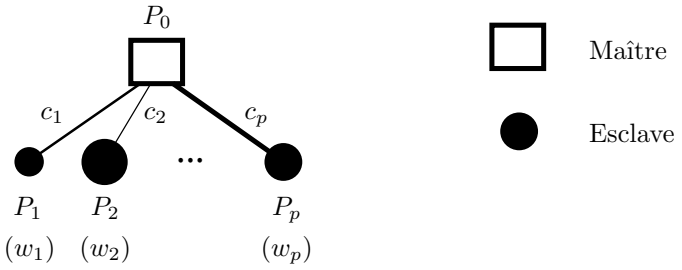


Figure 3.2 – Graphe en étoile hétérogène.

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 (comme dans le chapitre 2) le modèle un-port : le maître peut seulement communiquer avec un seul esclave à la fois. Ce modèle sera rediscuté dans le chapitre 4, lorsque nous généraliserons la plate-forme. Nous notons c_q l'inverse de la bande passante du lien entre P_0 et P_q. Il y a

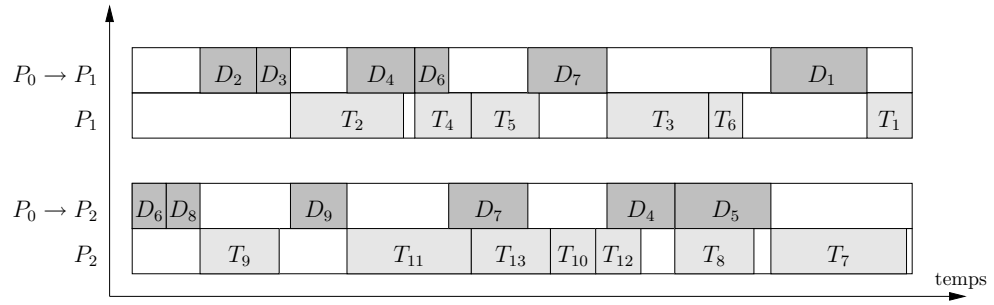


Figure 3.3 – Exemple d'ordonnancement pour le graphe d'application de la figure 3.1

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. Contrairement au chapitre 2, 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.

Par exemple, pour une plate-forme à deux esclaves, la figure 3.3 montre un ordonnancement possible pour le graphe d'application de la figure 3.1, dans le cas où la plate-forme est homogène. Avec cet ordonnancement, les tâches T_1 à T_6 sont exécutées par l'esclave P_1 et les tâches T_7 à T_{13} sont exécutées par l'esclave P_2. En tout, P_1 reçoit six données (D_1 à D_4, D_6 et D_7) et P_2 reçoit six données (D_4 à D_9). On peut remarquer que trois données (D_4, D_6 et D_7) doivent être envoyées aux deux esclaves.

Pour résumer, nous considérons un modèle maître-esclave totalement hétérogène : les esclaves ont des vitesses différentes et les liens réseau ont des capacités différentes. Les communications depuis le maître se font suivant le modèle un-port.

3.2.3 Fonction objectif

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($\mathcal{G}, \mathcal{P}$) le problème d'optimisation à résoudre [1].

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 ;

1. Nous avons fait le choix de conserver, sans les traduire, les noms originaux des problèmes tels qu'ils apparaissent dans les articles publiés. Nous parlions alors de *fichiers* partagés plutôt que de *données*, d'où le nom de TasksSharingFiles.

– 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.

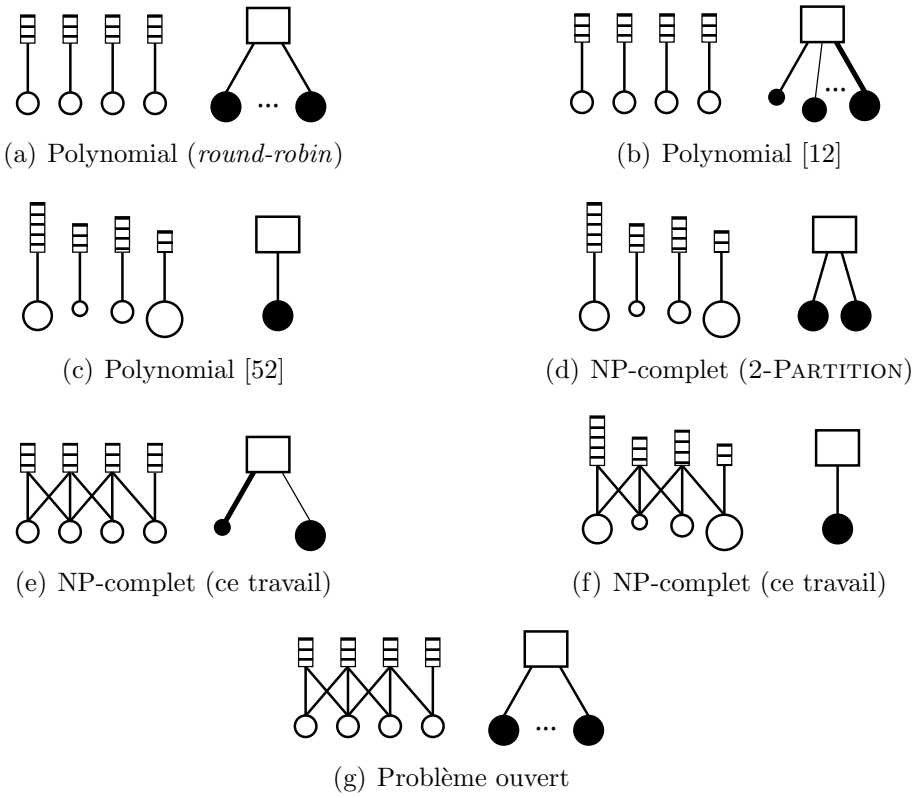
Ainsi, une manière de réduire le temps total d'exécution va être de chercher à réduire la quantité de données répliquées en plaçant, sur un même processeur, l'ensemble des tâches dépendant d'une même donnée, car le temps de communication se trouverait ainsi réduit. On a alors le risque de placer toutes 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.

3.3 Complexité

Il est connu que la plupart des problèmes d'ordonnancement sont difficiles [88, 28]. 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. Dans cette section, nous évoquons différents résultats qui sont rassemblés dans la figure 3.4. Dans la figure 3.4, les pictogrammes se lisent de la manière suivante : pour chacun des sept cas, le diagramme de gauche représente le graphe d'application et celui de droite le graphe de plate-forme. Les légendes sont les mêmes que pour les figures 3.1 et 3.2. Des tailles différentes pour les symboles représentent l'hétérogénéité. Le graphe d'application est ainsi composé de tâches et de données qui sont de tailles homogènes dans les situations (a), (b), (g) et (e), alors que ce n'est pas le cas pour les situations (c), (d) et (f). Les tâches dépendent d'une seule donnée (privée) dans les situations (a), (b), (c) et (d), contrairement aux situations (g), (e) et (f) où les dépendances sont quelconques. Pour ce qui est du graphe de plate-forme, il n'y a qu'un seul esclave dans les situations (c) et (f), et plusieurs esclaves sinon. La plate-forme est homogène dans les cas (a), (d) et (g) et hétérogène dans les cas (b) et (e). Nous allons maintenant détailler la complexité de ces sept instances du problème.

3.3.1 Avec un seul esclave

L'instance de TasksSharingFiles avec un seul esclave se révèle plus difficile que ce que l'on pourrait penser intuitivement.



(a) Polynomial (*round-robin*)

(b) Polynomial [12]

(c) Polynomial [52]

(d) NP-complet (2-Partition)

(e) NP-complet (ce travail)

(f) NP-complet (ce travail)

(g) Problème ouvert

Figure 3.4 – Résultats de complexité pour le problème d'ordonnancement de tâches partageant des données.

Aucune donnée partagée

Dans le cas particulier où chaque tâche dépend d'une unique donnée non partagée, c'est-à-dire $n = m$ et $\mathcal{E}$ se réduit aux n arêtes $e_{i,i} : D_i \rightarrow T_i$ (c'est la situation (c) de la figure 3.4), le problème peut être résolu en temps polynomial. Ce problème est en fait équivalent au problème de *flow-shop* à deux machines et l'algorithme de Johnson [52] [22, chapitre 7] peut être utilisé pour calculer le temps d'exécution optimal. L'algorithme de Johnson fonctionne de la manière suivante : les tâches sont séparées en deux groupes, $\mathcal{A}$ et $\mathcal{B}$. Le groupe $\mathcal{A}$ est composé des tâches dont le temps de communication (le temps nécessaire pour envoyer les données) est inférieur ou égal au temps de calcul ; les autres tâches, c'est-à-dire celles dont le temps de communication est supérieur au temps de calcul, sont mises dans le groupe $\mathcal{B}$. Il faut d'abord ordonnancer les tâches de $\mathcal{A}$, par ordre croissant de leur temps de communication. Ensuite, les tâches de $\mathcal{B}$ sont ordonnancées par ordre décroissant de leur temps de calcul.

Des données sont partagées

Le cas général avec un seul esclave et où les données sont partagées entre les tâches correspond à la situation (f) de la figure 3.4. Nous allons maintenant montrer que cette instance du problème est NP-complète. L'intérêt de cette démonstration est de montrer qu'il n'y a pas (sauf si P = NP) d'algorithme polynomial pour étendre l'algorithme de Johnson pour des graphes de dépendance quelconques.

Le problème de décision associé à l'instance générale de TasksSharingFiles avec un seul esclave peut être formellement défini par :

Définition 3.1 (TSF1-Dec$(\mathcal{G}, \mathcal{P}, p = 1, K)$)**.** Étant donnés un graphe d'application biparti $\mathcal{G}$, une plate-forme $\mathcal{P}$ avec un seul esclave ($p = 1$) et une borne de temps K, est-il possible d'ordonnancer toutes les tâches en K unités de temps ?

Theorème 3.1. TSF1-Dec$(\mathcal{G}, \mathcal{P}, p = 1, K)$ *est NP-complet.*

Démonstration. Il est évident que TSF1-Dec$(\mathcal{G}, \mathcal{P}, p = 1, K)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction de MEWC, le problème de clique de poids maximum (*Maximum Edge-Weighted Clique*) qui est NP-complet [67]. Considérons une instance arbitraire $\mathcal{I}_1$ de MEWC : étant donnés un graphe complet dont les arêtes sont pondérées $G_c = (V_c, E_c, w)$, où $w : E_c \to \mathbb{N}$ est la fonction de poids, une borne de taille B avec $3 \leq B \leq |V_c|$, et une borne de poids W, y a-t-il un sous-ensemble S de B sommets tel que $\sum_{e \in E_S} w(e) \geq W$? Ici, E_S dénote l'ensemble des $B \cdot (B-1)/2$ arêtes reliant les sommets de S. En d'autres termes, peut-on trouver B sommets engendrant un sous-graphe de poids au moins égal à W ? Remarquons que la formulation originale de MEWC dans [67] cherche un sous-ensemble de *au moins* B sommets au lieu de *exactement* B sommets comme nous le faisons ici. Il est cependant évident que notre formulation reste NP-complète (tout algorithme polynomial résolvant notre formulation pourrait être invoqué au plus $|V|$ fois pour résoudre la formulation originale).

Nous construisons l'instance suivante $\mathcal{I}_2$ de TSF1-Dec$(\mathcal{G}, \mathcal{P}, p = 1, K)$. Nous posons $\mathcal{D} = V_c \cup \{X\}$ et $\mathcal{T} = E_c \cup \{T_x\}$ (cf. figure 3.5), ce qui définit $\mathcal{V} = \mathcal{D} \cup \mathcal{T}$. Il y a $m = |V_c| + 1$ données et $n = |E_c| + 1 = (m-1) \cdot (m-2)/2 + 1$ tâches (le graphe original G_c étant complet, $|E_c| = |V_c| \cdot (|V_c| - 1)/2$).

La taille de la donnée X est 1 et la taille de chaque donnée correspondant à un nœud dans V_c est $d = W \cdot (2B-1)$. Le poids de la tâche T_x est $x = W \cdot (B^2 + 2B - 2)$. Remarquons que $x \geq 0$ car $B \geq 3$. Le poids de la tâche correspondant à une arête $e \in E_c$ est $2W + w(e)$.

Les relations entre les tâches et les données sont définies comme suit. D'abord, il y a une arête de la donnée X vers chacune des tâches de $\mathcal{T}$. Ensuite, il y a une arête d'un nœud (correspondant à une donnée) $v \in V_c \subset \mathcal{D}$ vers un nœud (correspondant à une tâche) $e \in E_c \subset \mathcal{T}$ si et seulement si v était une des deux

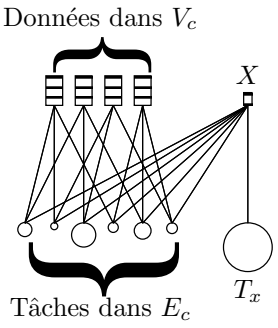


Figure 3.5 – Le graphe biparti utilisé dans la preuve du théorème 3.1 avec $|V_c| = 4$.

extrémités de l'arête e dans G_c. On a donc chaque tâche-arête (dans $\mathcal{T} \setminus \{T_x\}$) qui dépend exactement de trois données (X et les deux extrémités de l'arête). La plate-forme est très simple : un seul esclave avec des temps de calcul et des temps de communication unitaires : $c_1 = w_1 = 1$. Finalement, nous définissons la borne pour l'ordonnancement :

$$K = 1 + x + \sum_{e \in E_c} w(e) + 2W \cdot |E_c|$$

L'instance $\mathcal{I}_2$ peut clairement être construite en temps polynomial en la taille de $\mathcal{I}_1$. Il nous faut maintenant montrer que $\mathcal{I}_2$ admet une solution si et seulement si $\mathcal{I}_1$ en a une.

Supposons d'abord que $\mathcal{I}_1$ a une solution, c'est-à-dire que G_c possède B sommets engendrant un sous-graphe dont la somme des poids des arêtes est de au moins W. Soit $\mathcal{C} = \{v_1, v_2, \ldots, v_B\}$ l'ensemble de ces B sommets. L'idée intuitive pour construire l'ordonnancement est la suivante : après avoir envoyé X, le maître envoie les B données correspondant aux B sommets de $\mathcal{C}$. Comme ces données génèrent une grosse quantité de travail, le processeur esclave aura assez de travail à effectuer pendant qu'il recevra les autres données. L'idée est de garder le processeur esclave actif tout le temps, dès le moment où il a reçu la donnée X. La borne K est définie en conséquence : la première unité de temps est passée à recevoir X et le reste correspond à la somme des poids de toutes les tâches.

L'ordonnancement est défini comme suit :

1. Au temps $t = 0$, la donnée X est envoyée à l'esclave.
2. Le maître envoie les données (correspondant aux nœuds de V_c) aussi tôt que possible, c'est-à-dire au temps $t = 1 + (j-1) \cdot d$ pour la j^{e} donnée avec $1 \leq j \leq |V_c|$, d étant la taille de chaque donnée de V_c. Les B premières données envoyée sont celles de $\mathcal{C}$, dans n'importe quel ordre. Les $|V_c| - B$ données restantes sont ensuite envoyées dans n'importe quel ordre.

3. L'esclave a une file d'exécution qu'il traite de manière gloutonne avec un ordre FIFO. Au temps $t = 1$, T_x est disponible dans la file et l'esclave commence son exécution. À la réception de la première donnée de V_c, il n'y a pas de nouvelle tâche de prête. À la réception de la j^{e} donnée, avec $j \geq 2$, il y a $j - 1$ nouvelles tâches prêtes à être exécutées : elles correspondent à toutes les arêtes de G_c dont une extrémité est la j^{e} donnée, et dont l'autre extrémité est une des $j - 1$ données de V_c reçue précédemment. Ces $j - 1$ tâches sont insérées à la fin de la file d'exécution, dans n'importe quel ordre.

Nous avons dérivé un ordonnancement pour l'instance $\mathcal{I}_2$, mais respecte-t-il la borne d'exécution K ? Comme il a déjà été dit, ce n'est possible que si l'esclave n'est jamais inactif après avoir reçu la donnée X. Soit $\mathsf{RC}(j)$ la *capacité de réception* de l'esclave à la réception de la j^{e} donnée de V_c : $\mathsf{RC}(j)$ dénote la quantité de travail qu'il reste à exécuter pour la tâche en cours et celles qui sont prêtes dans la file d'exécution. Il s'agit donc du temps maximum que le processeur peut passer à attendre une nouvelle donnée sans risquer de devenir inactif. De la même manière, $\mathsf{RC}(0)$ dénote la capacité de réception de l'esclave à la réception de la donnée X, c'est-à-dire le temps nécessaire pour exécuter la tâche T_x. Nous voulons donc avoir $\mathsf{RC}(j) \geq d$ pour tout $j \geq 0$: cela permettrait à l'esclave de recevoir une nouvelle donnée sans devenir inactif.

Initialement, à cause de la tâche T_x, nous avons $\mathsf{RC}(0) = x$. Soit E_j l'ensemble des tâches de E_c qui dépendent uniquement des j premières données de V_c envoyées à l'esclave. Nous avons $\mathsf{RC}(1) = x - d$ (la première donnée n'apporte pas de travail) et

$$\mathsf{RC}(j) = x + \sum_{e \in E_j} w(e) + 2W \cdot |E_j| - d \cdot j$$

pour tout $j \geq 2$. Cette quantité est en fait la somme des temps d'exécution des tâches de $E_j \cup \{T_x\}$, moins le temps passé à envoyer les j premières données. Nous voulons montrer que $\mathsf{RC}(j) \geq d$ pour tout j, $0 \leq j \leq |V_c|$. Nous avons $\mathsf{RC}(0) - d = x - d \geq x - 2d = \mathsf{RC}(1) - d$. De plus, $x - 2d = W \cdot (B^2 - 2B) \geq 0$ car $B \geq 3$. Ainsi, $\mathsf{RC}(0) \geq d$ et $\mathsf{RC}(1) \geq d$. Pour $j \geq 2$, $|E_j| = j \cdot (j-1)/2$ et $\mathsf{RC}(j) - d = \sum_{e \in E_j} w(e) - W - h(j)$ avec $h(j) = W \cdot (j - B)^2$. Le minimum de $h(j)$ est zéro et est atteint pour $j = B$. Mais à cause du choix des B premières données envoyées à l'esclave, $\sum_{e \in E_B} w(e) \geq W$, d'où $\mathsf{RC}(B) - d \geq 0$. Pour $j \neq B$, $h(j) \geq h(B-1) = h(B+1) = W$ et $\mathsf{RC}(j) - d \geq h(j) - W \geq 0$. Cela conclut donc la preuve que le temps total d'exécution de l'ordonnancement est égal à K, et donc que c'est une solution à $\mathcal{I}_2$.

Supposons maintenant que $\mathcal{I}_2$ a une solution. Nous avons un ordonnancement qui s'exécute en $K = 1 + x + \sum_{e \in E_c} w(e) + 2W \cdot |E_c|$ unités de temps. Comme K est égal à un plus la somme des poids des tâches, et que le processeur esclave est

inactif jusqu'à ce que la donnée X a été reçue, la première données envoyée est nécessairement X et cette émission dure une unité de temps. Après le premier pas de temps, le processeur esclave doit être gardé occupé tout le temps. En posant E_j, l'ensemble des tâches de E_c qui dépendent uniquement des j premières données de V_c envoyées à l'esclave, nous devons avoir comme précédemment $\mathsf{RC}(j) \geq d$ pour tout $1 \leq j \leq |V_c|$. Pour $j \geq 2$, nous savons que $\mathsf{RC}(j) \leq x + \sum_{e \in E_j} w(e) + 2W \cdot |E_j| - d \cdot j$. Nous avions une égalité auparavant, mais peut-être que l'ordonnancement n'a pas envoyé les données le plus tôt possible, d'où l'inégalité ici. En prenant $j = B$, nous dérivons comme tout à l'heure que $\mathsf{RC}(B) - d \leq \sum_{e \in E_B} w(e) - W$. Comme l'esclave n'est jamais inactif après avoir reçu la B^{e} donnée, nous avons $\mathsf{RC}(B) - d \geq 0$ et nous en déduisons que $\sum_{e \in E_B} w(e) \geq W$. Les B premières données envoyées à l'esclave définissent donc une solution à $\mathcal{I}_1$. □

3.3.2 Avec plusieurs esclaves

Avec plusieurs esclaves, certaines instances du problème ont une complexité polynomiale.

Aucune donnée partagée

Pour commencer, un algorithme du tourniquet (*round-robin*) est optimal dans la situation (a) de la figure 3.4 : chaque tâche dépend d'une seule donnée non partagée, toutes les tâches ont le même poids, toutes les données ont la même taille et la plate-forme en étoile est homogène.

En gardant les mêmes hypothèses d'homogénéité et de non-partage pour le graphe d'application, mais en ayant des esclaves hétérogènes (situation (b) de la figure 3.4), le problème reste polynomial mais l'algorithme optimal, proposé par Beaumont, Legrand et Robert [12], devient complexe. Le problème où chaque processeur peut traiter n'importe quel nombre de tâches est d'abord transformé en un problème avec plus de processeurs mais où chaque processeur peut traiter au plus une tâche. Pour cela, chaque processeur est remplacé par un ensemble de nouveaux processeurs avec la même capacité de communication que le processeur originel, mais avec des capacités de calcul différentes. Ces nouveaux processeurs sont ordonnés par valeur croissantes de leurs bandes passantes depuis le maître, puis un algorithme glouton est utilisé pour allouer les tâches sur un maximum de processeurs. Voir [12] [61, section 3.5] pour une description plus précise et une preuve.

Dans le cas où chaque tâche dépend d'une seule donnée non partagée, le problème est NP-complet, même si la plate-forme est homogène et n'a que deux processeurs (situation (d) de la figure 3.4). Même s'il n'y a aucune donnée, ce problème reste NP-complet : dans ce cas, il se réduit à ordonnancer des tâches indépendantes

sur une machine à deux processeurs. Comme les tâches ont des tailles différentes, cela se réduit au problème 2-Partition [38, problème SP12] : étant donné un ensemble de valeurs, est-il possible de partitionner cet ensemble en deux sous-ensembles tels que les sommes des valeurs de ces sous-ensembles soient égales ? Ce résultat de NP-complétude ne tient cependant pas au sens fort : la taille des tâches joue un rôle clef dans la preuve et il y a des algorithmes pseudo-polynomiaux pour résoudre ce cas simple où il n'y a pas de données (cf. l'algorithme pseudo-polynomial pour 2-Partition dans [38, section 4.2]).

Des données sont partagées

Le théorème suivant donne un résultat intéressant : dans le cas où toutes les tâches et toutes les données ont une taille unitaire (c'est-à-dire que $d_i = t_i = 1$), le problème TSF2-Dec reste NP-complet. Remarquons que dans ce cas, l'hétérogénéité vient uniquement de la plate-forme d'exécution. Cela correspond à la situation (e) de la figure 3.4.

Définition 3.2 (TSF2-Equal-Dec$(\mathcal{G}, \mathcal{P}, p = 2, d_i = t_i = 1, K)$**).** Étant donnés un graphe biparti d'application $\mathcal{G}$ tel que $d_i = t_i = 1$ pour toutes les tâches et toutes les données, une plate-forme hétérogène $\mathcal{P}$ avec deux esclaves ($p = 2$) et une borne de temps K, est-il possible d'ordonnancer toutes les tâches en K unités de temps ?

Theorème 3.2. TSF2-Equal-Dec$(\mathcal{G}, \mathcal{P}, p = 2, d_i = t_i = 1, K)$ *est NP-complet.*

Démonstration. Il est évident que TSF2-Equal-Dec$(\mathcal{G}, \mathcal{P}, p = 2, d_i = t_i = 1, K)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction de Clique qui est NP-complet [38, problème GT19]. Considérons une instance arbitraire $\mathcal{I}_1$ de Clique : étant donnés un graphe $G_c = (V_c, E_c)$ et une borne B, y a-t-il une clique dans G_c (c'est-à-dire un sous-graphe entièrement connecté) de taille B ? Sans perte de généralité, nous supposons que $|V_c| \geq 9$ et $6 \leq B \cdot (B-1) < |E_c|$.

L'instance $\mathcal{I}_2$ de TSF2-Equal-Dec$(\mathcal{G}, \mathcal{P}, p = 2, d_i = t_i = 1, K)$ que nous construisons est la suivante. Nous posons $\mathcal{D} = V_c \cup X$ et $\mathcal{T} = E_c \cup \{T_y\}$ (cf. figure 3.6) ce qui définit $\mathcal{V} = \mathcal{D} \cup \mathcal{T}$. Ici, X est une collection de $x = |X|$ données additionnelles, il y a donc un total de $|V_c| + x$ données, une par nœud dans le graphe original G_c et une par nouvelle donnée dans X. Pour ce qui est des tâches, il y a autant de tâches qu'il y a d'arêtes dans le graphe original G_c, plus une tâche additionnelle T_y.

Les relations entre les tâches et les données sont définies comme suit. Premièrement, il y a une arête de chaque donnée dans $\mathcal{D}$ à la tâche T_y ; par conséquent, le processeur esclave qui exécutera T_y aura besoin d'avoir reçu toutes les données avant de pouvoir commencer l'exécution de T_y. Deuxièmement, il y a une arête

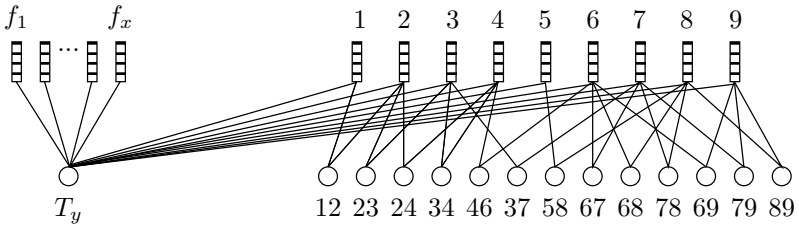


Figure 3.6 – Le graphe biparti d'application utilisé dans la preuve du théorème 3.2.

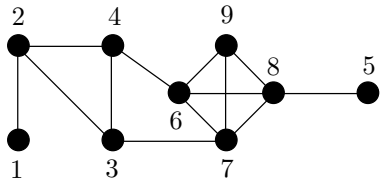


Figure 3.7 – Le graphe original G_c utilisé pour construire le graphe biparti de la figure 3.6.

d'un nœud (correspondant à une donnée) $V \in V_c \subset \mathcal{D}$ à un nœud (correspondant à une tâche) $e \in E_c \subset \mathcal{T}$ si et seulement si v était une des extrémités de l'arête e dans G_c. Dans la partie droite de la figure 3.6, le graphe biparti a été obtenu à partir du graphe original G_c présenté sur la figure 3.7. Les données sont les nœuds de G_c et les tâches sont les arêtes de G_c. C'est pourquoi chaque tâche (arête) dépend exactement de deux données (les extrémités de l'arête). Nous pouvons voir que G_c contient une clique de taille $B = 4$ (nœuds 6 à 9).

Comme cela a été spécifié dans le problème, toutes les tâches et toutes les données ont une taille unitaire. Pour compléter la description de l'application, nous posons $s = B \cdot (B-1)/2$, $r = |E_c| - s$ (notons que par hypothèse, $s < r$) et nous définissons $x = (3r-1) \cdot |V_c| - 2B + 2$. Nous vérifions que $x \geq 1$: nous avons $r \geq 4$ et $|V_c| \geq B$, nous dérivons donc $x \geq 9B + 2$.

Il reste à décrire la plate-forme d'exécution. Les caractéristiques des deux processeurs esclaves sont : $w_1 = 3 \cdot |V_c|$, $w_2 = (3 \cdot (r+1) \cdot |V_c| - 4)/s$, $c_1 = 1$ et $c_2 = 2$. Notons que $w_2 > w_1 > 2$ car $w_2 - w_1 = 3 \cdot (r/s - 1) \cdot |V_c| + (3 \cdot |V_c| - 4)/s > 0$. Finalement, nous définissons la borne pour l'ordonnancement :

$$K = 2 + 3 \cdot (r+1) \cdot |V_c| = 6 + s \cdot w_2.$$

Il est clair que l'instance $\mathcal{I}_2$ peut être construite en temps polynomial en la taille de $\mathcal{I}_1$. Nous avons maintenant à montrer que $\mathcal{I}_2$ admet une solution si et seulement si $\mathcal{I}_1$ en admet une.

Supposons d'abord que $\mathcal{I}_1$ a une solution, c'est-à-dire que G_c possède une clique de taille B. Soit $\mathcal{C} = \{v_1, v_2, \ldots, v_B\}$ l'ensemble des B sommets de la clique de G_c. L'idée intuitive est la suivante : après avoir envoyé à l'esclave P_2 les B données correspondant aux B nœuds de $\mathcal{C}$, P_2 sera capable de traiter les s tâches correspondant aux arêtes reliant les nœuds de $\mathcal{C}$ sans avoir besoin de recevoir de données supplémentaires de la part du maître. Pendant ce temps, P_1 devra traiter toutes les autres tâches. Notons que avant de pouvoir traiter la tâche T_y, P_1 devra avoir reçu toutes les données. L'ordonnancement est défini comme suit :

- D'abord, aux pas de temps $t = 0$ et $t = 1$, deux données sont envoyées par le maître à P_1. Ces deux données sont choisies de manière à ce qu'elles correspondent à deux nœuds quelconques v_a et v_b de V_c qui sont connectés dans G_c (c'est-à-dire que l'arête (v_a, v_b) appartient à E_c) et qui n'appartiennent pas tous les deux à la clique $\mathcal{C}$. Notons qu'une telle arête existe car le nombre d'arêtes dont les deux extrémités sont dans $\mathcal{C}$ est $B \cdot (B-1)/2 < |E_c|$ (par hypothèse). Au pas de temps $t = 2$, P_1 est capable de commencer l'exécution de la tâche correspondant à l'arête (v_a, v_b) et il termine cette tâche au pas de temps $2 + w_1 = 2 + 3 \cdot |V_c|$.
- Les B données correspondant à la clique $\mathcal{C}$ sont ensuite envoyées à P_2. Dès qu'il a reçu deux données, P_2 peut commencer à exécuter une tâche (les deux données correspondent à deux nœuds connectés dans G_c, la tâche qui représente l'arête entre eux est donc prête à être exécutée). P_2 a reçu toutes les B données au pas de temps $2c_1 + B \cdot c_2 = 2 + 2B$, c'est-à-dire avant d'avoir fini l'exécution de la première tâche au pas de temps $2c_1 + 2c_2 + w_2 = 6 + w_2 > 6 + w_1 = 6 + 3 \cdot |V_c| \geq 6 + 3B$ (car $B \leq |V_c|$). Ainsi, P_2 peut traiter les s tâches correspondant aux arêtes de la clique $\mathcal{C}$ sans interruption (c'est-à-dire sans attendre de recevoir plus de données), jusqu'à l'unité de temps $2c_1 + 2c_2 + s \cdot w_2 = 6 + s \cdot w_2 = K$.
- Finalement, après avoir envoyé les B données à P_2, toutes les données sauf deux sont envoyées à P_1 : nous n'avons pas besoin d'envoyer les deux premières données une nouvelle fois, mais nous envoyons toutes les autres, c'est-à-dire $|V_c| - 2 + x$ données. Nous envoyons les $|V_c| - 2$ données correspondant aux nœuds de V_c avant les x données correspondant aux nœuds de X. Quand P_1 termine sa première tâche, au pas de temps $2 + 3 \cdot |V_c|$, il a déjà reçu les $|V_c|$ premières données (la dernière est reçue au pas de temps $2c_1 + B \cdot c_2 + (|V_c| - 2) \cdot c_1 = |V_c| + 2B$). P_1 peut alors traiter les r tâches correspondant aux arêtes de G_c qui n'appartiennent pas à la clique $\mathcal{C}$ sans interruption jusqu'au pas de temps $2c_1 + r \cdot w_1 = K - w_1$. À cette date, P_1 vient de recevoir les x dernières données car $(|V_c| + x) \cdot c_1 + B \cdot c_2 = K - w_1$. P_1 traite alors la dernière tâche T_y et l'ordonnancement se termine en K unités de temps.

Nous avons donc dérivé une solution valide pour notre instance $\mathcal{I}_2$ du problème d'ordonnancement.

Supposons maintenant que $\mathcal{I}_2$ a une solution. Nous procédons en plusieurs étapes :

1. P_1 exécute obligatoirement la tâche T_y. Si ce n'était pas le cas, P_2 l'exécuterait mais comme T_y a besoin de $|V_c|+x$ données, le temps nécessaire à P_2 serait au moins de $(|V_c|+x)\cdot c_2+w_2 = 2\cdot(K-w_1-2B)+w_2 > 2\cdot(K-5\cdot|V_c|) > K$ (car $K \geq 15 \cdot |V_c|$), une contradiction.
2. P_1 ne peut pas exécuter plus de $(K-2c_1)/w_1 = r+1$ tâches car il faut envoyer deux données avant de pouvoir commencer à exécuter la première tâche.
3. Toutes les données envoyées par le maître après le pas de temps $K-w_1$ sont inutiles, car les tâches qu'elles permettraient d'exécuter ne seraient pas terminée au pas de temps K, ni par P_1, ni par P_2 (rappelons que $w_2 > w_1$). Comme P_1 exécute T_y, il reçoit $|V_c|+x$ données. Mais $K-w_1 = (|V_c|+x)\cdot c_1 + B\cdot c_2$, P_2 ne peut donc pas recevoir plus de B données du maître.
4. P_2 ne peut pas exécuter plus de s tâches car $(K-2c_2)/w_2 = (K-6)/w_2 + 2/w_2 = s + 2/w_2 < s+1$.

Au final, un total de $r+s+1$ tâches sont exécutées. Comme P_1 ne peut pas exécuter plus de $r+1$ tâches et P_2 plus de s tâches, ils exécutent respectivement $r+1$ et s tâches. P_2 exécute s tâches et ne reçoit pas plus de B données : ces données définissent une clique de taille B dans G_c et fournissent donc une solution à $\mathcal{I}_1$. □

Nous avons finalement montré que le problème de décision associé à TASKS-SHARINGFILES est NP-complet, même dans les cas simples où :

(i) il n'y a qu'un seul processeur, mais les tâches et les données sont de tailles hétérogènes ; ou

(ii) 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.

À l'heure actuelle, nous ne connaissons pas la complexité du problème dans le cas où la plate-forme est homogène et où les tâches et les données sont de taille unitaire (situation (g) de la figure 3.4). Nous ne la connaissons pas, même dans le cas où il n'y a qu'un seul processeur.

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, dans la section suivante, nous concevons des heuristiques polynomiales pour résoudre le problème TasksSharingFiles. Nous évaluerons ensuite leurs performances grâce à de nombreuses simulations.

3.4 Heuristiques

Dans cette section, nous présentons différentes heuristiques pour résoudre notre problème. Comme cette partie de notre travail était originalement motivée par les travaux de Casanova, Legrand, Zagorodnov et Berman [25, 24], nous commençons par rappeler les heuristiques utilisées par ces auteurs. Nous introduisons ensuite plusieurs nouvelles heuristiques dont l'objectif est d'obtenir une qualité de résultat (qu'on espère) équivalente mais avec une complexité algorithmique moindre.

3.4.1 Heuristiques de référence

Nous appelons *heuristiques de référence*, les heuristiques utilisées par Casanova *et al.* [25, 24]. Ce sont les heuristiques auxquelles nous allons comparer nos nouvelles heuristiques.

Structure des heuristiques

Toutes les heuristiques de référence sont construites sur le modèle présenté par l'algorithme 3.1 : tant qu'il reste des tâches à ordonnancer, une fonction objectif est évaluée pour tous les couples de tâches (restant à ordonnancer) et de processeur ; la tâche qui sera effectivement ordonnancée, ainsi que le processeur cible, sont choisis suivant la valeur de cette fonction objectif.

Algorithme 3.1 – Structure des heuristiques de référence.

```
1  A ← T                                  ▷ A est l'ensemble des tâches restant à ordonnancer
2  tant que A ≠ ∅ faire
3  |  pour chaque processeur P_i faire
4  |  |  pour chaque tâche T_j ∈ A faire
5  |  |  |  évaluer Objectif(T_j, P_i)
6  |  |  fin
7  |  fin
8  |  choisir, par rapport à Objectif(T_j, P_i), le « meilleur » couple composé d'une
   |  tâche T_j ∈ A et d'un processeur P_i
9  |  ordonnancer au plus tôt T_j sur P_i
10 |  supprimer T_j de A
11 fin
```

Fonction objectif

Pour toutes les heuristiques, la fonction objectif est la même. $\textsc{Objectif}(T_j, P_i)$ 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. Le calcul de ce temps de fin prend bien sûr en compte :

(i) les données requises par T_j qui sont déjà disponibles sur P_i (nous supposons que toute donnée qui a déjà été envoyée au processeur P_i est toujours disponible et n'a pas besoin d'être envoyée à nouveau) ;

(ii) le temps nécessaire au maître pour envoyer les autres données à P_i, sachant quelles communications sont déjà ordonnancées ;

(iii) les tâches déjà ordonnancées sur P_i.

Choix de la tâche

Les cinq heuristiques de référence diffèrent uniquement par la définition du « meilleur » couple (T_j, P_i). Plus précisément, elle diffèrent uniquement par la définition de la « meilleure » tâche. La « meilleure » tâche est toujours placée sur le processeur qui lui est le plus favorable (que nous notons $\pi(T_j)$), c'est-à-dire sur le processeur qui minimise la fonction objectif :

$$\textsc{Objectif}(T_j, \pi(T_j)) = \min_{1 \leq q \leq p} \textsc{Objectif}(T_j, P_q)$$

Voici le critère utilisé pour chacune des heuristiques de référence :

min-min : la « meilleure » tâche T_j est celle minimisant la fonction objectif quand elle est placée sur son processeur le plus favorable ; intuitivement, les plus petites tâches sont ordonnancées d'abord pour éviter les trous au début de l'ordonnancement :

$$\textsc{Objectif}(T_j, \pi(T_j)) = \min_{T_k \in \mathcal{A}} \min_{1 \leq l \leq p} \textsc{Objectif}(T_k, P_l).$$

max-min : la « meilleure » tâche est celle dont la fonction objectif, sur son processeur le plus favorable, est la plus grande ; l'idée est qu'une longue tâche ordonnancée à la fin retarderait la date de fin de toute l'exécution :

$$\textsc{Objectif}(T_j, \pi(T_j)) = \max_{T_k \in \mathcal{A}} \min_{1 \leq l \leq p} \textsc{Objectif}(T_k, P_l).$$

sufferage : la « meilleure » tâche est celle qui serait le plus pénalisée si elle n'était pas placée sur son processeur le plus favorable, c'est-à-dire que la « meilleure » tâche est celle maximisant :

$$\min_{P_q \neq \pi(T_j)} \textsc{Objectif}(T_j, P_q) - \textsc{Objectif}(T_j, \pi(T_j))$$

avec

$$\textsc{Objectif}(T_j, \pi(T_j)) = \min_{1 \leq l \leq p} \textsc{Objectif}(T_j, P_l).$$

sufferage II et **sufferage X :** ce sont des variantes de l'heuristique *sufferage*. La pénalité d'une tâche n'est plus calculée en utilisant le deuxième processeur le plus favorable, mais en considérant le premier processeur induisant une augmentation significative du temps de fin.

Plus précisément, pour une tâche donnée, les processeurs sont triés par valeur croissante des temps de fin. L'accroissement du temps de fin est calculé pour chaque paire de processeurs consécutifs suivant cet ordre. La valeur du premier accroissement qui est plus grand que la moyenne des accroissements plus un écart type est alors prise comme pénalité pour la tâche. Alors que l'heuristique *sufferage X* sélectionne simplement la tâche dont la pénalité est la plus grande, l'heuristique *sufferage II* sélectionne plutôt la tâche pour laquelle l'accroissement significatif est atteint le plus vite.

Complexité

La boucle à la ligne 3 de l'algorithme 3.1 évalue la fonction objectif pour toute paire de tâche (dans $\mathcal{A}$) et de processeur. Pour un processeur et une tâche donnés, il faut évaluer le temps de calcul de la tâche sur le processeur, ainsi que le temps de communication de toutes les données dont dépend la tâche. Il y a au plus $|\mathcal{E}|$ relations de dépendance entre les tâches de $\mathcal{A}$ et les données ($\mathcal{E}$ est l'ensemble des arêtes représentant les relations entre les tâches et les données, cf. section 3.2.1). Le calcul de la fonction objectif, pour l'ensemble des tâches de chaque processeur, a donc un complexité au pire cas de $O(|\mathcal{A}| + |\mathcal{E}|)$. En multipliant cette expression par le nombre de processeurs, puis en sommant ceci pour des valeurs de $|\mathcal{A}|$ variant de 1 à n, on trouve la complexité totale de l'heuristique qui est :

$$O(p \cdot n \cdot (n + |\mathcal{E}|)),$$

sauf pour *sufferage II* et *sufferage X*, car les processeurs doivent être triés pour chaque tâche (complexité de $O(p \cdot \log p)$), aboutissant à une complexité de

$$O(p \cdot n \cdot (n \cdot \log p + |\mathcal{E}|)).$$

3.4.2 Structure des nouvelles heuristiques

En définissant de nouvelles heuristiques, nous avons fait particulièrement attention à en réduire la complexité. Les heuristiques de référence sont très coûteuses

sur de gros problèmes. Nous avons cherché à construire des heuristiques qui sont d'un ordre de grandeur plus rapides que les heuristiques de référence, tout en essayant de préserver la qualité de l'ordonnancement produit. Pour réduire la durée du calcul des ordonnancements, Casanova *et al.* [24] avaient suggéré, plutôt que de considérer toutes les tâches dans leur ensemble, de grouper les tâches par paquets, puis de faire l'ordonnancement paquet de tâches par paquet de tâches. Une technique similaire pourrait évidemment être utilisée pour réduire encore le temps d'exécution de nos nouvelles heuristiques.

Afin d'éviter la boucle sur toutes les paires de processeur et de tâche de la ligne 3 de l'heuristique de référence, nous avons besoin d'être capable 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. Par exemple, le critère de Johnson [52] qui sépare les tâches en deux groupes (temps de communication inférieur ou supérieur au temps de calcul, cf. section 3.3.1) dépend des caractéristiques du processeur. C'est pourquoi nous construisons une liste triée de tâches pour chacun des processeurs. Notons que 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 d'*une seule tâche*. La tâche évaluée sur un processeur est la première tâche, dans la liste triée de tâches du processeur, qui n'a pas encore été ordonnancée. Notons que pour deux processeurs donnés, l'ordre des tâches dans leurs listes de tâches peut être différent. Ce n'est donc pas forcément la même tâche qui est évaluée sur chacun des processeurs. Nous choisissons ensuite le couple tâche/processeur avec le temps de fin le plus petit. Cette dernière étape nous permet d'équilibrer la charge sur les processeurs. Nous obtenons de cette manière la structure des heuristiques présentée par l'algorithme 3.2.

Il nous reste à définir les fonctions objectifs utilisées pour trier les tâches. Ce sera l'objet de la prochaine section.

3.4.3 Les fonctions objectifs

L'idée intuitive ayant conduit aux six fonctions objectifs suivantes[2] est plutôt simple :

duration : nous considérons uniquement le temps d'exécution de la tâche comme

2. Comme pour les noms des problèmes, nous avons choisi de conserver les appellations originelles des heuristiques.

Algorithme 3.2 – Structure des nouvelles heuristiques.

```
pour chaque processeur P_i faire
    pour chaque tâche T_j ∈ 𝒯 faire
        évaluer Objectif(T_j, P_i)
    fin
    construire la liste L(P_i) des tâches triées selon la valeur de Objectif(T_j, P_i)
fin
tant que il reste des tâches à ordonnancer faire
    pour chaque processeur P_i faire
        prendre T_j, la première tâche de L(P_i) qui n'a pas encore été ordonnancée
        évaluer CompletionTime(T_j, P_i)
    fin
    prendre le couple de tâche et de processeur (T_j, P_i) minimisant
    CompletionTime(T_j, P_i)
    ordonnancer au plus tôt T_j sur P_i
    marquer T_j comme ordonnancée
fin
```

si elle était la seule tâche à être ordonnancée sur la plate-forme :

$$\textsc{Objectif}(T_j, P_i) = t_j \cdot w_i + \sum_{e_{k,j} \in \mathcal{E}} d_k \cdot c_i.$$

Les tâches sont triées par valeurs croissantes de la fonction objectif, ce qui imite l'heuristique *min-min*.

payoff : en plaçant une tâche sur un processeur, le temps passé par le maître à envoyer les données nécessaires est payé par tous les processeurs, car le maître ne peut envoyer des données qu'à un seul esclave à la fois. En contrepartie, l'ensemble des processeurs y gagnent, car ils n'auront pas à exécuter cette tâche. La fonction objectif suivante encode le gain obtenu en ordonnançant la tâche T_j sur le processeur P_i :

$$\textsc{Objectif}(T_j, P_i) = \frac{t_j}{\sum_{e_{k,j} \in \mathcal{E}} d_k}.$$

Les tâches sont triées par gains décroissants. Notons que la fonction objectif pour calculer le gain de l'ordonnancement de la tâche T_j sur le processeur P_i devrait en réalité être : $\textsc{Objectif}(T_j, P_i) = t_j \cdot w_i / (\sum_{e_{k,j} \in \mathcal{E}} d_k \cdot c_i)$; mais comme les facteurs w_i et c_i ne changent pas l'ordre des tâches sur un processeur donné, nous avons simplement supprimé ces facteurs. De plus, l'ordre des tâches ne dépend pas du processeur, nous n'avons donc besoin que d'une seule liste avec cette fonction objectif.

advance : pour conserver un processeur occupé, nous avons besoin de lui envoyer toutes les données requises par la prochaine tâche qu'il aura à traiter avant qu'il ne termine l'exécution de la tâche courante. La durée d'exécution de la tâche courante doit donc être plus grande que le temps nécessaire à transférer les données. Nous avons tenté d'encoder ces contraintes en considérant la différence entre le temps de calcul et le temps de communication d'une tâche, d'où la fonction objectif :

$$\textsc{Objectif}(T_j, P_i) = t_j \cdot w_i - \sum_{e_{k,j} \in \mathcal{E}} d_k \cdot c_i.$$

Les tâches sont triées par valeurs décroissantes de la fonction objectif.

Johnson : nous trions les tâches sur chaque processeur comme le fait Johnson pour un *flow-shop* à deux machines (cf. section 3.3.1).

communication : comme les communications peuvent être un goulot d'étranglement, nous considérons le temps nécessaire à l'envoi des données dont dépend une tâche, comme si c'était la seule tâche à devoir être ordonnancée sur la plate-forme :

$$\textsc{Objectif}(T_j, P_i) = \sum_{e_{k,j} \in \mathcal{E}} d_k.$$

Les tâches sont triées par valeurs croissantes de la fonction objectif, comme pour *duration*. Comme pour *payoff*, la liste triée est indépendante du processeur et nous n'avons besoin que d'une seule liste triée avec cette fonction objectif. Cette fonction objectif simple est inspirée des travaux de Beaumont, Legrand et Robert sur l'ordonnancement de tâches homogènes sur plates-formes hétérogènes [12].

computation : symétriquement, nous considérons le temps d'exécution d'une tâche comme si elle ne dépendait d'aucune donnée :

$$\textsc{Objectif}(T_j, P_i) = t_j.$$

Les tâches sont ordonnées par valeurs croissantes de la fonction objectif. Une fois de plus, la liste triée est indépendante du processeur.

3.4.4 Politiques additionnelles

Dans la définition des fonctions objectifs précédentes, nous n'avons pas pris en compte le fait que les données étaient potentiellement partagées entre les tâches. La persistance des données envoyées sur les esclaves pourrait cependant être mieux exploitée en utilisant les informations de partage des données. Ainsi, en complément des fonctions objectifs précédentes, nous ajoutons les politiques suivantes dont le but est (d'essayer) de prendre en compte le partage des données :

shared : le transfert d'une donnée bénéficie à toutes les tâches qui en dépendent. Nous exprimons cela en utilisant, dans les fonctions objectifs, des tailles pondérées pour les données. La taille pondérée d'une donnée est obtenue en divisant la taille de la donnée par le nombre de tâches qui en dépendent. Par exemple, la fonction objectif pour l'heuristique *duration+shared* est :

$$\textsc{Objectif}(T_j, P_i) = t_j \cdot w_i + \sum_{e_{k,j} \in \mathcal{E}} \frac{d_k}{|\{T_l \mid e_{k,l} \in \mathcal{E}\}|} \cdot c_i.$$

readiness : pour un processeur donné P_i les tâches « prêtes » (*ready*) sont celles dont toutes les données se trouvent déjà sur P_i. Avec la politique *readiness*, s'il y a une tâche prête sur le processeur P_i à la ligne 9 de l'algorithme 3.2, nous prenons une tâche prête au lieu de la première tâche, non encore ordonnancée, de la liste triée $L(P_i)$.

locality : afin (d'essayer) de réduire la quantité de données répliquées, nous évitons de placer une tâche T_j sur un processeur P_i si certaines des données dont dépend T_j sont déjà présentes sur un autre processeur. Pour implémenter cette politique, nous modifions la ligne 9 de l'algorithme 3.2. Nous ne considérons plus simplement la première tâche non ordonnancée de $L(P_i)$, mais la première tâche non ordonnancée qui ne dépend pas de données déjà présentes sur un autre processeur. Si, après avoir parcouru toute la liste, il reste des tâches non ordonnancées, nous recommençons à partir du début de la liste avec le schéma original de sélection de tâche (première tâche non ordonnancée de $L(P_i)$).

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*.

3.4.5 Complexité

Les heuristiques fonctionnent en deux étapes : une première étape qui calcule les valeurs de la fonction objectif pour toutes les tâches sur tous les processeurs et qui construit les listes triées de tâches (ligne 1 de l'algorithme 3.2), puis une seconde étape qui alloue et ordonnance les tâches sur les processeurs (ligne 7 de l'algorithme 3.2).

Il y a $|\mathcal{E}|$ relations de dépendance entre les tâches et les données. Pour un processeur, l'évaluation des valeurs de la fonction objectif de toutes les tâches a donc une complexité de $O(n + |\mathcal{E}|)$ lorsque les données dont dépend chaque tâche sont utilisées pour le calcul (toutes les heuristiques, sauf *computation*) et de $O(n)$

sinon (heuristique *computation*). Il faut y ajouter le coût du tri des listes de tâches. La complexité de la première étape est ainsi de $O(p \cdot n \cdot \log n + p \cdot |\mathcal{E}|)$ pour les heuristiques *duration*, *advance* et *Johnson* (p listes), de $O(n \cdot \log n + |\mathcal{E}|)$ pour les heuristiques *payoff* et *communication* (une seule liste), et de $O(n \cdot \log n)$ pour l'heuristique *computation* (une seule liste). La complexité de la seconde étape est la même pour toutes les heuristiques. En notant ΔT le nombre maximal de données (plus une) dont dépend une tâche, l'évaluation du temps de fin d'une tâche sur un processeur se fait avec une complexité de $O(\Delta T)$. La complexité totale de la seconde étape est donc de $O(p \cdot n \cdot \Delta T)$. En remarquant que $n \cdot \Delta T \geq |\mathcal{E}|$ on trouve les complexités totales des heuristiques qui sont :

$$O(p \cdot n \cdot \Delta T + p \cdot n \cdot \log n)$$

pour les heuristiques utilisant plusieurs listes (*duration*, *advance* et *Johnson*), et

$$O(p \cdot n \cdot \Delta T + n \cdot \log n)$$

pour les autres (*payoff*, *communication* et *computation*). Nous avons remplacé le terme $n + |\mathcal{E}|$ dans la formule de complexité des heuristiques de référence par le terme $\log n + \Delta T$. Les résultats expérimentaux vont confirmer le gain en temps d'exécution.

Remarquons que toutes les politiques additionnelles peuvent être implémentées sans accroître la complexité des cas de base. C'est évident pour la politique *shared*. La politique *readiness* peut être implémentée sans surcoût en maintenant, pour chaque tâche sur chaque processeur, un compteur du nombre de données manquantes pour cette tâche sur ce processeur. Ce compteur est mis à jour à chaque envoi d'une donnée. Lorsqu'un de ces compteurs passe à zéro, la tâche concernée est placée dans l'ensemble des tâche prêtes pour le processeur correspondant. Pour la politique *locality*, il suffit de retenir, pour chaque tâche, où se trouvent les données déjà transférées dont elle dépend : sur un seul processeur (dont on conserve le numéro) ou bien déjà réparties sur plusieurs processeurs. Lorsqu'une donnée est envoyée à un esclave, cet indicateur est mis à jour pour toutes les tâches dépendant de cette donnée. Dans les deux cas, pour un processeur donné, on a au plus $|\mathcal{E}|$ mises à jour pour l'ensemble des tâches. Les politiques *readiness* et *locality* ajoutent donc, pour l'ensemble des processeurs, une complexité de $O(p \cdot |\mathcal{E}|)$, ce qui est complètement absorbé par la complexité totale des heuristiques (car $|\mathcal{E}| \leq n \cdot \Delta T$).

3.5 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. Pour

faire ces comparaisons sur un ensemble (que nous espérons) représentatif de plusieurs classes d'applications, nous avons testé les heuristiques sur des graphes de plate-forme et des graphes d'application (l'ensemble de tâches et des données ainsi que les relations de dépendances qui les lient) générés aléatoirement suivant divers paramètres. Étant donné le nombre d'heuristiques à évaluer, cela implique un très grand nombre d'exécutions.

Au chapitre 2, nous avions eu recours à des expériences grandeur nature. Nous avons pu voir (section 2.5) qu'il était difficile de garantir la stabilité de la plate-forme de test. Cela est dû à la nature distribuée de la plate-forme dont l'usage ne nous est pas dédié. Ce phénomène n'était pas trop gênant pour les expériences menées pour le chapitre 2, car il s'agissait alors uniquement d'illustrer les gains obtenus avec une distribution optimale des données calculée par nos algorithmes. Dans le cas présent, les expériences ont pour but de comparer les différentes heuristiques. L'instabilité latente de la plate-forme nous empêche de garantir que son état va rester le même entre deux expériences, et donc nous empêche de comparer rigoureusement les heuristiques. Des expériences en grandeur nature deviennent ainsi impraticables. C'est d'autant plus vrai si on tient compte du temps d'exécution réel de l'application ordonnancée qui est à multiplier par le nombre d'exécutions.

L'utilisation de simulations permet de s'affranchir de ces deux principaux problèmes [61, chapitre 7] :

- les conditions d'expérience (c'est-à-dire l'état de la plate-forme) sont facilement reproductibles ;
- un modèle de coût (pour les communications, comme pour l'exécution des tâches sur les processeurs) nous permet d'évaluer rapidement la durée d'exécution de l'application.

Nous avons pour cela écrit notre propre simulateur *ad hoc*. Le code consiste en quelques milliers de lignes de C++ [90], auxquelles se rajoutent une collection de scripts shell et/ou Perl [92] pour générer, suivant divers paramètres, les graphes d'application et de plates-formes, ainsi que pour extraire les résultats. Une grappe de 48 processeurs (Pentium IV Xeon 2,6 GHz) nous a permis d'effectuer un très grand nombre d'expériences que nous résumons dans cette section.

3.5.1 Plates-formes simulées

Nous avons testé nos heuristiques sur différents types de plates-formes qui ont été construites de la manière suivante :

processeurs : nous avons mesuré le temps de cycle de différents ordinateurs répartis entre Strasbourg et Lyon. De cet ensemble de valeurs, nous tirons aléatoirement des valeurs dont la différence avec la valeur moyenne est inférieure à l'écart type. Nous définissons de cette manière un ensemble hétérogène

réaliste de 20 processeurs.

liens de communication : les bandes passantes des 20 liens de communication entre le maître et les esclaves sont générées, suivant le même principe que l'ensemble des vitesses des processeurs, à partir de valeur mesurées entre des machines de Strasbourg et de Lyon.

ratio des coûts de communication et de calcul : telles qu'elles ont été générées, les valeurs absolues des bandes passantes des liens de communication et des vitesses des processeurs n'ont pas de réelle signification (en pratique, elles sont dépendantes de l'application et doivent donc être pondérées par les caractéristiques de l'application). Nous sommes plus particulièrement intéressés par les valeurs relatives des vitesses de processeurs et des bandes passantes des liens réseau. C'est pourquoi nous normalisons les moyennes des caractéristiques des processeurs et des liens de communication. Nous imposons alors le ratio entre le coût de communication et le coût de calcul, de manière à modéliser trois principaux types de problèmes : intensif en calculs (ratio = 1/10) intensif en communications (ratio = 10) et intermédiaire (ratio = 1). Les caractéristiques des processeurs et de liens de communication étant normalisées, ce ratio sera obtenu en faisant varier les moyennes des tailles des tâches et des données.

3.5.2 Graphes d'application

Nous faisons tourner les heuristiques sur les quatre types de graphe d'application suivants. Dans chaque cas, les tailles des données et des tâches sont tirées aléatoirement de manière uniforme entre 0,5 et 5. Les graphes sont schématisés sur la figure 3.8.

étoile : chaque graphe contient 100 sous-graphes en étoile, où chaque graphe en étoile est composé de 20 tâches dépendant d'une seule et même donnée (fig. 3.8(a)).

deux-un : chaque tâche dépend exactement de deux données : une donnée qui est partagée avec des autres tâches et une donnée privée (non partagée) (fig. 3.8(b)).

partitionné : le graphe est divisé en 20 morceaux de 100 tâches et dans chaque morceau, chaque tâche dépend aléatoirement de 1 à 10 données (fig. 3.8(c)). Le graphe contient donc au moins 20 composantes connexes,

aléatoire : chaque tâche dépend aléatoirement de 1 à 50 données (fig. 3.8(d)).

Notre objectif est d'utiliser des graphes représentatifs d'une grande classe d'applications. Les graphes en étoile représentent des applications massivement parallèles. Les graphes de type deux-un proviennent des articles Casanova *et al.* [25, 24].

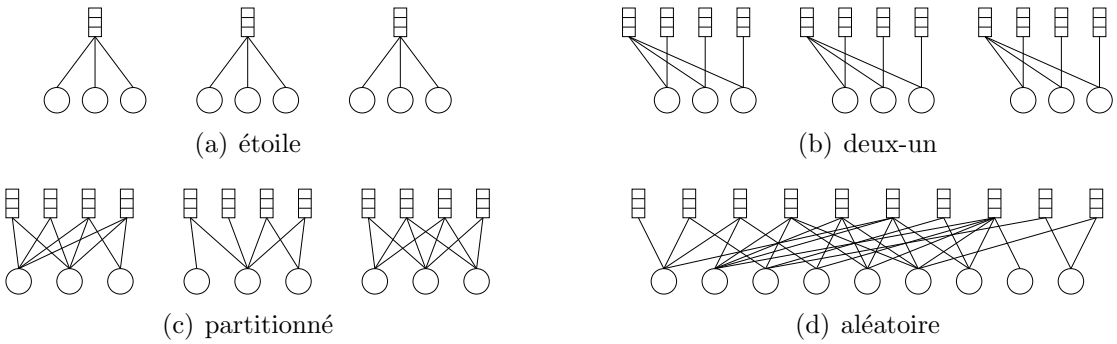


Figure 3.8 – Les quatre types de graphes d'application utilisés pour les simulations.

Les graphes partitionnés traitent d'applications présentant quelque régularité. Les graphes aléatoires représentent des applications totalement irrégulières. 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. Pour éviter toute interférence entre les caractéristiques du graphe et le ratio des coûts de communication et de calcul, nous normalisons les ensembles de tâches et de données de manière à ce que la somme des tailles des données soit égale à la somme des tailles des tâches multipliée par le ratio des coûts de communication et de calcul.

3.5.3 Résultats

Le tableau 3.1 résume l'ensemble des expérimentations. Dans ce tableau, nous rapportons les dix meilleures heuristiques avec leur performance et leur coût. La

Tableau 3.1 – Performance et coût relatifs des dix meilleures heuristiques. Moyennes sur 12 000 tests ; les écarts types sont entre parenthèses.

Heuristique	Performance relative	Coût relatif
sufferage	1,110 (±14,8%)	377 (±40,7%)
min-min	1,130 (±17,5%)	419 (±45,7%)
computation+readiness	1,133 (± 9,7%)	1,57 (±27,1%)
duration+locality+readiness	1,133 (±11,4%)	1,50 (±30,3%)
duration+readiness	1,133 (±11,5%)	1,45 (±25,4%)
payoff+shared+readiness	1,138 (±11,1%)	1,50 (±40,5%)
payoff+readiness	1,139 (±11,1%)	1,25 (±20,0%)
payoff+shared+locality+readiness	1,145 (±11,0%)	1,57 (±36,8%)
payoff+locality+readiness	1,145 (±11,1%)	1,32 (±17,7%)
computation+locality+readiness	1,147 (±10,8%)	1,62 (±29,4%)

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 3.1 est un résumé de 12 000 tests aléatoires (1 000 tests sur les quatre types de graphe 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 3.1 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 appelons *optimum relatif* cette performance relative optimale de 1. Pour chacun des tests, le *coût relatif* est calculé suivant les mêmes règles en utilisant l'heuristique la plus rapide. Nous rapportons des résultats plus détaillés sur les figures 3.9 et 3.10 qui donnent une vue duale des expériences. Nous avons sélectionné 24 variantes représentatives parmi les 44 heuristiques. Sur la figure 3.9 nous rapportons les performances relatives moyennes de ces variantes, réparties par ratio des coûts de communication et de calcul. Sur la figure 3.10 nous rapportons les performances relatives moyennes de ces variantes, réparties par type de graphe. Des résultats plus détaillés peuvent être trouvés en annexe B.

Nous pouvons voir, sur le tableau 3.1, que l'heuristique *sufferage* donne les meilleurs résultats en moyenne : elle est à 11% de l'optimum relatif. Les neuf heuristiques suivantes dans le classement ne sont pas très loin : elles sont entre 13% et 14,7% de l'optimum relatif. Parmi ces neuf heuristiques, seule *min-min* est une heuristique de référence. L'heuristique *max-min* est quasiment la plus mauvaise de toutes les heuristiques : seules deux variantes de *advance* sont classées derrière. Cela s'explique par le fait que *max-min* va d'abord favoriser les tâches dont les données requises ne se trouvent sur aucun esclave, générant ainsi beaucoup de communications dès le début de l'ordonnancement, et retardant l'exécution des tâches suivantes. Les heuristiques *sufferage II* et *sufferage X* obtiennent des performances moins bonnes que le *sufferage* de base, malgré le temps supplémentaire passé à calculer l'ordonnancement. Les différentes heuristiques *sufferage* basent leurs choix d'ordonnancement sur une prédiction de ce qui se passerait si le choix en question n'était pas fait. La prédiction qui est faite ne prend pas en compte l'éventuel transfert de données dû à l'ordonnancement d'autres tâches. Ce n'est donc pas surprenant que les heuristique *sufferage II* et *sufferage X*, qui essaient de faire des prédictions à plus longue échéance, se trompent plus et obtiennent

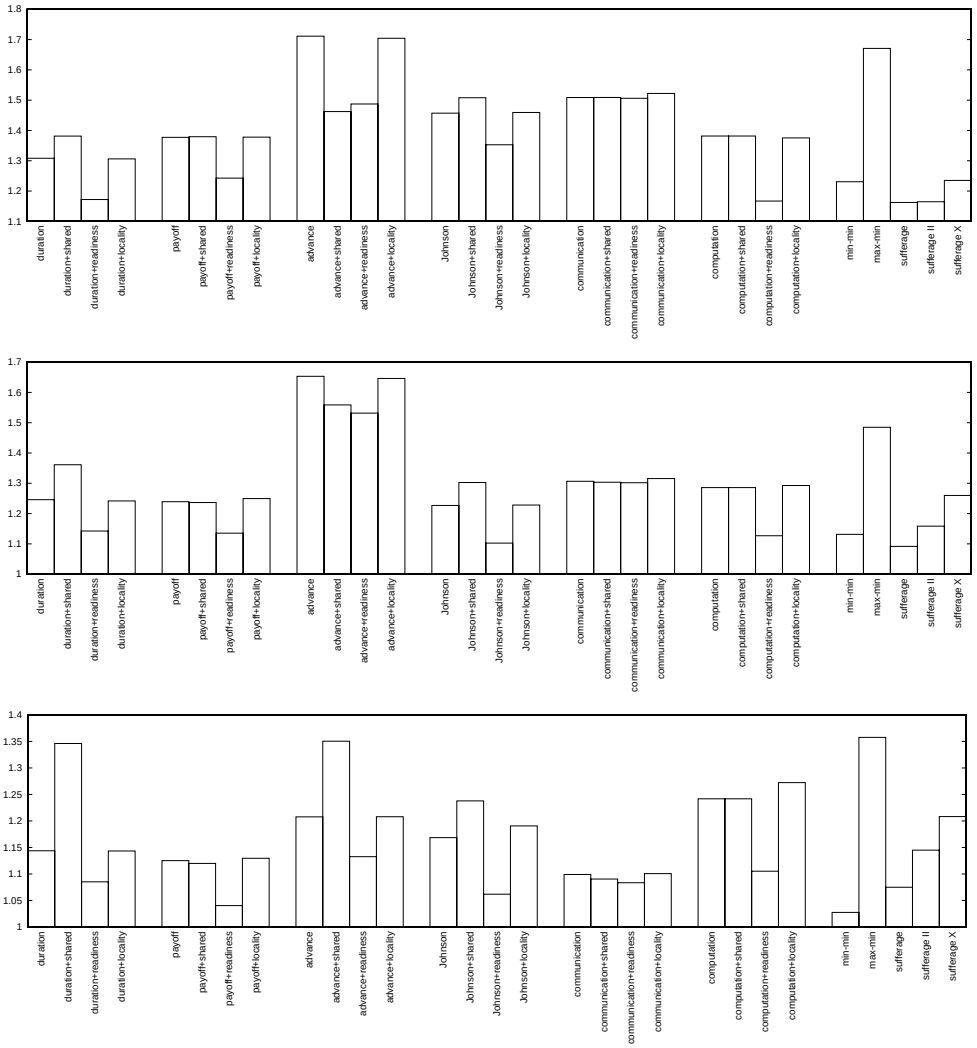


Figure 3.9 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les quatre types de graphes avec un ratio des coûts de communication et de calcul égal, de haut en bas, à : 1/10, 1 et 10.

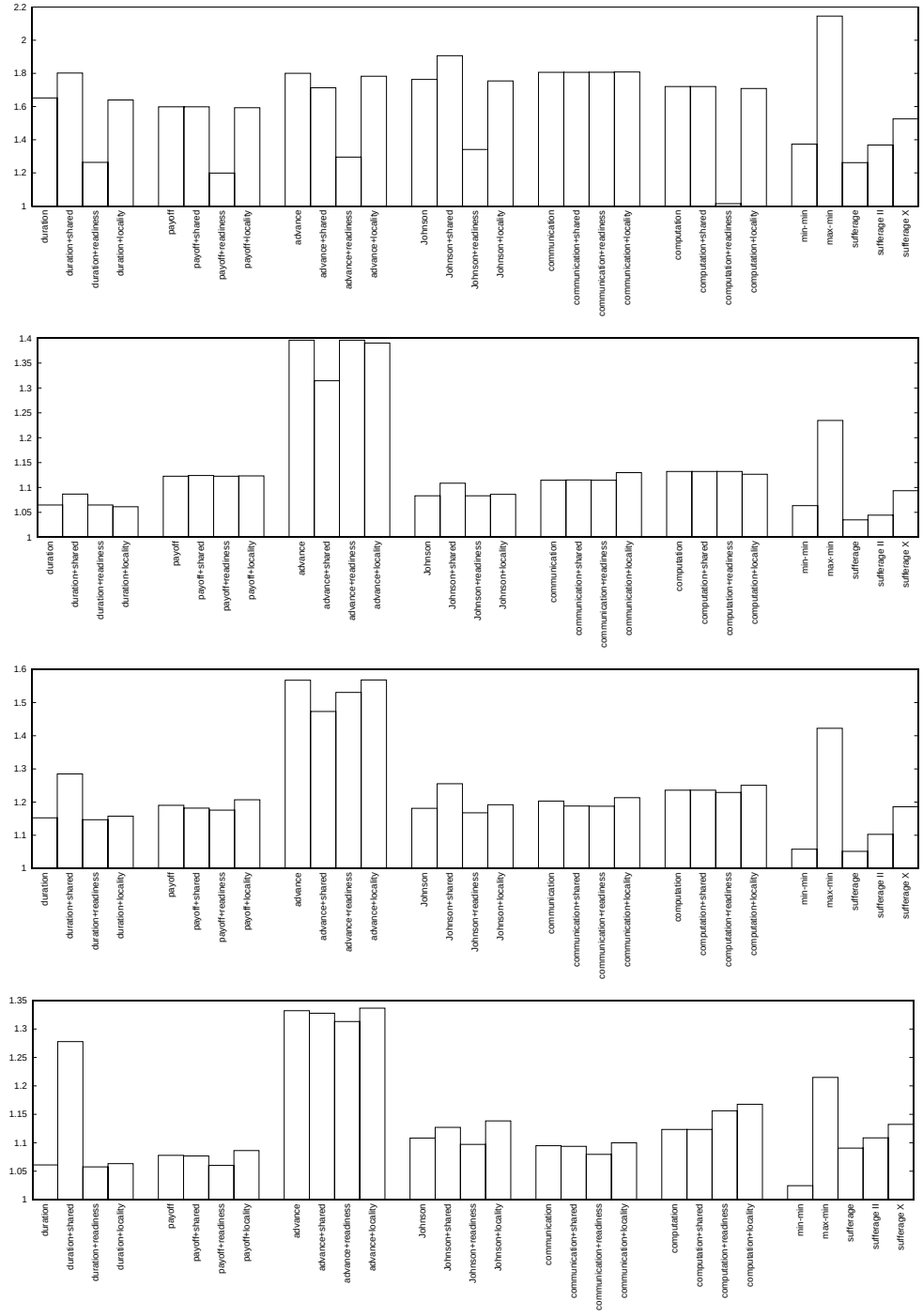


Figure 3.10 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les trois ratios des coûts de communication et de calcul avec, de haut en bas, les type de graphes d'application : *étoile*, *deux-un*, *partitionné* et *aléatoire*.

finalement de plus mauvais résultats.

En se concentrant sur nos nouvelles heuristiques, nous pouvons remarquer que les performances de celles qui apparaissent dans le tableau 3.1 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. Cela reflète une plus grande stabilité dans la qualité des résultats produits.

Ce sont les heuristiques *duration*, *computation* et *payoff* qui obtiennent (avec la politique *readiness*) les meilleures performances moyennes. L'heuristique *communication* obtient des performances bien plus mauvaises. Pour les heuristiques *computation* et *communication*, les résultats sont cependant à nuancer. Nous pouvons observer, sur la figure 3.9, que les performances de *computation* se dégradent lorsque le ratio des coûts de communication et de calcul augmente alors que c'est l'inverse pour *communication*. Cette observation n'est pas surprenante, au regard de la définition des heuristiques. La définition de *computation* ne fait intervenir que le temps de calcul des tâches, c'est donc normal qu'elle soit plus adaptée aux problèmes où ce sont les calculs qui sont prépondérants. Une explication similaire tient pour *communication*. L'heuristique *duration* qui combine les deux permet d'avoir des performances moins sensible aux différents ratios des coûts de communication et de calcul.

Les heuristiques *advance*, *payoff* et *Johnson* sont toutes les trois basées sur la même constatation qu'il faut essayer de maximiser le recouvrement des communications par les calculs. Elles obtiennent cependant des performances très différentes. Dans tous les cas, l'heuristique *advance* et ses variantes ont de mauvaises performances par rapport aux autres heuristiques. L'heuristique *payoff* est, parmi les trois, celle dont les performances sont les meilleures. Il semble que l'heuristique *advance* ait tendance à trop favoriser les tâches dont le temps de communication est important. Cette observation se vérifie également par le fait que la variante *shared* qui réduit l'importance du temps de communication lui soit favorable. Au contraire, pour que l'heuristique *payoff* favorise une tâche dont le temps de communication est important, il faut que l'avance apportée ensuite par les calculs soit proportionnellement bien plus grande. L'heuristique *Johnson* obtient quant à elle des performances intermédiaires entre *payoff* et *advance*. La meilleure variante des heuristiques *Johnson* (c'est-à-dire celle avec *readiness*) obtient une performance relative moyenne de 1,172. L'algorithme de Johnson [52] (optimal sans partage) ne s'adapte donc pas bien, dans le cas général, à des applications dont les tâches partagent des données.

Nous pouvons voir sur la figure 3.9 que, pour l'ensemble des heuristiques, la différence entre les heuristiques est plus significative quand le ratio des coûts de communication et de calcul est petit. Dans le cas opposé, il est vraisemblable que les communications du maître vers les esclaves deviennent le vrai goulot d'étran-

glement pour toutes les stratégies d'ordonnancement.

Pour ce qui est des variantes, c'est la politique *readiness* qui apporte un réel gain aux heuristiques. Par rapport aux heuristiques de base, cette variante apporte un gain en moyenne supérieur à 8%, sauf pour *communication* où le gain est négligeable. La politique *readiness*, dont l'objectif est de réduire le nombre de données répliquées, se montre donc particulièrement efficace. Pour *communication*, on peut remarquer que les tâches dépendant d'un même ensemble de données (ou presque) vont avoir tendance à être regroupées dans les listes de tâches. Ainsi lorsqu'une tâche provoque l'acheminement des données en question sur un esclave, les autres tâches suivent et se retrouvent donc en tête de liste. La politique *readiness* n'a ainsi que peu d'influence. Ce phénomène est parfaitement illustré sur les graphes en étoile. Ce sont d'ailleurs ces graphes en étoile qui marquent la différence entre les heuristiques *communication* et *computation*. Les deux heuristiques sont très mauvaises dans leur version de base, mais *computation+readiness* devient meilleure de 20% que toutes les autres heuristiques (y compris les heuristiques de référence). C'est à cause des résultats sur les graphes en étoile que *computation+readiness* se retrouve dans le tableau 3.1 alors que ce n'est pas le cas pour *communication+readiness*.

La variante *locality* ne change quasiment pas les performances des heuristiques. Il semble que cette politique, dont le but était d'améliorer la localité dans l'utilisation des données, ne soit pas assez agressive. La dernière variante, *shared*, n'a des effets sensibles que sur les heuristiques *advance*, *duration* et *Johnson* : les performances de l'heuristique *advance* sont améliorées (sans toutefois en faire une heuristique de qualité) alors que celles de *duration* et *Johnson* sont dégradées. Pour ces dernières, il est vraisemblable que la variante *shared* favorise les tâches ayant de gros temps de communication plutôt que celles dépendant de données très partagées comme nous le souhaitions. Une approche différente pour prendre en compte le partage des données serait de réévaluer le classement des tâches sur les processeurs, au fur et à mesure que des données y sont transférées. C'est ce que feront les heuristiques dynamiques dont il sera question dans le prochain chapitre.

Dans le tableau 3.1 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 leur bonnes performances comparativement à *sufferage*, nous pensons que les huit nouvelles variantes du tableau 3.1 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.

3.6 Conclusion

Nous avons traité dans ce chapitre le problème de l'ordonnancement d'un grand ensemble de tâches, indépendantes mais partageant des données, sur des plates-formes hétérogènes de type maître-esclave.

Des problèmes similaires ont déjà été étudiés par différents auteurs. Dans le cadre de fouilles de données par des algorithmes K-means, da Silva, Carvalho et Hruschka [32] 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 [33], une parallélisation de BLAST [6], 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 [83]. 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*.

Pour ce qui est de notre contribution, nous avons, d'un point de vue théorique, montré deux nouveaux résultats de complexité : nous avons montré que le partage éventuel des données rendait le problème NP-complet, que ce soit avec un seul processeur, ou bien avec deux processeurs hétérogènes si les tâches et les données sont de taille unitaire. D'un point de vue pratique, nous avons amélioré les heuristiques proposées par Casanova, Legrand, Zagorodnov et Berman [25, 24]. Nous avons réussi à concevoir des heuristiques aux performances similaires mais dont le coût en temps d'exécution est d'un ordre de grandeur plus faible. En particulier nous avons montré que la politique *readiness* est un critère important pour obtenir de bonnes performances. En l'associant avec l'heuristique *duration*, nous obtenons une heuristique qui donne, de manière stable, de bons résultats.

Nous nous sommes limités au modèle maître-esclave qui a l'inconvénient que les communications depuis le maître peuvent devenir un goulot d'étranglement. Ce travail nous a cependant permis de mettre en évidence les caractéristiques utiles pour les heuristiques. Cela va nous servir dans le prochain chapitre, où nous allons nous intéresser au cas plus général où :

– les données d'entrées sont distribuées sur plusieurs entrepôts plutôt que d'être concentrées sur un seul maître ;

– le réseau d'interconnexion est quelconque (au lieu d'une étoile jusqu'à présent) et les différents entrepôts peuvent s'échanger des données sans passer par un point central.

Nous verrons comment étendre les heuristiques *min-min* et *duration* à de telles plates-formes distribuées. Pour l'heuristique *duration*, nous verrons comment il est possible de mettre à jour les valeurs de la fonction objectif pour tenir compte de la charge induite sur les processeurs par les précédents choix d'ordonnancement. Nous verrons également s'il est envisageable de s'affranchir de la boucle d'évaluation des temps de fin des tâches sur les processeurs qui a lieu pour chaque choix d'ordonnancement.

Chapitre 4

Avec plusieurs serveurs

4.1 Introduction

Nous avons traité, dans le chapitre précédent, de l'ordonnancement d'un grand nombre de tâches hétérogènes. Ces tâches ont besoin, pour s'exécuter, d'un certain nombre de données d'entrée. Les données peuvent être partagées entre plusieurs tâches, c'est-à-dire que plusieurs tâches peuvent avoir besoin des mêmes données. La plate-forme considérée était de type maître-esclave. Le maître détenait toutes les données, son rôle était de distribuer aux esclaves les données nécessaires à l'exécution des tâches. L'objectif était d'ordonnancer les tâches sur les différents esclaves afin de minimiser le temps total d'exécution de l'ensemble des tâches. Ce modèle de plate-forme centralisé a l'inconvénient que toutes les communications se font depuis le maître. Ces communications peuvent alors devenir un goulot d'étranglement pour l'ensemble de l'ordonnancement des tâches.

Nous allons, dans ce chapitre, 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. À chaque serveur est associé un entrepôt (local) pour le stockage des données. Les données sont initialement stockées dans un ou plusieurs de ces entrepôts (certaines données peuvent être répliquées). Après avoir décidé qu'un serveur S_i aura à exécuter une tâche T_j, les données d'entrées pour T_j qui ne sont pas déjà disponibles dans l'entrepôt local de S_i seront envoyées à travers le réseau. Plusieurs transferts de données peuvent avoir lieu en parallèle suivant différents chemins.

Ce chapitre s'articule en deux parties. Comme dans le chapitre précédent, nous commençons par montrer la complexité théorique du problème. D'un point de vue pratique, nous concevons ensuite plusieurs nouvelles heuristiques. La première de ces heuristiques est une extension de l'heuristique *min-min* pour l'adapter à notre

modèle décentralisé. Cette extension se révèle difficile, et nous détaillons à la fois les problèmes rencontrés et les solutions proposées. Les autres heuristiques ont pour but de conserver les bonnes performances des variantes du *min-min*, mais avec un coût d'un ordre de grandeur plus petit. Pour la conception de ces nouvelles heuristiques, nous nous inspirons des résultats du chapitre précédent. Dans ce dernier, toutes nos heuristiques basaient leurs décisions sur l'état du système à un instant initial. Ici par contre, certaines des heuristiques nouvellement conçues sont capables de s'adapter dynamiquement à la charge des serveurs et de prendre en compte d'éventuelles réplications des données.

Dans la section 4.2, nous présentons formellement notre problème d'ordonnancement, que nous appelons TSFDR (*Tasks Sharing Files from Distributed Repositories*). Nous montrons ensuite, dans la section 4.3 que le simple fait d'ordonnancer les déplacements des données est un problème difficile. Après cette partie théorique, nous passons à la conception d'heuristiques en temps polynomial pour résoudre le problème TSFDR. Dans la section 4.4, après avoir proposé des algorithmes pour ordonnancer les communications, nous montrons comment nous avons étendu l'heuristique *min-min* à notre modèle. De nouvelle heuristiques beaucoup moins coûteuses seront proposées en section 4.5. Toutes ces heuristiques sont ensuite comparées entre elles à l'aide de nombreuses simulations dont les résultats sont exposés en section 4.6. Nous concluons finalement en section 4.7.

4.2 Modèles

Dans cette section, nous posons formellement le problème d'optimisation à résoudre. Nous développons également un petit exemple.

4.2.1 Tâches et données

Concernant les tâches et les données, nos hypothèses sont les mêmes qu'au chapitre 3. Nous conservons donc les mêmes notations (cf. section 3.2.1).

La figure 4.1 (page 74) montre un petit exemple avec $m = 11$ données et $n = 10$ tâches. On peut voir dans cet exemple (qui sera utilisé plus tard) que la tâche T_1 dépend des données D_1, D_2 et D_5 et que la données D_5 est une donnée d'entrée pour toutes les tâches T_1 à T_{10}.

4.2.2 Graphe de plate-forme

Pour les plates-formes, nous généralisons le modèle du chapitre précédent. Les tâches sont ordonnancées et exécutées sur une plate-forme hétérogène composée d'un ensemble de *serveurs* reliés entre eux par des routeurs et des liens. Nous notons

$\mathcal{P} = (\mathcal{S} \cup \mathcal{R}, \mathcal{L})$ le graphe de plate-forme où chaque nœud de $\mathcal{S} = \{S_1, \ldots, S_s\}$ est un serveur, chaque nœud de $\mathcal{R} = \{R_1, \ldots, R_r\}$ est un routeur et chaque lien $l = (u, v) \in \mathcal{L}$ représente un lien de communication de $u \in \mathcal{S} \cup \mathcal{R}$ à $v \in \mathcal{S} \cup \mathcal{R}$. Un lien de communication peut ainsi relier deux serveurs, deux routeurs ou un routeur et un serveur. Nous supposons que le graphe $\mathcal{P}$ est connecté, c'est-à-dire qu'il y a un chemin entre chaque paire de serveurs. Par défaut, nous supposons que tous les liens sont bidirectionnels, $\mathcal{P}$ est donc un graphe non orienté. La suite du travail pourrait cependant être facilement adapté à des liens orientés.

Chaque serveur $S_i = (E_i, C_i)$ 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 ne mettons pas de restriction sur la possibilité de réplication des données, ce qui signifie que chaque entrepôt est supposé être assez grand pour contenir une copie de toutes les données. La figure 4.2 (page 74) 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. Avec les notations précédentes : pour qu'une grappe C_i puisse exécuter une tâche T_j, l'entrepôt E_i doit détenir toutes les données D_k telles que $e_{k,j} \in \mathcal{E}$. Ainsi, avant que C_i puisse commencer l'exécution de T_j, le serveur S_i doit avoir reçu de la part des autres serveurs toutes les données dont dépend T_j qui n'étaient pas déjà stockées dans E_i. 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. Ce modèle est discuté plus en détails en section 4.2.5.

Le coût des communications entre deux serveurs est défini en fonction du chemin entre ces serveurs, tel qu'il est fixé par le routage. Considérons d'abord le cas de serveurs *adjacents* dans le graphe de plate-forme. Nous disons que deux serveurs S_i et S_j sont adjacents si le chemin reliant S_i à S_j ne passe par aucun serveur intermédiaire. Bien évidemment, un tel chemin peut utiliser un ou plusieurs routeurs. Par exemple, sur la figure 4.2, les serveurs S_3 et S_2 sont adjacents mais S_3 et S_1 ne le sont pas, car les communications entre S_3 et S_1 doivent passer par le serveur S_2. Supposons qu'un serveur S_i envoie une donnée F_j (stockée dans son entrepôt E_i) à un serveur adjacent S_k en utilisant un chemin composé des liens de communication $l_1, l_2, \ldots, l_{x-1}, l_x$. Nous notons par b_y la bande passante du lien l_y. La bande passante disponible pour le chemin est alors le minimum des bandes passantes des différents liens, ainsi $d_j / \min_{1 \leq y \leq x} b_y$ unités de temps sont nécessaires pour transférer la donnée. Pour les communications mettant en jeu

des serveurs intermédiaires, nous utilisons un modèle *store-and-forward* entre les serveurs : les données sont transférées de serveur adjacent en serveur adjacent, 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. Les communications à travers des serveurs intermédiaires, avec le modèle *store-and-forward* pour ces communications, nous permettent de modéliser des systèmes où, pour des raisons de sécurité ou de confidentialité, les serveurs doivent utiliser seulement des canaux de communications prédéfinis.

Nous faisons finalement la supposition que, quand des données se trouvent dans l'entrepôt d'un serveur, elles sont disponibles sans surcoût pour sa grappe. En d'autre termes, nous supposons 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. De plus, nous supposons que les seuls temps de communication sont dus à la communication des données. Nous ne considérons pas de coût de migration lors de l'allocation d'une tâche à un serveur. Si on devait cependant modéliser le fait que le code d'une tâche se trouve sur un serveur et doive également être transféré sur l'entrepôt associé à la grappe où la tâche est allouée, il suffirait d'ajouter une donnée virtuelle correspondant à ce code dans le graphe biparti des relations entre les tâches et les données.

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 $\sum_{k=1}^{p_i} s_{i,k}$. La situation où un serveur est simplement composé d'un entrepôt mais n'a pas de ressource de calcul peut aisément être modélisée : il suffit de créer une (fausse) grappe de vitesse nulle.

4.2.3 Fonction objectif

Notre objectif est de minimiser le temps total d'exécution (ou *makespan*). L'exécution est terminée lorsque la dernière tâche a été complétée. 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 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, devient une source potentielle pour la donnée.

Nous appelons TSFDR($\mathcal{G}, \mathcal{P}$) (*Tasks Sharing Files from Distributed Repositories*) le problème d'optimisation à résoudre.

4.2.4 Déroulement de l'exemple

Considérons l'exemple présenté sur les figures 4.1 (graphe d'application) et 4.2 (graphe de plate-forme). Sur la figure 4.2, nous avons indiqué le placement initial des données sur les serveurs. Supposons que le placement des tâches suivant ait été décidé, comme illustré sur la figure 4.3 :

- le serveur S_1 exécute les tâches T_9 et T_{10} ;
- le serveur S_2 exécute les tâches T_4 et T_8 ;
- le serveur S_3 exécute les tâches T_6 et T_7 ;
- le serveur S_4 exécute la tâche T_5 ;
- le serveur S_5 exécute les tâches T_1, T_2 et T_3.

Nous ne discutons pas ici comment déterminer un tel placement : c'est un point difficile dont il sera question plus loin dans ce chapitre. Nous nous concentrons plutôt sur les communications requises par ce placement. Voici la liste des données nécessaires à chacun des serveurs :

- le serveur S_1 a besoin des données D_5 et D_{10} pour T_9, et des données D_5, D_{10} et D_{11} pour T_{10} ;
- le serveur S_2 a besoin des données D_3, D_4 et D_5 pour T_4, et des données D_5 et D_9 pour T_8 ;
- le serveur S_3 a besoin des données D_5 et D_7 pour T_6, et des données D_5, D_7 et D_8 pour T_7 ;
- le serveur S_4 a besoin des données D_4, D_5 et D_6 pour T_5 ;
- le serveur S_5 a besoin des données D_1, D_2 et D_5 pour T_1, des données D_2 et D_5 pour T_2, et des données D_3 et D_5 pour T_3.

Initialement, S_1 détient D_{10} et D_{11} dans son entrepôt, il n'a donc besoin que de recevoir la donnée D_5. Remarquons que S_1 n'a besoin de recevoir D_5 qu'une seule fois, même s'il exécute deux tâches dépendant de cette donnée. En fait, comme

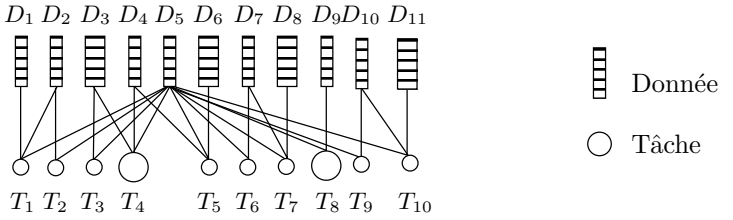


Figure 4.1 – Graphe biparti rassemblant les relations entres les tâches et les données.

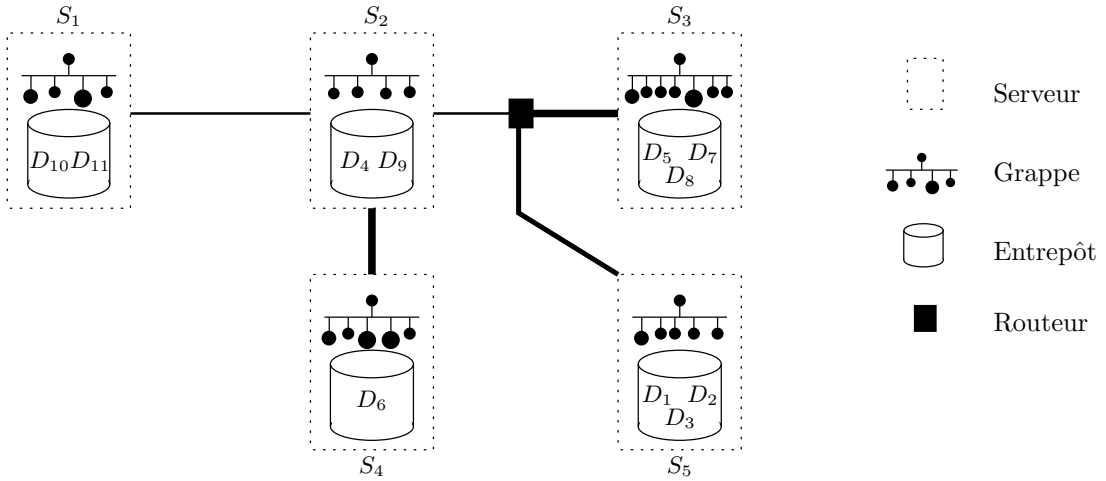


Figure 4.2 – Graphe de plate-forme avec la distribution initiale des données.

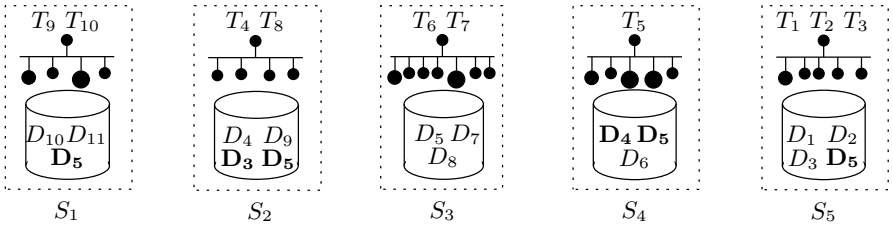


Figure 4.3 – Serveurs du graphe de plate-forme de la figure 4.2 avec les tâches allouées aux serveurs et le placement final des données (les données dupliquées sont en caractères gras).

toutes les tâches dépendent de D_5, chaque serveur sauf S_3 a besoin de recevoir D_5. En plus de D_5, S_2 a besoin de recevoir la donnée D_3, S_3 n'a besoin d'aucune autre donnée, S_4 a besoin de D_4, D_5 et D_6, alors que S_5 a seulement besoin de D_5.

Comme S_3 est la seule source pour D_5, nous avons à décider d'un ordre pour ordonnancer l'envoi de D_5 aux autres processeurs. Nous avons également à (essayer de) ordonnancer les communications indépendantes en parallèle : par exemple, le transfert de D_5 de S_3 vers S_5 et le transfert de D_4 de S_2 vers S_4 peuvent avoir lieu en parallèle. Nous revenons sur ces différents scénarios de routage et d'ordonnancement des communications en section 4.4.

Il se pourrait bien qu'à la fin de l'exécution toutes les données soient répliquées partout. Cette situation n'arrivera vraisemblablement que pour des graphes d'application bipartis fortement couplés. Les heuristiques décrites dans les sections 4.4 et 4.5 sont suffisamment efficaces pour éviter une réplication complète des données quand ce n'est pas la solution optimale. Elles pourraient aussi être aisément étendues, pour s'adapter au cas d'entrepôts avec des capacités de stockage limitées, en utilisant une politique de remplacement des données. Voir, par exemple, les travaux de Otoo et Shoshani [74] pour une étude comparative de différentes politiques de remplacement Cette dernière étude compare, dans le cadre de systèmes distribués, une nouvelle politique, LCB-K (*Least Cost Beneficial based on K backward references*), à des politiques plus classiques comme LRU (*Least Recently Used*), LFU (*Least Frequently Used*), etc.

4.2.5 Discussion du modèle

Dans cette section, nous discutons les choix qui ont été fait pour le modèle dans la section 4.2.2.

Graphe de plate-forme

Les graphes de plate-forme nous permettent de prédire les contentions ainsi que la bande passante disponible sur les liens. Ces graphes ne sont pas supposés être une représentation exacte du matériel sous-jacent. Nous considérons plutôt des représentations qui modélisent comment le réseau se comporte d'un point de vue de l'application. En particulier, les bandes passantes des liens de communication ne sont pas les capacités en crête des liens physiques, mais la bande passante effective réellement disponible pour l'application. De même, si une application est capable d'ouvrir plusieurs connexions TCP elle peut, la plupart du temps, obtenir un meilleur débit agrégé qu'avec une seule connexion [44, 2, 58]. Nous tenons compte de ces capacités dans la bande passante disponible. Des travaux comme ceux de Quinson *et al.* [63, 64, 77] sur la découverte des caractéristiques de plates-formes distribuées peuvent aider à instancier un tel modèle.

Si plusieurs communications sont ouvertes *via* un lien de dorsale (*backbone link*), chaque communication se voit attribuer (du point de vue de l'application) approximativement la même quantité de bande passante [23]. Par conséquent, si le réseau est de type WAN, et contient des liens de dorsale, nous pouvons utiliser un multigraphe où nous modélisons chaque lien de dorsale par un ensemble suffisamment grand d'arêtes de mêmes caractéristiques.

Modèle un-port

Par rapport aux chapitres précédents, nous avons généralisé la modélisation de la plate-forme. Nous avons cependant conservé le modèle un-port pour les communications des serveurs : nous avons supposé qu'à chaque instant, il y a au plus deux communications impliquant un serveur donné, une en émission et l'autre en réception (voir la discussion du modèle au chapitre 2). En fait, nous ne travaillons plus exactement dans le modèle un-port. Le graphe de plate-forme nous permet de prédire les contentions, et ainsi d'éviter la première des limitations majeures du modèle un-port, comme définies par Casanova dans [23]. Notre gestion des liens de dorsale nous permet aussi d'éviter certains inconvénients du modèle un-port. Nous n'évitons cependant pas la seconde limitation, qui dit que le partage de liens par plusieurs communications simultanées peut être bénéfique. Un modèle de partage de bande passante pourrait sembler plus réaliste, par exemple dans le cadre des tâches divisibles ou celui de l'optimisation de régime permanent. Un tel modèle serait cependant difficile à manipuler avec des graphes d'application irréguliers comme ceux considérés ici. C'est pourquoi nous choisissons d'utiliser le modèle pessimiste un-port, dont la sérialisation des communications nous permet de garantir le comportement dans le pire cas.

Pas de latences pour les communications

Les latences pour les communications sont ignorées dans notre modèle. Nous nous attendons en effet à ce que ces latences soient négligeables par rapport aux temps de transfert des données. Cette simplification nous permet de simplifier les expressions du modèle. Le temps de transfert des données est cependant lié à la bande passante des liens de communication. Il est alors possible, avec des liens très rapides, que la latence devienne non négligeable [23]. Si besoin, il n'est pas difficile d'ajouter des latences dans notre modèle. Nous indiquons, à chaque fois, comment modifier nos heuristiques pour tenir compte des latences, et nous discutons, dans les sections 4.5.1 et 4.5.3 de la complexité additionnelle induite par ces modifications.

Routage fixé

Nous supposons que le routage dans le graphe de plate-forme est fixé, ce qui est habituellement le cas pour les réseau locaux. Pour des connexions à longue distance, cette supposition n'a pas de conséquence. En effet, notre modèle s'apparente à ceux proposés par Casanova [23] ou bien Lacour, Pérez et Priol [60] dans lesquels la complexité du réseau entre deux sites distants n'est pas représentée : les connexions à longue distance se font avec un seul lien logique.

4.3 Complexité

Comme nous l'avons vu au chapitre précédent (section 3.3), à cause du partage des données, le problème est NP-complet, même dans les cas simples où :

(i) il n'y a qu'un seul processeur, mais les tâches et les données sont de tailles hétérogènes ; ou

(ii) les tâches et les données sont de tailles unitaires, mais la plate-forme est composée de deux processeurs hétérogènes avec deux liens de bandes passantes différentes.

Le but de cette section est de montrer que le simple fait de décider où déplacer les données pour exécuter les tâches est un problème combinatoire difficile, même dans le cas simple où toutes les données ont la même taille, tous les liens de communication la même bande passante et où il n'y a pas de routeur dans le graphe de plate-forme (tous les liens relient directement deux serveurs). Nous supposons que toutes les tâches ont un poids nul ou bien (ce qui est équivalent) que tous les processeurs ont une vitesse infinie. 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. Nous définissons formellement le problème de décision associé à cette instance très particulière de TSFDR comme suit :

Définition 4.1 (TSFDR-Move-Dec$(\mathcal{G}, \mathcal{P}, K)$**).** Étant donnés un graphe d'application biparti $\mathcal{G} = (\mathcal{D} \cup \mathcal{T}, \mathcal{E})$, un graphe de plate-forme $\mathcal{P} = (\mathcal{S} \cup \mathcal{R}, \mathcal{L})$ avec :

- des tailles de données uniformes ($d_i = 1$),
- un réseau d'interconnexion homogène ($l_i = 1$),
- aucun routeur ($\mathcal{R} = \emptyset$),
- des temps de calcul nuls ($t_i = 0$ ou $s_{i,k} = \infty$),

et une borne de temps K, est-il possible d'ordonnancer toutes les tâches en K unités de temps ?

Theorème 4.1. TSFDR-MOVE-DEC$(\mathcal{G}, \mathcal{P}, K)$ *est NP-complet.*

Démonstration. Il est évident que TSFDR-MOVE-DEC$(\mathcal{G}, \mathcal{P}, K)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction du problème de coupe minimale dans un graphe non orienté (MINCUT) qui est NP-complet [38, problème ND17]. Plus précisément, nous utilisons une réduction du problème MINCUT où chaque sommet a un grand degré, mais nous montrons que cette restriction, que nous notons MINCUTLARGEDEGREE, reste NP-complète :

Définition 4.2 (MINCUTLARGEDEGREE$(\mathcal{H}, B)$**).** Étant donnés un graphe non orienté $\mathcal{H} = (V, E)$ avec un nombre pair de sommets et une borne B, $1 \leq B \leq |V|$, et où chaque sommet a un degré au moins égal à $B + 1$, existe-t-il une partition $V = V_1 \cup V_2$ avec $|V_1| = |V_2| = |V|/2$ telle que le nombre d'arêtes coupées n'excède pas B :

$$|\{e = (u, v) \in E \mid u \in V_1, v \in V_2\}| \leq B\,?$$

Lemme 4.2. MINCUTLARGEDEGREE$(\mathcal{H}, B)$ *est NP-complet.*

Nous faisons la démonstration du lemme 4.2 avant de revenir à la preuve du théorème 4.1.

Démonstration. Il est évident que MINCUTLARGEDEGREE$(\mathcal{H}, B)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction du problème MINCUT original. Considérons une instance arbitraire $\mathcal{I}_1$ du problème MINCUT : étant donné un graphe $H' = (V', E')$ avec un nombre pair de sommets $|V'|$ et une borne B', existe-t-il une partition $V' = V'_1 \cup V'_2$ avec $|V'_1| = |V'_2| = |V'|/2$ telle que $|\{e = (u, v) \in E' \mid u \in V'_1, v \in V'_2\}| \leq B'\,?$ Pour construire une instance $\mathcal{I}_2$ de MINCUTLARGEDEGREE où chaque sommet a un degré au moins égal à $B + 1$, nous étendons chaque sommet $v \in V'$ de manière à former une clique « privée » de taille $B' + 2$. Les seules arêtes ajoutées sont celles de la clique. Formellement,

$$V = \{v_{i,k} \mid 1 \leq i \leq |V'|, 0 \leq k \leq B' + 1\}$$

et

$$E = \{(v_{i,0}, v_{j,0}) \mid (v_i, v_j) \in E'\} \\ \cup \{(v_{i,k}, v_{i,k'}) \mid 0 \leq k \leq B' + 1, 0 \leq k' \leq B' + 1, k \neq k', 1 \leq i \leq |V'|\}.$$

Intuitivement, $v_{i,0}$ correspond au sommet original $v_i \in V'$ et $v_{i,j}$ avec $j \geq 1$ est un nouveau sommet (dans la clique). Nous posons finalement $B' = B$. La taille de $\mathcal{I}_2$ est clairement polynomiale en la taille de $\mathcal{I}_1$.

Supposons d'abord que $\mathcal{I}_1$ a une solution, c'est-à-dire une partition équitable $V' = V'_1 \cup V'_2$ ne coupant pas plus de B' arêtes. Nous posons

$$V_1 = \{v_{i,k} \mid 0 \leq k \leq B + 1, v_i \in V'_1\}$$

et nous mettons les autres sommets dans V_2 : $V_2 = V \setminus V_1$. Cela nous donne une partition équitable de V. Le nombre d'arêtes coupées est le même que dans la solution de $\mathcal{I}_1$ car chaque clique a été entièrement placée d'un côté ou de l'autre de la partition et qu'il n'y a pas d'autres arêtes reliant les nouveaux sommets. C'est donc une solution à $\mathcal{I}_2$.

Réciproquement, supposons que $\mathcal{I}_2$ a une solution, c'est-à-dire une partition équitable $V = V_1 \cup V_2$ ne coupant pas plus de B arêtes. Dans une telle solution, chaque clique a été placée entièrement d'un côté ou de l'autre de la partition. S'il en était autrement, nous aurions, pour un indice donné i, b sommets $v_{i,k} \in V_1$ et $B + 2 - b$ sommets $v_{i,k} \in V_2$ avec $1 \leq b \leq B + 1$: une contradiction car il y aurait $b \cdot (B + 2 - b) > B$ arêtes coupées. Nous dérivons maintenant aisément une solution à $\mathcal{I}_1$: V_1' est l'ensemble des sommets v_i tels que $v_{i,0} \in V_1$, de même pour V_2'. V_1 et V_2 sont de même taille et chaque clique est placée dans un seul ensemble. V_1 et V_2 ont donc le même nombre de cliques, et V_1' et V_2' ont même cardinal. Pour finir, toutes les arêtes coupées relient deux sommets appartenant à des cliques différentes, leur nombre est donc le même dans la partition $V = V_1 \cup V_2$ que dans la partition $V' = V_1' \cup V_2'$, d'où le résultat. □

Nous revenons maintenant à la preuve de complétude de TSFDR-Move-Dec$(\mathcal{G}, \mathcal{P}, K)$. Nous partons d'une instance arbitraire $\mathcal{I}_1$ de MinCutLarge-Degree : étant donnés un graphe $H = (V, E)$ avec un nombre pair de sommets et une borne B, $1 \leq B \leq |V|$ et où chaque sommet a un degré au moins égal à $B + 1$, existe-t-il une partition $V = V_1 \cup V_2$ avec $|V_1| = |V_2| = |V|/2$ telle que $|\{e = (u, v) \in E \mid u \in V_1, v \in V_2\}| \leq B$? Posons $|V| = 2p$ et, sans perte de généralité, supposons que $p \geq 3$ et que $B \leq p - 1$. La figure 4.4 montre un exemple avec $p = 4$ et $B = 2$.

Nous construisons l'instance $\mathcal{I}_2$ de TSFDR-Move-Dec$(\mathcal{G}, \mathcal{P}, K)$ suivante. Nous associons une donnée $d_i \in \mathcal{D}$ à chaque sommet $v_i \in V$. De plus nous introduisons plusieurs nouvelles données :

- $B \cdot (p + 1)$ données $x_{i,j}$, $1 \leq i \leq B, 1 \leq j \leq p + 1$;
- $B \cdot (p - 2)$ données $y_{i,j}$, $1 \leq i \leq B, 1 \leq j \leq p - 2$;
- $p + 1$ données z_i, $1 \leq i \leq p + 1$;

de manière à avoir un total de $m = 3p + 1 + B \cdot (2p - 1)$ données dans $\mathcal{D}$.

Pour ce qui est des tâches, il y en a autant que d'arêtes dans le graphe original H, plus B tâches supplémentaires de manière à avoir $n = |E| + B$ tâches. Plus précisément, nous créons une tâche $T_{i,j}$ pour chaque arête $e = (v_i, v_j) \in E$ et nous ajoutons B tâches T_i', $1 \leq i \leq B$. Les relations entre les tâches et les données sont définies comme suit. D'abord, si $(v_i, v_j) \in E$, il y a deux arêtes dans $\mathcal{E}$, une de la donnée d_i à la tâche $T_{i,j}$ et une autre de d_j à $T_{i,j}$. En plus de ces données d_i et d_j, la tâche $T_{i,j}$ dépend de toutes les données z_k. En d'autres termes, nous ajoutons $(p + 1) \cdot |E|$ arêtes $(z_k, T_{i,j})$ dans $\mathcal{E}$. Ensuite, les B dernières tâches T_i' ont toutes

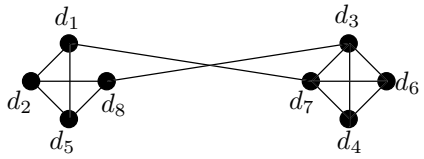


Figure 4.4 – Le graphe original pour l'instance de MinCutLargeDegree.

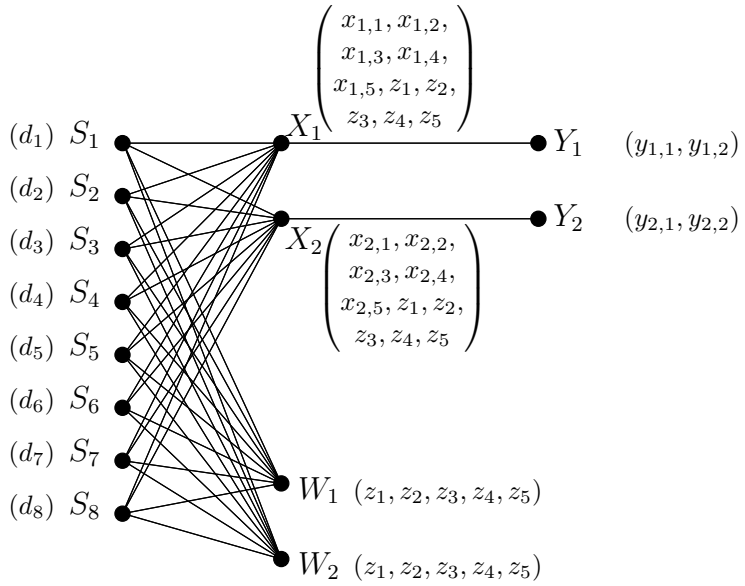


Figure 4.5 – Le graphe de plate-forme utilisé pour la réduction.

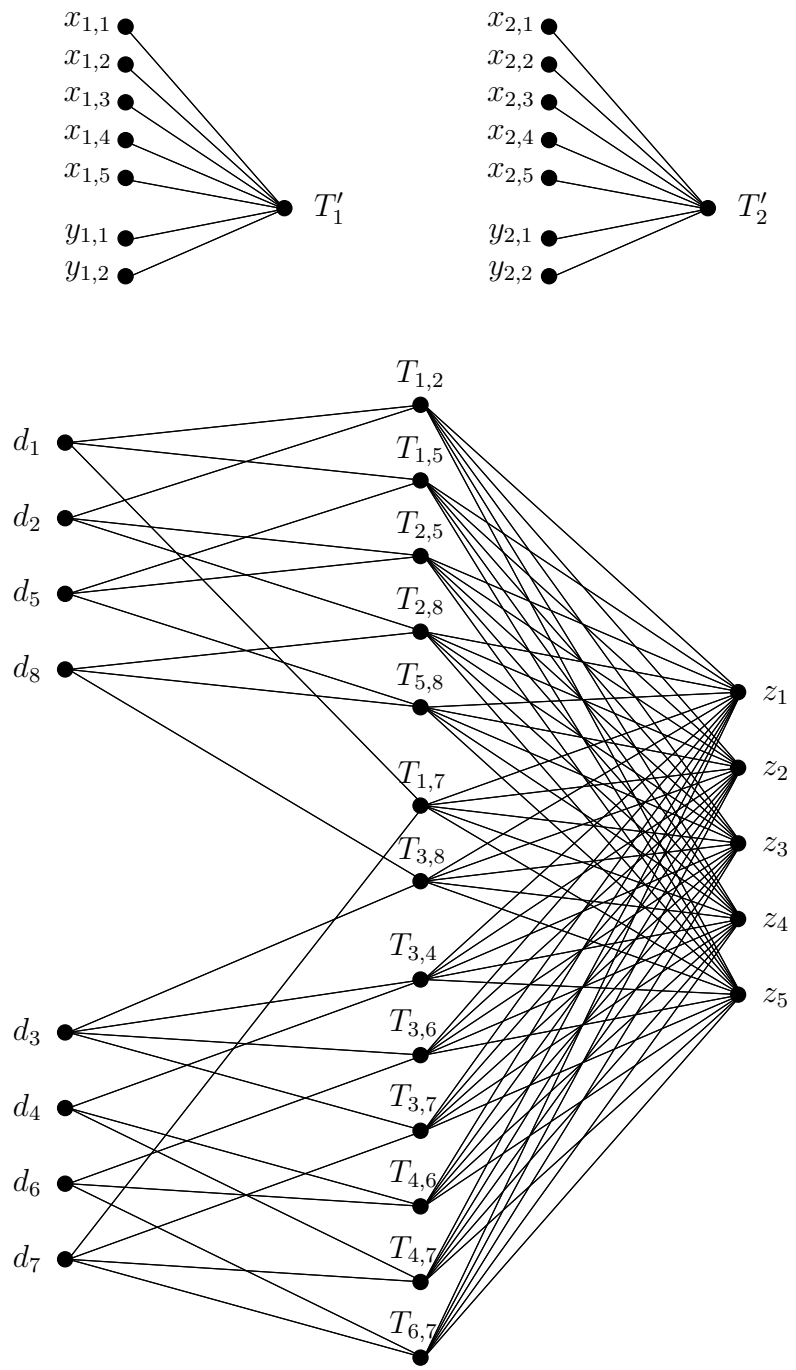


Figure 4.6 – Le graphe d'application biparti utilisé pour la réduction.

$2p-1$ données d'entrée : pour un i donné, il y a une arête de chaque $x_{i,j}$ et de chaque $y_{i,j}$ vers T'_i. Pour résumer, il y a $(p+3)\cdot|E| + B\cdot(2p-1)$ arêtes dans $\mathcal{E}$. La transformation du graphe de la figure 4.4 est illustrée sur la figure 4.6.

Il nous reste à décrire le graphe de plate-forme $\mathcal{P}$. Il y a $2p+2B+2$ serveurs dans P que nous notons S_i pour $1 \leq i \leq 2p$, X_i et Y_i pour $1 \leq i \leq B$, W_1 et W_2. Il y a un lien de communication de chaque S_i vers W_1, vers W_2 et vers chaque X_j, soit $2p\cdot(B+2)$ liens. Il y a B liens supplémentaires, un de Y_i vers X_i pour chaque i. Il y a donc un total de $2p\cdot(B+2)+B$ liens dans $\mathcal{P}$. Voir la figure 4.5 pour une illustration.

La distribution initiale des données dans les entrepôts des serveurs est la suivante :

- pour tout i, $1 \leq i \leq 2p$, la donnée d_i est stockée sur S_i ;
- pour tout i, $1 \leq i \leq B$ et pour tout j, $1 \leq j \leq p+1$, les données $x_{i,j}$ sont stockées sur X_i ;
- pour tout i, $1 \leq i \leq B$, et pour tout j, $1 \leq j \leq p-2$, les données $y_{i,j}$ sont stockées sur Y_i ;
- pour tout k, $1 \leq k \leq p+1$, les données z_k sont répliquées $B+2$ fois pour être stockées sur W_1, sur W_2 et sur chaque X_i avec $1 \leq i \leq B$.

Nous posons finalement la borne de l'ordonnancement : $K = p$. Cela complète la description de $\mathcal{I}_2$ dont la taille est clairement polynomiale en la taille de $\mathcal{I}_1$. Comme cela a été spécifié dans le problème, toutes les données ont une taille unitaire, tous les liens de communication une bande passante unitaire, et les calculs ont une durée nulle.

Nous avons maintenant à montrer que $\mathcal{I}_2$ admet une solution si et seulement si $\mathcal{I}_1$ en admet une. Nous supposons d'abord que $\mathcal{I}_2$ admet une solution. Nous procédons en plusieurs étapes :

1. Le serveur X_i exécute nécessairement la tâche T'_i. Autrement, si un autre serveur l'exécutait, les $p+1$ données $x_{i,j}$ (pour tout j) qui résident dans l'entrepôt de X_i devraient être transférées vers un autre serveur. Comme X_i est la seule source pour ces données, cela demanderait $p+1$ messages depuis X_i, ce qui est impossible en p pas de temps.

2. Suivant le même raisonnement, un serveur étiqueté S ou Y ne peut exécuter aucune des tâches $T_{i,j}$ car ces tâches dépendent des $p+1$ données z_i dont les sources sont les serveurs X, W_1 et W_2.

3. Comme X_i exécute T'_i, il doit recevoir les $p-2$ données $y_{i,j}$ (pour tout j), dont la seule source est le serveur Y_i. X_i ne peut recevoir au plus que deux autres données durant les p étapes de l'ordonnancement. Par conséquent, X_i peut exécuter au plus une tâche $T_{j,k}$ s'il reçoit les deux données manquantes.

4. Combien de données d_i ont été reçues par W_1 et W_2 ? Soit V_1 l'ensemble des données d_i envoyées par les serveurs S à W_1, et V_2 celles envoyées à W_2. Nous ne connaissons pas encore quelles données ont été envoyées aussi bien à W_1 qu'à W_2. Supposons pour le moment qu'une certaine donnée d_i n'a été envoyée ni à W_1, ni à W_2. Toutes les tâches $T_{i,j}$ où $(i,j) \in E$ ont dû être exécutées ailleurs. Les seuls candidats sont les serveurs X. Il y a B serveurs X, et chacun peut exécuter au plus une telle tâche, alors que le degré de v_i dans H est au moins $B+1$, une contradiction.
5. Ainsi, chaque donnée d_i a été envoyée soit à W_1, soit à W_2 (ou aux deux). Il y a un total de $2p$ données à envoyer à ces deux serveurs en p unités de temps, chaque serveur ne peut donc recevoir plus de p données. Ainsi, aucune donnée n'a été envoyée deux fois. Nous en concluons que V_1 et V_2 forment une partition de V avec $|V_1| = |V_2| = p$.
6. Il reste à montrer qu'il n'y a pas plus de B arêtes coupées par la partition $V = V_1 \cup V_2$. Ces arêtes correspondent aux tâches qui peuvent seulement être exécutées par les serveurs X, d'où le résultat.

Nous avons finalement caractérisé une solution pour $\mathcal{I}_1$.

Supposons maintenant que $\mathcal{I}_1$ a une solution, c'est-à-dire qu'il existe une partition $V = V_1 \cup V_2$ avec $|V_1| = |V_2| = p$ telle que

$$b = |\{e = (u,v) \in E \mid u \in V_1, v \in V_2\}| \leq B.$$

Pour tout i, $1 \leq i \leq B$, nous décidons d'allouer la tâches T'_i au serveur X_i. Pour chaque arête $(v_i, v_j) \in E$ telle que v_i et v_j appartiennent tous les deux à V_1, nous exécutons la tâche $T_{i,j}$ sur le serveur W_1. Nous prenons une décision similaire quand v_i et v_j appartiennent tous les deux à V_2 en exécutant la tâche sur W_2. Notons que pour exécuter l'ensemble des tâches allouées à W_1 et à W_2, chaque donnée d_i n'aura à être transférée que sur un seul des deux serveurs. Il reste b tâches à allouer : celles correspondant aux arêtes coupées. Comme $b \leq B$, nous allouons une de ces tâches à chacun des premiers serveurs X_i, $1 \leq i \leq b$. Cette allocation implique en tout $B \cdot (p-2) + 2p + 2b$ communications : $p-2$ pour chaque tâche T'_i ; une pour chaque donnée d_i, soit vers W_1, soit vers W_2 ; et deux pour chaque arête coupée. Nous avons à ordonnancer ces communications en $K = p$ unités de temps sans violer les contraintes un-port. Intuitivement, W_1 reçoit p données d_i où $v_i \in V_1$ et de même pour W_2 : ces $2p$ messages sont indépendants et peuvent être ordonnancés n'importe quand. Chaque X_i reçoit $p-2$ données $y_{i,j}$ de Y_i, il a donc deux pas de temps de disponibles, exactement ce dont il a besoin pour recevoir les deux données nécessaires à l'exécution d'une tâche correspondant à une arête coupée. Chaque serveur S_i envoie sa donnée d_i au plus $B+1$ fois, une

fois vers un des serveurs W, et autant de fois que v_i est un sommet incident à une arête coupée. Plutôt que de décrire l'ordonnancement en extension, ce qui serait laborieux, nous construisons un graphe biparti avec les émetteurs d'un côté (les serveurs S et Y), les récepteurs de l'autre (les serveurs W et X), et nous créons une arête pour chacune des communications à ordonnancer. Comme $B + 1 \leq p$, le degré maximal des nœuds du graphe n'excède pas p. Le théorème de König sur la coloration des arêtes d'un graphe biparti [57] [84, chapitre 20] assure que nous pouvons partitionner l'ensemble des arêtes en p partitions disjointes, et nous ordonnançons une partition (qui, par définition, met en jeu des communications indépendantes) par unité de temps.

Au final, nous obtenons un ordonnancement valide pour exécuter toutes les tâches en p unités de temps, et donc une solution à $\mathcal{I}_2$. □

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 donnée, 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, pour les données ainsi que dans le graphe de plate-forme. Il est alors vraisemblable qu'il n'existe pas d'algorithmes 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.

4.4 Adaptation du *min-min*

Dans la continuité de ce qui a été fait au chapitre précédent, nous commençons par adapter le schéma du *min-min*. Nous introduirons plus tard, en section 4.5, plusieurs nouvelles heuristiques dont la principale caractéristique est une complexité algorithmique moindre que celle du *min-min* en conservant une qualité de résultat comparable. Cette section est organisée comme suit. Nous rappelons le principe de base de l'heuristique *min-min* en section 4.4.1 Dans la section 4.4.2, nous concevons deux algorithmes pour ordonnancer un ensemble de communications dont la destination est un même serveur, une étape cruciale dans l'allocation d'une nouvelle tâche à un serveur. Nous décrivons finalement notre adaptation de l'heuristique *min-min* en section 4.4.3.

4.4.1 Principe du *min-min*

Le principe du *min-min* est plutôt simple :

Tant qu'il reste des tâches à ordonnancer, faire

(i) pour chaque tâche T_k qui reste à ordonnancer, et pour chaque processeur $C_{i,j}$ dans le système, évaluer le temps de complétion minimum (MCT) de T_k si elle est placée sur $C_{i,j}$;

(ii) prendre un couple $(T_k, C_{i,j})$ avec un MCT minimum, et ordonnancer la tâche T_k sur le processeur $C_{i,j}$.

La difficulté avec cette heuristique, est d'évaluer le temps de complétion minimum (MCT). En voulant ordonnancer une tâche sur un processeur donné, il faut tenir compte, d'une part des données nécessaires qui se trouve déjà dans l'entrepôt correspondant, et d'autre part des données à transférer à travers le réseau de serveurs. Si des données à transférer sont déjà répliquées dans les entrepôts de plusieurs serveurs, il faut d'abord décider quelle seront les serveurs qui serviront de source pour les transferts. Un fois que la source pour chaque donnée à transférer a été choisie, il est facile de déterminer quelles communications doivent avoir lieu. L'ordonnancement de ces communications est cependant un problème difficile, comme il en sera question un peu plus loin.

Une fois que l'ordonnancement des communications a été décidé, il reste à évaluer le temps de calcul. Il faudrait pour cela calculer un ordonnancement des tâches au sein des processeurs de la grappe associée au serveur. Nous choisissons plutôt d'utiliser une heuristique et de considérer l'ensemble de la grappe comme un seul processeur dont la puissance de calcul est égale à la somme des puissances de calcul des processeurs de la grappe (l'approche grossière évoquée dans la section 4.2.2). La date du début de l'exécution de la nouvelle tâche est définie comme la date la plus tardive entre

(i) la date d'arrivée de la dernière donnée requise ;

(ii) la date de fin (suivant l'évaluation grossière) du calcul des tâches déjà allouées sur cette grappe.

Avec cette approche simplifiée pour les calculs, le seul problème qu'il nous reste concerne l'ordonnancement des communications.

4.4.2 Ordonnancement des 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 commençons par montrer la difficulté du problème avant de présenter deux algorithmes possibles pour ordonnancer les communications.

Complexité de l'ordonnancement de communications

Au moment d'ordonnancer les communications requises pour une tâche T_k, il faut prendre en compte les communications qui ont déjà été ordonnancées pour les tâches précédentes. Autrement dit, en essayant d'ordonnancer les communications pour une nouvelle tâche, il faut tenir compte du fait que les liens réseau sont déjà occupés à certains moments par des communications précédemment ordonnancées. Nous montrons maintenant que ce problème est NP-complet, même avec notre hypothèse de routage fixé.

Remarque. Dans le cas où le routage est libre, le problème combiné du routage et de l'ordonnancement des communications est déjà NP-complet, même si tous les liens sont entièrement disponibles. Voir la démonstration en annexe D.

Nous définissons formellement le problème de décision associé à notre problème d'ordonnancement des communications comme suit :

Définition 4.3 (CommSchedWithHistory$(\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T)$**).** Étant donnés un graphe de plate-forme $\mathcal{P} = (\mathcal{S} \cup \mathcal{R}, \mathcal{L})$ avec $\mathcal{R} = \emptyset$, un ensemble fini $\mathcal{M}$ de communications ayant la même destination D, un historique $\mathcal{H}$ spécifiant, pour chaque lien de communication, à quels intervalles de temps il est occupé à cause de communications précédemment ordonnancées, et une borne de temps T, existe-t-il un ordonnancement valide des communications dont la durée totale ne dépasse pas T ?

Theorème 4.3. CommSchedWithHistory$(\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T)$ *est NP-complet.*

Démonstration. Il est évident que CommSchedWithHistory$(\mathcal{P}, \mathcal{M}, D, \mathcal{H}, T)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction assez simple de 2-Partition. Considérons une instance arbitraire $\{a_1, a_2, \ldots, a_n\}$ de 2-Partition avec $\sum_{1 \leq i \leq n} a_i = N$. Nous construisons, à partir de cette instance, une plate-forme simplement composée de deux serveurs reliés par un lien de communication qui est disponible à tout instant t dans $[0\ ;\ N/2] \cup [1 + N/2\ ;\ +\infty[$.

Nous posons $T = N + 1$. Nous pouvons alors voir facilement qu'il y a une solution pour l'instance de 2-Partition si et seulement si il y a une solution pour l'instance correspondante de CommSchedWithHistory. Pour aboutir à ce résultat, nous faisons la supposition que les communications ne sont jamais préemptées, c'est-à-dire que l'envoi d'une donnée se fait en une seule fois et n'est pas découpé en plusieurs envois qui seraient effectués à différents instants non contigus. □

À la vue de ces résultats, nous décidons (de manière heuristique) d'utiliser de simples algorithmes gloutons pour ordonnancer les communications.

Algorithme d'ordonnancement des communications

Nous proposons maintenant deux algorithmes pour ordonnancer les communications requises pour transférer les données nécessaires vers le serveur sur lequel on souhaite ordonnancer une tâche. Le premier algorithme ordonnance les nouvelles communications dans le premier intervalle de temps disponible pour le lien de communication, alors que le second, encore plus simple, ordonnance toujours les nouvelles communications *après* toutes celles qui ont déjà été ordonnancées. Dans la suite, nous appelons :

transfert la communication d'une donnée à travers le réseau de serveurs, depuis le serveur source (dont l'entrepôt détient la donnée) vers le serveur destination (où nous voulons ordonnancer la nouvelle tâche) ;

communications locales les communications entre serveurs adjacents ; ainsi, un transfert est composé d'une ou plusieurs communications locales.

Une première idée est de mémoriser pour chacun des liens la date et la durée de toutes les communications déjà ordonnancées, puis d'utiliser un *ordonnancement par insertion* pour ordonnancer les nouvelles communications dans le premier intervalle de temps disponible (cette stratégie est aussi appelée *first-fit*). Ce schéma devrait être assez précis mais peut devenir très coûteux. Une seconde idée est d'ordonnancer les nouvelles communications le plus tôt possible mais toujours *après* toutes les communications utilisant déjà le même lien. Dans les deux cas, si une donnée est présente sur plusieurs serveurs, nous décidons (de manière heuristique) d'amener la donnée depuis le serveur le plus proche, le serveur le plus proche étant celui depuis lequel la donnée serait amenée le plus rapidement si le réseau était entièrement disponible.

Les deux algorithmes sont résumés par l'algorithme 4.1 : les communications locales sont ordonnancées sur chaque serveur intermédiaire, ces serveurs intermédiaires étant considérés suivant les valeurs décroissantes de leur distance à la destination des transferts. Les différences entre les deux algorithmes se trouvent dans le choix de la communication locale (ligne 10) et dans son ordonnancement (ligne 11). Dans la variante avec insertion, les communications locales sont considérées par durées décroissantes (ligne 10), puis ordonnancées dans le premier intervalle de temps possible (ligne 11). Une liste des intervalles de temps disponibles doit donc être maintenue. Dans la variante plus simple, les communications locales sont considérées par dates de disponibilité croissantes (ligne 10), c'est-à-dire le maximum entre la date d'arrivée de la donnée D_i sur le serveur émetteur R, la date de fin de la dernière émission de R et la date de fin de la dernière réception de S. Une nouvelle communication de R à S est toujours ordonnancée après la dernière communication ayant R comme émetteur et S comme récepteur (ligne 11).

Bien sûr, ces algorithmes ne calculent pas toujours une solution optimale. Il est vraisemblable que le problème général est NP-complet (même sans les contraintes

Algorithme 4.1 – Ordonnancement d'un ensemble de transferts de même destination Ω.

```
   ▷ calcul des communications locales
 1 pour chaque transfert τ de destination Ω faire
 2 |   construire la liste L des communications locales composant τ
 3 |   pour chaque communication locale c de L faire
 4 |   |   calculer la distance de c par rapport à Ω
 5 |   fin
 6 fin
 7 calculer H, le maximum des distances des communications locales
   ▷ ordonnancement des communications locales
 8 pour h ← H à 0 par pas de −1 faire
 9 |   tant que il reste des communications locales de distance h à ordonnancer
   |   faire
10 |   |   choisir une communication locale c de distance h
   |   |   ▷ cette communication locale consiste en l'envoi d'une donnée D_i depuis
   |   |     un serveur R vers un serveur S
11 |   |   ordonnancer c, en prenant en compte la date d'arrivée de D_i sur R
12 |   fin
13 fin
```

de disponibilité pour les liens) à cause de sa ressemblance avec le problème d'ordonnancement de *flow-shop* [39] d'une part, et le problème de l'ordonnancement d'un ensemble de transferts de fichiers d'autre part. Ce dernier problème a été étudié dans le cas où les connexions sont directes, d'abord par Coffman, Garey, Johnson et LaPaugh [30], puis par d'autres [27, 70]. Le cas où les transferts peuvent passer par des serveurs intermédiaires suivant un modèle *store-and-forward* a également été étudié [93, 80, 71]. Le problème de l'ordonnancement d'un ensemble de transferts de fichiers a été montré NP-complet dans le cas général ainsi que dans un certain nombre de cas particuliers. Les hypothèses de départ (contraintes de port, nombre de destinations, nombre maximal de sauts dans le cas *store-and-forward*, etc.) diffèrent cependant légèrement des nôtres et les résultats ne peuvent donc pas être directement transposés à notre problème.

Voici un exemple simple où aucun des deux algorithmes présentés ne donne la solution optimale. Considérons deux données D_1 et D_2 de tailles $d_1 = 2$ et $d_2 = 4$, à transférer d'un serveur A à un serveur C via un serveur B. La bande passante du lien entre A et B est 1, celle du lien entre B et C est 0,5. La date de disponibilité de D_1 sur le serveur A est $t = 1$ alors que celle de D_2 est $t = 0$. Il n'y a aucune communication précédente. Les deux algorithmes vont transférer D_2 avant D_1 sur chacun des liens à partir de $t = 0$: l'envoi de D_2 se termine à $t = 4$ sur B et à $t = 12$ sur C ; D_1 suit à $t = 6$ sur B et à $t = 16$ sur C. L'envoi de D_1 en premier

donne cependant un meilleur résultat, même si nous ne pouvons pas commencer avant $t = 1$: D_1 arrive sur B à $t = 3$ et sur C à $t = 7$, alors que D_2 arrive sur B à $t = 7$ et sur C à $t = 15$.

Complexité

Nous notons par :
- ΔT le degré maximum d'une tâche dans le graphe d'application, c'est-à-dire le nombre maximum de données dont une tâche dépend ;
- $\Delta \mathcal{P}$ le diamètre du graphe de plate-forme, c'est-à-dire la plus grande distance entre deux serveurs.

Comme l'ensemble des transferts de même destination correspond au transfert de l'ensemble des données dont dépend une tâche, il y a au plus ΔT transferts à ordonnancer et chaque transfert est composé de au plus $\Delta \mathcal{P}$ communications locales.

Avec l'*ordonnancement par insertion*, la ligne 11 implique le parcours d'au plus m trous pour chacun des lien de communication concernés. Le transfert d'une seule donnée implique au plus $\Delta \mathcal{P}$ liens de communication. Ainsi, la complexité globale de la ligne 11 pour le transfert d'une seule donnée est de $O(m \cdot \Delta \mathcal{P})$. Comme une tâche dépend au plus de ΔT données, et qu'il faut trier les communications locales correspondantes pour chacune des (au plus) $\Delta \mathcal{P}$ distances, le coût global de la ligne 10 est de $O(\Delta \mathcal{P} \cdot \Delta T \cdot \log \Delta T)$. La complexité de l'algorithme d'ordonnancement par insertion est donc :

$$O\Big(\underbrace{\Delta \mathcal{P} \cdot \Delta T}_{\text{lignes 1 à 7}} + \underbrace{\Delta \mathcal{P} \cdot \Delta T \cdot \log \Delta T}_{\text{ligne 10}} + \underbrace{\Delta \mathcal{P} \cdot m \cdot \Delta T}_{\text{ligne 11}}\Big)$$

qui se simplifie en :

$$O\left(\Delta T \cdot m \cdot \Delta \mathcal{P}\right)$$

car, par définition, $\log \Delta T \leq \Delta T \leq m$.

Sans ordonnancement par insertion, la ligne 11 calcule le maximum des dates de disponibilités des liens de communication concernés. Ainsi, la complexité globale de la ligne 11 pour le transfert d'une seule donnée est de $O(\Delta \mathcal{P})$. La complexité de l'algorithme devient alors :

$$O\Big(\underbrace{\Delta \mathcal{P} \cdot \Delta T}_{\text{lignes 1 à 7}} + \underbrace{\Delta \mathcal{P} \cdot \Delta T \cdot \log \Delta T}_{\text{ligne 10}} + \underbrace{\Delta \mathcal{P} \cdot \Delta T}_{\text{ligne 11}}\Big).$$

qui se simplifie en :

$$O\left(\Delta \mathcal{P} \cdot \Delta T \cdot \log \Delta T\right).$$

Dans le reste du chapitre, nous noterons par O_c la complexité de l'algorithme d'ordonnancement des communications. Alors, O_c sera égal à l'une ou l'autre des deux formules ci-dessus, selon la version de l'algorithme choisie.

4.4.3 Le *min-min* adapté

L'algorithme 4.2 présente notre implémentation de l'heuristique *min-min* pour des serveurs distribués. À chaque fois qu'une donnée est requise, elle est transférée depuis le serveur le plus proche parmi ceux qui en détiennent une copie. Cette politique demande de maintenir dynamiquement une liste des sources possibles pour chaque donnée. Un autre politique pourrait être d'utiliser l'algorithme 4.1 sur tous les ensembles de sources possibles (soit une source par donnée dans chaque ensemble), et de choisir la meilleure solution. Connaissant la complexité de l'heuristique *min-min*, une politique aussi coûteuse ne serait pas réaliste.

Avec $p = \max_{1 \leq i \leq s} p_i$, la complexité globale de l'heuristique *min-min* est :

$$O\Big(\underbrace{s \cdot n \cdot (n \cdot O_c + s \cdot |\mathcal{E}|)}_{\text{ligne 4}} + \underbrace{|\mathcal{E}| + n \cdot (\log n + p)}_{\text{lignes 12 à 14}}\Big)$$

qui se simplifie en :

$$O\left(s \cdot n \cdot (n \cdot O_c + s \cdot |\mathcal{E}|) + n \cdot p\right).$$

La partie coûteuse de la première boucle de l'algorithme est la ligne 4 qui correspond à l'évaluation du temps de communication des données requises par une tâche sur un processeur donné. À chaque fois qu'une donnée est requise, le serveur le plus proche parmi ceux détenant une copie de la donnée est recherché. Pour l'ensemble des tâches de chaque processeur, cette recherche coûte $O(s \cdot |\mathcal{E}|)$, car il y a au plus $|\mathcal{E}|$ relations de dépendance entre les tâches et les données. Il faut y ajouter le coût de l'ordonnancement des communications (O_c) multiplié par le nombre de tâches concernées (n). En sommant pour toutes les itérations de la boucle, on trouve la complexité totale de cette partie de l'algorithme qui est : $O(s \cdot n \cdot (n \cdot O_c + s \cdot |\mathcal{E}|))$.

Nous remarquons qu'en passant d'un système avec un seul entrepôt, comme dans le chapitre 3, 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. Cela est dû à notre vue simplifiée du problème : dans la phase de décision, nous voyons chaque grappe C_j comme une seule ressource de calcul de vitesse cumulée $\sum_{k=i}^{p_j} s_{j,k}$.

Le dernier terme de la formule de complexité provient de l'ordonnancement des tâches sur les grappes. Une fois que les communications sont ordonnancées, nous avons, pour chaque tâche, la date de disponibilité des données dont elle dépend,

Algorithme 4.2 – Le *min-min* adapté.

```
1  tant que il reste des tâches à ordonnancer faire
2  |  pour chaque tâche restante T_i faire
3  |  |  pour chaque serveur S_j faire
4  |  |  |  utiliser l'algorithme 4.1 pour calculer la date t à laquelle toutes les
   |  |  |  données nécessaires à l'exécution de T_i seront disponibles sur S_j
5  |  |  |  évaluer, en fonction de t, le temps de complétion minimum de T_i sur
   |  |  |  S_j en considérant la grappe C_j comme un seul processeur
6  |  |  fin
7  |  fin
8  |  choisir un couple (T_i, S_j) dont le temps de complétion minimum est minimal
9  |  placer T_i sur S_j
10 |  utiliser l'algorithme 4.1 pour ordonnancer les communications nécessaires à
   |  l'exécution de T_i sur S_j
11 fin
12 pour chaque serveur S_j faire
13 |  ordonnancer de manière gloutonne les tâches placées sur S_j
14 fin
```

avec un coût global de $O(|\mathcal{E}|)$ pour l'ensemble des tâches. Sur chaque grappe, nous ordonnançons les tâches sur les processeurs de manière gloutonne. Les tâches sont triées par valeurs croissantes des dates de disponibilité de leurs données, avec, dans le pire cas, un coût global de $O(n \cdot \log n)$. Ensuite, dès qu'un processeur est libre, nous choisissons la tâche la plus grande parmi celles dont les données sont disponibles. Nous ordonnançons cette tâche sur le processeur qui terminera son exécution le plus tôt (en prenant en compte la date à laquelle le processeur est disponible, sachant quelles tâches lui sont déjà attribuées). La complexité globale de cet ordonnancement de tâches est $O(n \cdot p)$ si nous notons $p = \max_{1 \leq i \leq s} p_i$.

Variante de l'heuristique *min-min* : l'heuristique *sufferage*

L'heuristique *sufferage* est une variante du *min-min* qui produit parfois de meilleurs ordonnancements, mais dont la complexité est légèrement supérieure (cf. chapitre 3). La différence se trouve à l'étape 8 où, au lieu de choisir un couple (T_i, S_j) avec un MCT minimal, la tâche choisie T_i est celle qui serait la plus pénalisée si elle n'était pas placée sur son processeur le plus favorable, mais sur son deuxième plus favorable, c'est-à-dire la tâche avec la plus grande différence entre ses deux plus petits MCT. La tâche est alors placée sur un serveur S_j qui assure un MCT minimal pour la tâche.

4.5 Heuristiques moins coûteuses

Comme nous l'avons déjà vu au chapitre précédent, 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 plus rapides d'un ordre de grandeur, 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 dans le chapitre 3, 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, faire

(i) pour chaque grappe C_i, choisir la « meilleure » tâche candidate T_k qui reste à ordonnancer ;

(ii) 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* que nous utiliserons pour estimer le temps de complétion minimum d'une tâche sur un serveur donné. Suivant les résultats de notre étude dans le cas où il n'y a qu'un seul entrepôt (cf. section 3.5), notre fonction de *coût* va représenter la durée totale du transfert des données puis de l'exécution des tâches sur les serveurs. Nous avons conçu deux types d'heuristiques : des heuristiques statiques où les *coûts* sont estimés une fois pour toute, et des heuristiques dynamiques où les *coûts* sont réévalués au fur et à mesure que les décisions de placement et d'ordonnancement sont prises.

4.5.1 Heuristiques statiques

Dans nos heuristiques statiques, nous commençons par construire, pour chaque grappe, une liste de toutes les tâches triées suivant les valeurs croissantes d'une *fonction de coût*. Ensuite, chaque fois que nous ordonnançons une nouvelle tâche,

(i) nous définissons comme candidate locale pour la grappe C_i la tâche qui a le *coût* le plus faible et qui n'a pas encore été ordonnancée ;

(ii) parmi toutes les candidates locales, nous prenons celle de plus bas *coût* et nous la plaçons sur la grappe correspondante ;

(iii) nous ordonnançons les communications nécessaires ainsi que le calcul comme nous l'avons fait pour l'heuristique *min-min*.

L'algorithme 4.3 présente la structure de ces heuristiques statiques. Nous allons décrire les différentes étapes de l'algorithme et expliquer nos choix de conception.

Algorithme 4.3 – Structure des heuristiques statiques.

```
pour chaque tâche T_i faire
    pour chaque serveur S_j faire
        calculer le coût a(t_i, S_j)
    fin
    retenir S(T_i) tel que a(T_i, S(T_i)) = min_{1≤j≤s} a(T_i, S_j)
fin
trier les tâches par ordre de coût a(T_i, S(T_i)) croissant
tant que il reste des tâches à ordonnancer faire
    choisir la prochaine tâche non ordonnancée T_i
    placer T_i sur la grappe de S(T_i)
    utiliser l'algorithme 4.1 pour ordonnancer les communications requises par
    l'exécution de T_i sur S(T_i)
fin
pour chaque serveur S_j faire
    ordonnancer de manière gloutonne les tâches placées sur S_j
fin
```

La fonction de coût

Nous définissons le *coût* d'une tâche T_i sur un serveur S_j comme une évaluation du temps de complétion minimum de T_i sur S_j. Poursuivant ce qui a été fait pour l'heuristique *min-min*, le temps de complétion minimum est défini comme la somme du temps nécessaire pour envoyer les données requises vers S_j (avec l'ordonnancement calculé par l'algorithme 4.1) plus le temps nécessaire à la grappe C_j pour traiter la tâche, quand la grappe est vue comme un ressource de calcul simple.

En réalité, nous accélérons le calcul des *coûts* en approximant le temps de communication. Au lieu d'utiliser l'algorithme 4.1 qui est précis mais coûteux, nous utilisons la somme des temps de transfert (sur-approximation pessimiste par sérialisation de tous les transferts) ou le maximum de ces temps de transfert (sous-approximation optimiste en considérant que tous les transferts peuvent avoir lieu en parallèle). Si nous pré-calculons une fois pour toute le coût de l'envoi d'une donnée élémentaire entre chaque paire de serveurs, ce qui nécessite $O(s^2 \cdot \Delta\mathcal{P})$ opérations, l'évaluation du coût du transfert d'une donnée se fait en temps constant. Pour un serveur donné, la complexité de l'évaluation des coûts de l'ensemble des tâches est donc de $O(n + |\mathcal{E}|)$ et la complexité globale de l'évaluation des *coûts* pour tous les couples de tâche et de serveur est de $O(s \cdot (n + |\mathcal{E}|) + s^2 \cdot \Delta\mathcal{P})$.

Dans un modèle avec des latences pour les communications, sans utiliser de pré-calcul, la complexité de l'évaluation du coût du transfert d'une donnée est de $O(\Delta\mathcal{P})$. L'évaluation de l'ensemble des *coûts* nécessiterait donc $O(s \cdot (n + |\mathcal{E}| \cdot \Delta\mathcal{P}))$

opérations.

Pour chaque tâche, son meilleur serveur est finalement retenu (ligne 5) et les coûts des tâches sur leurs meilleurs serveurs respectifs sont triés (ligne 7) avec un coût de $O(n \cdot \log n)$.

Ordonnancement des tâches

Une fois la décision d'allocation prise, pour ordonnancer les communications, nous utilisons l'algorithme 4.1, la source de chaque donnée étant le serveur le plus proche, comme pour l'heuristique *min-min*. Le choix des sources des données plus l'ordonnancement des communications requises a donc, comme pour l'heuristique *min-min*, une complexité totale de $O(s \cdot |\mathcal{E}| + n \cdot O_c)$. Comme précédemment, une fois que les communications sont ordonnancées, nous avons, pour chaque tâche, la date de disponibilité des données dont elle dépend, avec un coût global de $O(|\mathcal{E}|)$ pour l'ensemble des tâche. Pour ordonnancer les tâches, nous procédons comme expliqué dans la section 4.4.3 pour l'heuristique *min-min*, avec un coût de $O(n \cdot \log n + n \cdot \max_{1 \leq i \leq s} p_i)$.

Complexité globale

Dans le reste du chapitre, nous notons l'heuristique présentée ci-dessus par *static* lorsque le coût des communications est surévalué (somme des temps de transfert) et par *static+max* lorsque le coût des communications est sous-évalué (maximum des temps de transfert). Dans les deux cas, la complexité de l'heuristique est d'un ordre de grandeur moindre que celle de l'heuristique *min-min* car nous n'avons plus le terme n^2 :

$$O\Big(\underbrace{s \cdot (n + |\mathcal{E}|) + s^2 \cdot \Delta\mathcal{P}}_{\text{lignes 1 à 6}} + \underbrace{n \cdot \log n}_{\text{ligne 7}} + \underbrace{s \cdot |\mathcal{E}| + n \cdot O_c}_{\text{lignes 9 à 11}} + \underbrace{|\mathcal{E}| + n \cdot (\log n + p)}_{\text{lignes 13 à 15}}\Big)$$

qui se simplifie en :

$$O\left(n \cdot (\log n + O_c + p) + s^2 \cdot \Delta\mathcal{P} + s \cdot |\mathcal{E}|\right)$$

car s est inférieur à $|\mathcal{P}|$ qui apparaît dans O_c (quelle que soit la variante).

4.5.2 Variantes des heuristiques statiques

À partir de cette première heuristique, nous concevons plusieurs variantes : *critic* qui reconsidère, à la fin de l'heuristique, l'ordre des communications en utilisant une approche par chemin critique ; *readiness* qui commence par regarder si une tâche peut être ordonnancée sur un serveur avec un coût de communication nul ; et *mct* qui sélectionne la « meilleure » candidate parmi les candidates locales en utilisant leurs temps de complétion minimum.

Ordonnancement par chemin critique : variante *critic*

Au lieu d'ordonnancer les communications tâche après tâche, nous ordonnançons de manière globale toutes les communications nécessaires. Nous espérons ainsi être capable de donner une priorité plus grande aux communications les plus importantes.

Considérons le transfert d'une donnée D d'un serveur S_i vers un serveur S_k, passant par un seul serveur intermédiaire S_j. Le transfert est donc composé de deux communications locales, la deuxième ($S_j \xrightarrow{D} S_k$) dépendant de la première ($S_i \xrightarrow{D} S_j$). Pour ordonnancer l'ensemble des communications, nous construisons le graphe de dépendance des communications locales où chaque sommet représente une communication locale et où il y a une arête d'un sommet à un autre si et seulement si le premier représente une communication locale devant être réalisée avant que celle représentée par le second ne puisse avoir lieu. Dans notre exemple, il y aurait une arête du sommet représentant $S_i \xrightarrow{D} S_j$ vers le sommet représentant $S_j \xrightarrow{D} S_k$. Notre graphe de dépendance est évidemment un graphe orienté acyclique (DAG). Il contient au plus $O(m \cdot s)$ sommets (au pire, chaque donnée est répliquée sur chaque serveur à la fin).

Nous associons un *poids* à chaque sommet $S_i \xrightarrow{D} S_j$ du graphe de dépendance. Ce poids est égal à la somme des temps de calcul des tâches placées sur la grappe C_j et dépendant de la donnée D, plus la durée de la communication elle-même. Nous définissons le *coût* d'un sommet comme la somme de son poids et des poids de tous les sommets atteignables depuis ce sommet. Finalement, en utilisant une approche par chemin critique, nous ordonnançons les communications (parmi celles ayant leur dépendances satisfaites) par ordre décroissant des *coûts* des sommets. En pratique, nous maintenons les communications prêtes à être ordonnancées (celles dont toutes les dépendances sont satisfaites) triées par *coût* dans une structure de donnée en *tas* [31, chapitre 7]. Cela nous permet de prendre la prochaine communication à ordonnancer à coût constant. Après l'ordonnancement d'une communication, les communications devenues prêtes sont insérées dans le tas.

La complexité de cet ordonnancement des communications est $O(m{\cdot}s{\cdot}\log(m{\cdot}s))$ sans ordonnancement par insertion (notons que l'ensemble des coûts est calculé par un simple parcours de graphe en $O(m \cdot s)$). Avec ordonnancement par insertion, la complexité s'accroît par un facteur additionnel de $O(m^2 \cdot s)$ dû au coût des insertions pour chacune des communications locales.

Priorité aux tâches prêtes : variante *readiness*

Dans la version de base de l'heuristique, les tâches sont sélectionnées en suivant strictement l'ordre croissant de leur *coûts* (ligne 9 de l'algorithme 4.3). Avec la

variante *readiness*, chaque fois que nous essayons d'ordonnancer une nouvelle tâche, nous considérons la prochaine tâche dans cet ordre *sauf* s'il y a une tâche T_i qui est prête pour un serveur S_j auquel cas nous ordonnançons immédiatement T_i sur S_j. Une tâche est appelée *prête* pour un serveur si l'entrepôt correspondant détient toutes les données dont la tâche dépend : une tâche prête est une tâche qui peut être ordonnancée avec un coût de communication nul. La variante *readiness* est une adaptation directe de la variante de même nom qui avait grandement amélioré la qualité des heuristiques du chapitre précédent, dans la version du problème avec un seul serveur.

Nous maintenons, pour chaque serveur et pour chaque tâche, le nombre de données manquant à la tâche sur le serveur. Chaque fois qu'une donnée arrive sur un serveur, nous décrémentons le nombre de données manquantes pour toutes les tâches qui en dépendent. La maintenance des listes de tâches prêtes coûte donc $O(s \cdot |\mathcal{E}|)$.

Le placement des tâches sur les serveurs est différé : variante *mct*

Dans l'heuristique *static*, une tâche est placée sur un serveur où son *coût* est minimal. Il n'y a donc, avec les heuristiques *static* et *static+readiness*, aucun équilibrage de la charge des processeurs. Pour remédier à ce problème, et suivant les idées du chapitre 3, nous n'utilisons le *coût* que pour déterminer l'ordre dans lequel les tâches sont considérées. Pour cela, nous trions, pour chaque serveur, les tâches par *coûts* croissants. Une fois que les listes triées ont été calculées, il nous reste à placer les tâches sur les serveurs et à les ordonnancer. Les tâches sont toujours ordonnancées les unes après les autres. Quand nous voulons ordonnancer une nouvelle tâche, nous évaluons, pour chaque serveur S_i, le temps de fin de la première tâche de sa liste qui n'a pas encore été ordonnancée. L'évaluation du temps de fin est identique à l'évaluation équivalente effectuée par l'heuristique *min-min*. Nous prenons ensuite le couple de tâche et de serveur avec le plus petit temps de fin. De cette manière, nous obtenons notre heuristique *static+mct*, présentée par l'algorithme 4.4, qui inclut par défaut la variante *readiness* (lignes 9 à 13 de l'algorithme).

La complexité globale de l'heuristique *static+mct* est :

$$O\Big(\underbrace{s \cdot (n + |\mathcal{E}|) + s^2 \cdot \Delta\mathcal{P}}_{\text{lignes 1 à 4}} + \underbrace{s \cdot n \cdot \log n}_{\text{lignes 5}} + \underbrace{s \cdot |\mathcal{E}|}_{\text{lignes 9 à 13}} + \underbrace{s^2 \cdot |\mathcal{E}| + n \cdot s \cdot O_c}_{\text{lignes 14 et 19}} + \underbrace{|\mathcal{E}| + n \cdot (\log n + p)}_{\text{lignes 21 à 23}}\Big)$$

qui se simplifie en :

$$O\left(s \cdot n \cdot (\log n + O_c) + s^2 \cdot (|\mathcal{E}| + \Delta\mathcal{P}) + n \cdot p\right).$$

Algorithme 4.4 – Structure de l'heuristique *static+mct*.

```
pour chaque serveur S_j faire
    pour chaque tâche T_i faire
        calculer le coût a(T_i, S_j)
    fin
    construire la liste L(S_j) des tâches triées par valeurs croissantes de a(T_i, S_j)
fin
tant que il reste des tâches à ordonnancer faire
    pour chaque serveur S_j faire
        si il y a des tâches prêtes pour S_j alors
            T_i ← n'importe quelle tâche prête pour S_j
        sinon
            T_i ← la première tâche non ordonnancée de L(S_j)
        fin
        utiliser l'algorithme 4.1 pour calculer la date t à laquelle toutes les
        données nécessaires à l'exécution de T_i seront disponibles sur S_j
        évaluer, en fonction de t, le temps de complétion minimum de T_i sur S_j
        en considérant la grappe C_j comme un seul processeur
    fin
    choisir un couple (T_i, S_j) dont le temps de complétion minimum est minimal
    placer T_i sur S_j
    utiliser l'algorithme 4.1 pour ordonnancer les communications nécessaires à
    l'exécution de T_i sur S_j
fin
pour chaque serveur S_j faire
    ordonnancer de manière gloutonne les tâches placées sur S_j
fin
```

Par rapport à l'heuristique *static*, la complexité de l'heuristique s'est (grossièrement) accrue d'un facteur s, correspondant à l'évaluation du temps de fin de chaque tâche sur tous les serveurs.

L'effet attendu du raffinement *mct* est de mieux équilibrer la charge sur les serveurs, ce qui est particulièrement important dans les situations où la version de base aurait placé toutes les tâches sur le même serveur. Dans certaines situations extrêmes (par exemple avec de faibles coûts de communication et un serveur nettement plus rapide que les autres), la solution optimale pourrait bien être d'exécuter toutes les tâches sur le même serveur. En pratique, dans la plupart des situations, l'exécution des tâches va être distribuée parmi les serveurs et la variante *mct* se charge d'équilibrer la charge des serveurs au fur et à mesure que les décisions d'allocation des tâches sont prises. On peut remarquer que l'heuristique *static+mct* correspond directement à l'heuristique *duration+readiness* qui était

parmi les meilleures heuristiques du chapitre précédent.

4.5.3 Heuristiques dynamiques

Dans nos heuristiques statiques, nous commençons par définir l'ordre dans lequel les tâches sont considérées et toutes les décisions d'ordonnancement (choix des communications et ordonnancement) sont induite 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 de nouvelles sources pour ces données qui sont créées durant l'exécution. Pour remédier à ce défaut, nous introduisons un schéma plus dynamique, tout en essayant de conserver la faible complexité des heuristiques. La structure de nos heuristiques dynamiques est décrite par l'algorithme 4.5.

Un coût dynamique

Nous voulons 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 :

(i) des données dont T dépend qui ont été répliquées, et du lieu de leur réplication ;

(ii) de la quantité de tâches déjà placées sur S.

La première de ces connaissances permet de choisir, pour les transferts de données, les sources qui sont les plus proches du serveur S, et donc de réduire l'estimation du temps de communication nécessaire. Le problème principal est l'implémentation d'une telle fonction pour une complexité que l'on veut la plus petite possible. Nous voulons précalculer cette fonction de *coût* au maximum, et non pas simplement appeler la fonction de *coût* originale à chaque fois que nous avons besoin de connaître les coûts.

Pour atteindre ce but, chaque serveur détient une table de tous les autres serveurs triés par distance croissante (en temps de communication). La construction d'une telle liste coûte le calcul des distances entre toutes les paires de serveurs, plus le tri des s tables, c'est-à-dire $O(s^2 \cdot \Delta\mathcal{P} + s^2 \cdot \log s)$. À chaque fois que le transfert d'une donnée D est décidé, nous recalculons, pour chaque serveur S_i, la source la *plus proche* pour la donnée D. Ce calcul est fait en temps constant en vérifiant, dans la table définie ci-dessus, si le nouveau serveur détenant D a un rang plus petit que le précédent serveur le *plus proche* pour cette donnée. Comme,

Algorithme 4.5 – Structure des heuristiques dynamiques.

```
pour chaque tâche T_i faire
    pour chaque serveur S_j faire
        calculer le coût a(T_i, S_j)
    fin
fin
tant que il reste des tâches à ordonnancer faire
    pour chaque serveur S_i faire
        prendre une (des) tâche(s) de plus petit(s) coût(s) dynamique(s) sur S_i
    fin
    parmi les tâches prises à l'étape 7, sélectionner le(s) couple(s) (T_i, S_j) de plus
    petit(s) coût(s) dynamique(s), en tenant compte de la quantité de travail déjà
    attribuée à la grappe C_j
    pour chaque couple (T_i, S_j) sélectionné et dans n'importe quel ordre faire
        placer T_i sur le serveur S_j
        utiliser l'algorithme 4.1 pour ordonnancer les communications nécessaires
        à l'exécution de T_i sur S_j
        réévaluer les coûts dont les valeurs ont changé
    fin
fin
pour chaque serveur S_j faire
    ordonnancer de manière gloutonne les tâches placées sur S_j
fin
```

dans le pire cas, chaque donnée est envoyée à chacun des serveurs, il y a au plus, pour l'ensemble des serveurs, $O(s^2 \cdot m)$ de ces mises à jour en temps constant.

Chaque fois que la source la *plus proche* pour une donnée D_k et un serveur S_j change, nous recalculons la valeur du *coût* de la tâche T_i sur le serveur S_j si et seulement si la tâche T_i dépend de la donnée D_k, c'est-à-dire que $e_{k,i} \in \mathcal{E}$. Chacune de ces mises à jour s'effectue en temps constant si nous sur-approximons le temps de communication (somme des temps de transfert, à laquelle nous retranchons la diminution du temps de transfert qui vient d'être modifié), et coûte $O(\Delta T)$ si nous le sous-approximons (maximum des temps de transfert). Pour un serveur donné, la valeur de la source la *plus proche* peut changer au plus s fois pour chacune des données. Il y a donc, globalement, $O(s.|\mathcal{E}|)$ *coûts* de tâche à mettre à jour pour un serveur donné. La complexité globale pour maintenir nos *coûts* dynamiques est ainsi de :

$$O(s^2 \cdot (\Delta\mathcal{P} + \log s) + s^2 \cdot |\mathcal{E}| \cdot u)$$

où $u = 1$ avec sur-approximation et $u = \Delta T$ sinon.

Dans un modèle avec des latences pour les communications, la source la plus proche dépend de la taille de la donnée considérée. Dans une première solution, chaque serveur pourrait détenir une table de tous les autres serveurs triés par distance croissante (en temps de communication) pour n'importe quelle taille de donnée, ce qui est très coûteux et accroîtrait la complexité de la construction des tables par un facteur multiplicatif de m (la complexité des mises à jour ne change pas). Une deuxième solution serait de déterminer le serveur le plus proche en utilisant une taille de donnée moyenne : la complexité ne changerait pas, mais on n'aurait alors qu'une approximation de la source la plus proche. Entre les deux, on peut imaginer de manipuler des classes de tailles de données, ce qui accroîtrait la complexité de la construction des tables par un facteur égal au nombre de classes.

Choix de la (des) prochaine(s) tâche(s) à ordonnancer

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 :

- prendre une seule tâche à la fois et mettre les *coûts* à jour à chaque fois ;
- 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. Comme pour l'heuristique *static*, nous noterons par *max* lorsque le coût des communications est sous-évalué.

Heuristique *dynamic1*

Un moyen simple de prendre une seule tâche serait de chercher, sur chaque serveur, quelle tâche a la plus petit coût et ensuite de rechercher le minimum parmi les serveurs. C'est exactement ce que fait l'heuristique *min-min*. Un tel schéma de sélection coûte $O(n^2 \cdot s)$ ce qui peut être prohibitif. Afin d'accélérer la recherche de la tâche de moindre coût, nous maintenons sur chaque serveur un *tas* des *coûts* des tâches. Ensuite, sur chaque serveur, la sélection de la tâche de plus petit *coût* est faite en temps constant et sa suppression (lorsque la tâche a finalement été ordonnancée) se fait avec un coût de $O(\log n)$. Chaque fois qu'un *coût* est mis à jour, la maintenance des structures de tas a un un coût additionnel de $O(\log n)$ pour la suppression d'un élément du tas et l'insertion d'un nouvel élément [1]. Le *coût* d'une tâche est mis à jour à chaque fois qu'une donnée dont la tâche dépend

1. La suppression d'un élément quelconque dans un tas coûte $O(\log n)$ si on maintient une table associant chaque élément stocké dans le tas avec sa position dans le tas. La maintenance d'une telle table n'augmente pas la complexité théorique des opérations sur le tas.

est répliqué sur un des serveurs, c'est-à-dire au plus $s \cdot |\mathcal{E}|$ fois pour l'ensemble des tâches. La complexité générale de l'heuristique *dynamic1* est donc de :

$$O\Big(\underbrace{s \cdot (n + |\mathcal{E}| + s \cdot \Delta\mathcal{P} + n \cdot \log n)}_{\text{lignes 1 à 5}} + \underbrace{n \cdot s}_{\text{ligne 8}} + \underbrace{s \cdot |\mathcal{E}| + n \cdot O_c}_{\text{ligne 13}} + \underbrace{s \cdot (s \cdot (\Delta\mathcal{P} + \log s) + s \cdot |\mathcal{E}| \cdot u + s \cdot |\mathcal{E}| \cdot \log n)}_{\text{ligne 14}} + \underbrace{|\mathcal{E}| + n \cdot (\log n + p)}_{\text{lignes 17 et 19}}\Big).$$

qui se simplifie en :

$$O\left(n \cdot (\log n + O_c + p) + s^2 \cdot (|\mathcal{E}| \cdot (u + \log n) + \Delta\mathcal{P} + \log s)\right).$$

Par rapport à l'heuristique *static*, la mise à jour des *coûts* des tâches accroît donc la complexité d'un facteur additionnel $s^2 \cdot (|\mathcal{E}| \cdot (u + \log n) + \log s)$.

Heuristique *dynamic2*

Un autre moyen de réduire la complexité de la sélection des « meilleures » tâches candidates est de sélectionner, sur chaque serveur, k ($1 \leq k \leq n$) tâches de plus petit *coûts* au lieu d'une seule. Une telle sélection peut être réalisée en temps linéaire dans le pire cas [20] [31, chapitre 10]. Nous sélectionnons ainsi les k tâches de plus petits coûts sur chaque serveur. Comme une tâche peut apparaître dans la sélection de plusieurs serveurs, nous trions ces $k \cdot s$ couples de tâche et de serveur en fonctions de leurs *coûts* et nous prenons les k tâches distinctes de moindre coût. Le choix de k nous permet ainsi d'obtenir une heuristique intermédiaire entre les heuristiques *static* ($k = n$) et *dynamic1* ($k = 1$). La complexité globale de l'heuristique *dynamic2* est donc :

$$O\Big(\underbrace{s \cdot (n + |\mathcal{E}| + s \cdot \Delta\mathcal{P})}_{\text{lignes 1 à 5}} + \underbrace{\frac{n}{k} \cdot n \cdot s + \frac{n}{k} \cdot k \cdot s \cdot \log(k \cdot s)}_{\text{ligne 8}} + \underbrace{s \cdot |\mathcal{E}| + n \cdot O_c}_{\text{ligne 13}} + \underbrace{s \cdot (s \cdot (\Delta\mathcal{P} + \log s) + s \cdot |\mathcal{E}| \cdot u)}_{\text{ligne 14}} + \underbrace{|\mathcal{E}| + n \cdot (\log n + p)}_{\text{lignes 17 et 19}}\Big).$$

qui se simplifie en :

$$O\left(n \cdot \left(s \cdot \left(\frac{n}{k} + \log(k \cdot s)\right) + O_c + p\right) + s^2 \cdot (|\mathcal{E}| \cdot u + \Delta\mathcal{P} + \log s)\right).$$

En passant de *dynamic1* à *dynamic2* nous avons remplacé, dans la formule de complexité, le terme $s \cdot (n \cdot \log n + s \cdot |\mathcal{E}| \cdot \log n)$ par $s \cdot n \cdot (n/k + \log(k \cdot s))$. On peut donc *a priori* supposer que, du point de vue de la complexité, *dynamic2* est plus avantageuse que *dynamic1* si la taille des paquets (k) ou le nombre de serveurs

(s) ou le nombre d'arêtes dans le graphe d'application ($|\mathcal{E}|$) sont assez grands. Un nombre de tâches (n) trop grand aura quant à lui plutôt tendance à être à l'avantage de *dynamic1* (tous les autres paramètres étant fixés). Nous verrons ce qu'il en est en pratique dans la section suivante. Nous verrons également quelle est la perte de qualité de *dynamic2* par rapport à *dynamic1* pour différentes valeurs de k.

Variantes des heuristiques dynamiques

Comme pour les heuristiques statiques, nous avons une variante *critic* et une variante *mct* pour ces heuristiques dynamiques. La variante *readiness* est incluse d'office dans les heuristiques dynamiques (ligne 8 de l'algorithme 4.5).

4.6 Évaluation par simulations

Afin de comparer nos heuristiques, nous avons procédé comme au chapitre précédent. Nous avons exécuté nos heuristiques sur des graphes d'application et de plate-forme générés aléatoirement et simulé l'exécution des ordonnancements produits. Comme pour les expériences du chapitre précédent, nous avons écrit notre propre simulateur *ad hoc*. Le code consiste en quelques milliers de lignes de C++ [90], auxquelles se rajoutent une collection de scripts shell et/ou Perl [92] pour générer les graphes et extraire les résultats. Une grappe de 48 processeurs (Pentium IV Xeon 2,6 GHz) nous a permis d'effectuer un très grand nombre d'expériences que nous résumons dans cette section.

Nous commençons par décrire les graphes de plate-forme (section 4.6.1) ainsi que les graphes d'application (section 4.6.2) que nous avons utilisés pour comparer toutes les heuristiques. Les résultats obtenus sont ensuite présentés et discutés en section 4.6.3. Certaines heuristiques, sélectionnées en fonction de leur qualité et de leur coût, sont également comparées pour de plus grandes tailles de problème.

4.6.1 Plates-formes simulées

Nous avons testé nos heuristiques sur différents types de plates-formes qui ont été construites de la manière suivante :

graphes de plate-forme : ils sont composés de 7 serveurs. Le graphe d'interconnexion entre les serveurs est soit une clique, un arbre aléatoire ou un anneau. Pour ce qui est du routage (fixé, par nos hypothèses), il est naturellement fixé sur les arbres. Sur les anneaux, il n'y a que deux chemins possibles entre deux serveurs ; nous utilisons le plus court des deux (en terme de temps de communication). Pour les cliques, la liberté est plus grande dans le choix du

routage. Nous étudions deux situations extrêmes où le routage est fixé soit suivant une règle de plus court chemin en terme de temps de communication (la plate-forme est dénotée *clique-temps*), soit en utilisant toujours le chemin direct en un seul lien reliant deux serveurs (dénotée *clique-distance*). La seconde politique de routage correspond à des serveurs connectés *via* des liens de type WAN, suivant la vision de Casanova [23, fig. 2].

grappes : nous avons mesuré le temps de cycle de différents ordinateurs répartis entre Strasbourg et Lyon. De cet ensemble de valeurs, nous tirons aléatoirement des valeurs dont la différence avec la valeur moyenne est inférieure à l'écart type. Nous définissons de cette manière des grappes hétérogènes réalistes contenant 8, 16 ou 32 processeurs.

liens de communication : les bandes passantes des différents liens de communication entre les serveurs sont générées, suivant le même principe que l'ensemble des vitesses des processeurs, à partir de valeur mesurées entre des machines de Strasbourg et de Lyon

ratio des coûts de communication et de calcul : telles qu'elles ont été générées, les valeurs absolues des bandes passantes des liens de communication et des vitesses des processeurs n'ont pas de réelle signification (en pratique, elles sont dépendantes de l'application et doivent donc être pondérées par les caractéristiques de l'application). Nous sommes plus particulièrement intéressés par les valeurs relatives des vitesses de processeurs et des bandes passantes des liens réseau. C'est pourquoi nous normalisons les moyennes des caractéristiques des processeurs et des communications. Nous imposons alors le ratio entre le coût de communication et le coût de calcul, de manière à modéliser trois principaux types de problèmes : intensif en calculs (ratio = 0,1) intensif en communications (ratio = 10) et intermédiaire (ratio = 1). Les caractéristiques des processeurs et de liens de communication étant normalisées, ce ratio sera obtenu en faisant varier les moyennes des tailles des tâches et des données.

4.6.2 Graphes d'application

Nous faisons tourner les heuristiques sur les mêmes types de graphe d'application que dans la section 3.5 (cf. fig. 3.8, p. 60). Dans chaque cas, les tailles des données et des tâches sont tirées aléatoirement de manière uniforme entre 0,5 et 5. Les quatre types de graphe sont donc :

étoile : chaque graphe contient 70 sous-graphes en étoile, où chaque graphe en étoile est composé de 21 à 22 tâches dépendant d'une seule et même donnée (fig. 3.8(a)).

deux-un : chaque tâche dépend exactement de deux données : une donnée qui est partagée avec des autres tâches et une donnée non partagée (fig. 3.8(b)).

partitionné : le graphe est divisé en 20 morceaux de 75 tâches et dans chaque morceau, chaque tâche dépend aléatoirement de 1 à 10 données (fig. 3.8(c)). Le graphe contient donc au moins 20 composantes connexes,

aléatoire : chaque tâche dépend aléatoirement de 1 à 50 données (fig. 3.8(d)).

Chacun de nos graphes contient 1 500 tâches et 1 750 données, sauf pour les graphes en étoile qui contiennent également 1 500 tâches mais seulement 70 données. Pour éviter toute interférence entre les caractéristiques du graphe et le ratio des coûts de communication et de calcul, nous normalisons les ensembles de tâches et de données de manière à ce que la somme des tailles des données soit égale à la somme des tailles des tâches multiplié par le ratio des coûts de communication et de calcul. La distribution initiale des données est faite de manière aléatoire.

4.6.3 Résultats

Le tableau 4.1 résume toutes les expériences. Dans ce tableau, nous rapportons les performances des heuristiques, ainsi que leurs coûts (c'est-à-dire leurs temps d'exécution). C'est un résumé de 48 000 tests aléatoires (1 000 tests sur les quatre types de graphe 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 :

- *min-min* et ses variantes *min-min+critic* et *sufferage*.
- Les trois heuristiques statiques : *static*, *static+readiness*, *static+mct*. Ces heuristiques sont déclinées avec les variantes *max* (évaluation optimiste des coûts des communications, cf. section 4.5.1) et *critic*.
- Les heuristiques dynamiques *dynamic1* et *dynamic2*, la deuxième utilisant des ensembles, soit de $k = 10$, soit de $k = 100$ tâches. Ces heuristiques sont également déclinées avec les variantes *max*, *critic* et *mct*.
- L'heuristique naïve *randomtask* qui choisit le candidat local (le même pour tous les serveurs) de manière aléatoire, mais qui utilise ensuite les mêmes optimisations et schémas d'ordonnancement que les autres heuristiques. En d'autres termes, seul le choix de la prochaine tâche à exécuter est aléatoire. Le choix du serveur où elle sera exécutée (ce qui est crucial) est quant à lui déterminé par les politiques d'allocation précédentes. Cette heuristique est déclinée avec les variantes *critic* et *mct*.

Chaque heuristique (sauf *min-min+critic*) a été testée avec les deux variantes de l'algorithme 4.1, c'est-à-dire avec ordonnancement des communications par insertion (*insert*) ou sans. Des résultats plus détaillés peuvent être trouvés an annexe C.

Tableau 4.1 – Performance relative et coût des heuristiques : version de base et variante *mct*, avec ou sans ordonnancement des communications par insertion (*insert*). Moyennes sur toutes les 48 000 configurations ; les écarts types sont entre parenthèses.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,14 (±8%)	5 196 (±77%)			1,07 (±7%)	18 002 (±104%)		
min-min+critic	2,15 (±31%)	5 194 (±77%)						
sufferage	1,17 (±15%)	5 494 (±80%)			1,06 (±10%)	21 984 (±99%)		
static	2,29 (±34%)	2 (±107%)			1,56 (±28%)	4 (±91%)		
static+critic	2,67 (±41%)	3 (±87%)			1,61 (±29%)	5 (±79%)		
static+max	2,57 (±45%)	3 (±106%)			1,82 (±37%)	4 (±91%)		
static+max+critic	2,81 (±46%)	3 (±84%)			1,86 (±37%)	6 (±78%)		
static+readiness	2,19 (±33%)	3 (±104%)	1,48 (±24%)	7 (±92%)	1,53 (±29%)	4 (±92%)	1,19 (±13%)	13 (±65%)
static+readiness+critic	2,54 (±39%)	3 (±86%)	2,30 (±40%)	8 (±85%)	1,57 (±30%)	5 (±79%)	1,29 (±16%)	14 (±63%)
static+readiness+max	2,53 (±44%)	3 (±101%)	1,50 (±25%)	7 (±87%)	1,81 (±36%)	4 (±88%)	1,22 (±15%)	14 (±65%)
static+readiness+max+critic	2,79 (±46%)	3 (±81%)	2,28 (±40%)	8 (±81%)	1,85 (±36%)	6 (±78%)	1,32 (±17%)	16 (±64%)
dynamic1	2,41 (±38%)	10 (±80%)	1,34 (±17%)	14 (±79%)	1,67 (±40%)	11 (±77%)	1,10 (±9%)	19 (±72%)
dynamic1+critic	2,65 (±41%)	10 (±79%)	2,07 (±33%)	14 (±77%)	1,70 (±39%)	13 (±77%)	1,19 (±11%)	21 (±71%)
dynamic1+max	2,84 (±45%)	14 (±85%)	1,43 (±23%)	18 (±74%)	2,09 (±45%)	16 (±86%)	1,17 (±13%)	26 (±73%)
dynamic1+max+critic	3,46 (±48%)	15 (±88%)	2,24 (±39%)	18 (±75%)	2,12 (±44%)	19 (±89%)	1,27 (±16%)	28 (±73%)
dynamic2+10	2,58 (±41%)	42 (±117%)	2,03 (±41%)	21 (±108%)	1,66 (±39%)	43 (±112%)	1,44 (±31%)	28 (±93%)
dynamic2+10+critic	2,65 (±41%)	43 (±114%)	2,55 (±43%)	22 (±105%)	1,69 (±38%)	45 (±107%)	1,50 (±30%)	30 (±91%)
dynamic2+10+max	3,61 (±53%)	46 (±104%)	3,32 (±54%)	38 (±108%)	2,09 (±45%)	47 (±99%)	1,92 (±38%)	50 (±103%)
dynamic2+10+max+critic	3,47 (±49%)	47 (±102%)	3,73 (±48%)	39 (±106%)	2,12 (±44%)	50 (±94%)	1,99 (±37%)	53 (±102%)
dynamic2+100	2,99 (±56%)	8 (±104%)	2,83 (±49%)	10 (±87%)	1,72 (±38%)	9 (±88%)	1,69 (±39%)	19 (±66%)
dynamic2+100+critic	2,78 (±43%)	8 (±95%)	2,86 (±42%)	11 (±82%)	1,74 (±37%)	11 (±77%)	1,74 (±38%)	21 (±66%)
dynamic2+100+max	4,26 (±60%)	10 (±78%)	4,65 (±51%)	17 (±87%)	2,14 (±45%)	12 (±72%)	2,09 (±33%)	29 (±94%)
dynamic2+100+max+critic	3,55 (±50%)	11 (±76%)	4,01 (±42%)	18 (±87%)	2,16 (±44%)	14 (±71%)	2,16 (±32%)	31 (±96%)
randomtask	129,94 (±318%)	2 (±92%)	1,37 (±22%)	7 (±80%)	85,92 (±336%)	5 (±84%)	1,10 (±10%)	14 (±67%)

Comme précédemment, nous calculons les *performances relatives* et les *coûts relatifs* des heuristiques pour chacun des 48 000 tests (cf. section 3.5.3).

Les heuristiques *min-min*

Les heuristiques *min-min* et *sufferage* obtiennent, en moyenne, des performances similaires. Leur variantes où les communications sont ordonnancées par insertion sont les meilleures heuristiques, mais seulement de peu.

Les versions de base (des nouvelles heuristiques)

Les versions de base de nos heuristiques sont beaucoup plus rapides que les versions du *min-min* mais au prix d'une grande perte de qualité des ordonnancements produits (au moins deux fois plus mauvais). Comme on l'a observé avec un seul serveur (chapitre 3), la variante *readiness* a un impact significatif sur les performances. Pour l'heuristique statique, elle apporte un gain d'environ 5% pour un surcoût de 8%. Néanmoins, aucune de ces heuristiques n'inclus de schéma d'équilibrage de charge. Ainsi, avec la variante *readiness*, la plupart des tâches peuvent se retrouver ordonnancées sur une seule grappe, induisant un gros déséquilibre. C'est parfaitement illustré par *static+readiness* qui obtient une performance plus mauvaise sur les graphes en *étoile*.

Les heuristiques *dynamic* contiennent toutes les deux la politique *readiness* et une règle d'équilibrage de charge. Il pourrait donc être surprenant que les variantes des heuristiques *static* sont toutes meilleures que leurs contreparties *dynamic*. Cela s'explique par le fait que les heuristiques *dynamic* ne prennent pas en compte les contentions des liens réseau. Dans le but d'équilibrer la charge, ces heuristiques peuvent alors placer des tâches sur des serveurs qui sont connectés aux serveurs détenant des données par des liens de communication congestionnés. En pratique, les tâches ne seront pas exécutées simultanément et la charge ne sera donc pas équilibrée. C'est parfaitement illustré avec les graphes en *étoile* sur les plates-formes en *anneau* : *dynamic1* a une performance équivalente à 10% près de celle de *static* (avec ou sans *readiness*) sur les configurations intensives en communication (la politique d'équilibrage de charge n'a quasiment pas d'impact) et sur les configurations intensives en calculs (la congestion du réseau n'a quasiment pas d'impact), mais une performance 70% plus mauvaise sur les configurations intermédiaires. Cela suggère d'utiliser la variante *readiness* conjointement avec la variante *mct* qui équilibre naturellement la charge.

Avec *dynamic2*, comme prévu, plus la taille des paquets est grande, moins l'heuristique est chère, et moins bonne est la performance. De même, l'heuristique *randomtask* de base est sans surprise la pire des heuristiques avec une performance relative de 130.

L'évaluation pessimiste des coûts des communications aboutit presque toujours à de meilleures performances que l'évaluation optimiste (variante *max*), car la plupart des configurations sont propices à la congestion. La variante *critic* donne en général de plus mauvais résultats, en dépit du temps passé à réordonnancer les communications : c'est très vraisemblablement parce qu'elle favorise d'abord les longues communications, retardant ainsi toute l'exécution.

Pour finir, nous avons introduit l'heuristique *randomtask* naïve comme une borne inférieure de la qualité du placement et de l'ordonnancement des tâches. Avec une performance de 504, l'heuristique *randomtask* est extrêmement mauvaise sur les graphes en *étoile*. Il apparaît que, quand le graphe d'application contient des propriétés de partage évidentes, comme dans les graphes en *étoile*, nos heuristiques sont capables d'en tirer avantage. Au contraire, sur les graphes en *étoile*, *randomtask* est aussi mauvaise que d'habitude et l'écart se creuse. Sur les autres types de graphes d'application, la performance de *randomtask* n'est pas aussi mauvaise que ce qu'on pouvait attendre. Cela peut être dû à notre choix d'avoir une distribution initiale aléatoire des données sur les serveurs. Néanmoins, nos heuristiques montrent globalement des améliorations par rapport à *randomtask*.

La variante *mct*

La variante *mct* améliore grandement la qualité de nos heuristiques tandis que leur coût reste très bas. Par exemple, *dynamic1+mct* produit des ordonnancements qui ne sont que 18% plus longs que ceux de *min-min*, et elle les produit 376 fois plus rapidement. L'heuristique *dynamic2+mct* ne gagne pas autant et reste loin derrière *dynamic1+mct*. L'amélioration apportée par la variante *mct* est le mieux illustrée par *randomtask* : elle produit, 738 fois plus vite, des ordonnancements qui sont seulement 21% plus mauvais que ceux du *min-min*. Sur les graphes d'application très réguliers en *étoile*, la variante *mct* aboutit à des performances aussi bonnes que celles des heuristiques de référence, quelle que soit l'heuristique de base (sauf pour *dynamic2*). Sur les graphes de plate-forme *clique-distance*, *dynamic1* est de loin la meilleure heuristique. En fait, ce sont les seuls graphes de plate-forme qui correspondent bien à la définition des *coûts* dynamiques car ils ont très peu de congestion réseau.

Nos heuristiques ont le même comportement, quel que soit le ratio des coûts de communication et de calcul. De plus, les variantes *mct* de *dynamic1*, *randomtask* et *static+readiness* ont quasiment la même performance relative, quel que soit ce ratio, ce qui montre leur robustesse.

Ordonnancement par insertion

Nous avons également testé les heuristiques avec l'algorithme d'ordonnancement des communication par insertion (plutôt qu'avec l'algorithme glouton comme précédemment). Comme prédit, la qualité des résultats s'accroît significativement. Le surcoût est prohibitif pour les variantes du *min-min* alors que, comme on pouvait s'y attendre à la vue des formules de complexité, il l'est beaucoup moins pour nos heuristiques. En effet, le surcoût est dû à l'algorithme d'ordonnancement des communication qui est utilisé n fois moins souvent par nos heuristiques que par l'heuristique *min-min*. Par exemple, la version *dynamic1+mct+insert* produit des ordonnancement qui sont 3% *meilleurs* que ceux du *min-min* original, et il les produit 288 fois plus rapidement ! L'heuristique *randomtask+mct+insert* se comporte encore mieux : elle obtient une performance similaire encore plus rapidement.

Étude sur des plates-formes plus grandes

Nous avons 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 (voir le tableau 4.2). Nous avons étudié deux types de plate-forme :

(i) des graphes de plate-forme à 50 serveurs, avec 25 000 tâches et 32 000 données (1 250 pour les *étoiles*) ;

(ii) des graphes de plate-forme à 100 serveurs, avec 50 000 tâches et 62 500 données (2 500 pour les *étoiles*).

Les deux ensembles de plates-formes ont la même charge en moyenne par serveur, qui est 2,5 fois plus importante que sur nos graphes de plate-forme originaux à 7 serveurs. Ces configurations de simulation sont trop grosses pour les heuristiques *min-min*. La configuration à 50 serveurs est la plus grosse sur laquelle nous avons pu tester la variante *insert*, connaissant notre environnement de simulation.

Sur ces configuration plus grosses, nous remarquons principalement que :

1. la variante *mct* est, une fois de plus essentielle ;
2. les variantes *mct* des heuristiques *dynamic* ont le comportement (relatif) prévu : *dynamic1* est plus longue mais meilleure que *dynamic2+10* qui est plus longue mais meilleure que *dynamic2+100* ;
3. *randomtask+mct+insert* est maintenant significativement plus mauvaise que *dynamic1+mct+insert*, alors qu'elle était aidée par la petite taille des configurations à 7 serveurs (de manière plus marquée sur les graphes d'application *deux-un*) ;
4. *dynamic1+mct* est significativement meilleure que *randomtask+mct*, sauf pour les graphes en *étoile* où elles ont des résultats similaires.

Tableau 4.2 – 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%)		

Résumé

Dans le tableau 4.2, nous présentons une sélection des heuristiques, en fonction de leur performance et de leur coût. Nous pouvons voir que, exceptés *min-min* et *sufferage* qui deviennent rapidement trop chères, un bon choix est d'utiliser *dynamic1+mct* (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, *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, *dynamic1+mct* construit un ordonnancement en moyenne en 511 secondes, alors que cela coûterait *a priori* plus de 48 heures à *min-min*.

4.7 Conclusion

Nous avons traité, dans ce chapitre, le problème de l'ordonnancement d'un grand ensemble de tâches indépendantes, mais partageant des données, sur des collections de serveurs distribués.

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 [7] 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 [78, 79] 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 *DataPresent* qui est dans l'idée proche de notre politique *readiness*. 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 : le graphe d'application ressemble à nos graphes en *étoile*.

Nous avons cherché à apporter des solutions d'ordonnancement pour des graphes d'application plus généraux. D'un point de vue théorique, nous avons montré la complexité du simple problème de décider où déplacer les données. D'un point de vue pratique, notre contribution est à deux volets :

– Nous avons proposé une extension de l'heuristique *min-min* pour l'adapter

à notre problème ; cela s'est montré plus difficile que prévu, à cause des problèmes d'ordonnancement des communications.
– Nous avons réussi à concevoir une collection de nouvelles heuristiques qui ont de bonnes performances mais dont le coût (en terme de temps de calcul) est d'un ordre de grandeur plus petit que celui de l'heuristique *min-min*. Les meilleures heuristiques ont été obtenues en combinant les variantes *readiness*, *mct* et *insert*. Plus particulièrement, sur les petites plates-formes, l'heuristique *randomtask+mct+insert* produit des ordonnancements dont le *makespan* est seulement 4% plus long que pour ceux produits par la meilleure variante de *min-min* (*sufferage+insert*), et les produit 371 fois plus rapidement que la variante de *min-min* la plus rapide (*min-min* de base). Sur des plates-formes plus larges, le temps d'exécution des heuristiques *min-min*, ainsi que des variantes *insert*, devient prohibitif. L'heuristique de choix est alors *dynamic1+mct*.

Les nombreuses simulations que nous avons menées nous ont permis de comparer les différentes heuristiques, et de voir lesquelles étaient les plus pertinentes. Nous prévoyons maintenant d'explorer d'autres voies pour la conception de nouvelles heuristiques. Toutes nos heuristiques n'utilisent en effet qu'une vision locale des tâches et des serveurs. Il serait intéressant d'essayer de concevoir des heuristiques utilisant les propriétés des graphes de plate-forme et des graphes d'application, afin d'avoir une approche plus globale du problème. Il serait également intéressant d'explorer des algorithmes d'ordonnancement à la volée (*on-line scheduling*) où les tâches à exécuter arrivent au fur et à mesure dans le système. Bien que les heuristiques présentées dans ce travail soient toutes statiques (*off-line*), les heuristiques *dynamic* et *randomtask* pourraient être utilisées pour de tels ordonnancements. Cependant, pour éviter des phénomènes de famine (des tâches dont l'exécution serait sans cesse retardée), elles devraient être améliorées. Cette amélioration peut, par exemple, se faire en tenant compte de l'âge des tâches, autrement dit du temps passé par chaque tâche à attendre d'être ordonnancée. Une solution serait alors d'imposer un âge limite au-delà duquel les tâches sont à ordonnancer de manière prioritaire.

Chapitre 5

Conclusion et perspectives

5.1 Conclusion

Nous avons, dans cette thèse, étudié des stratégies d'ordonnancement et d'équilibrage de charge dans le cadre de plates-formes hétérogènes distribuées. Notre problème était d'ordonnancer un ensemble de tâches indépendantes afin de réduire le temps total d'exécution du système. Ces tâches utilisent des données d'entrée qui peuvent être *partagées* : chaque tâche peut utiliser plusieurs données, et chaque donnée peut être utilisée par plusieurs tâches. Les tâches ont des durées d'exécution différentes, et les données ont des tailles différentes. Toute la difficulté est alors de réussir à placer sur un même processeur des tâches partageant des données, tout en conservant un bon équilibrage de la charge des différents processeurs.

Dans un premier temps, nous nous sommes limités au cas où il n'y a pas de partage de données et où les tailles des tâches et des données sont homogènes. Nous avons alors introduit, pour des plates-formes de type maître-esclave, deux algorithmes calculant une distribution optimale des données. Nous avons également présenté une heuristique, bien plus rapide que nos algorithmes, tout en étant garantie. Nous avons enfin proposé une politique sur l'ordre dans lequel les esclaves devaient être considérés par le maître. Les gains obtenus par nos solutions ont été illustrés par des résultats expérimentaux avec une application scientifique réelle.

Le partage des données a été introduit dans le chapitre suivant, ainsi que l'hétérogénéité pour les tailles des tâches et des données. La plate-forme restait de type maître-esclave. Nous avons alors exploré la complexité du problème, et nous avons montré deux nouveaux résultats : avec un seul processeur, si les tailles des tâches et des données sont hétérogènes, le problème est déjà NP-complet. Si ces tailles sont homogènes, le problème est NP-complet à partir de deux processeurs hétérogènes. Nous avons ensuite conçu plusieurs nouvelles heuristiques pour résoudre le problème d'ordonnancement. Ces nouvelles heuristiques, ainsi que des heuristiques

classiques comme *min-min* et *sufferage* (nos heuristiques dites de référence) ont été comparées entre elles à l'aide de simulations intensives. Nous avons ainsi montré que nos nouvelles heuristiques réussissent à obtenir des performances aussi bonnes que les heuristiques de référence, tout en ayant une complexité algorithmique d'un ordre de grandeur plus faible.

Dans le dernier chapitre nous avons 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 montré que le simple fait d'ordonnancer les transferts des données est un problème difficile. D'un point de vue pratique, nous avons montré comment adapter les heuristiques au cas des serveurs distribués. Nous avons également introduit des heuristiques capables d'adapter dynamiquement leurs choix au fur et à mesure que des données sont répliquées sur les serveurs. De la même manière que précédemment, nous avons montré, à l'aide de nombreuses simulations, que nos nouvelles heuristiques ont des performances aussi bonnes que les heuristiques de référence, tout en ayant une complexité algorithmique d'un ordre de grandeur plus faible.

5.2 Perspectives

Cette thèse s'achève ici, mais cela ne signifie pas que tous les problèmes ont été résolus ! Nos travaux peuvent être poursuivis suivant différents axes. Nous allons maintenant présenter quelques unes des perspectives ouvertes.

Complexité

Nous avons exploré la complexité théorique de l'ordonnancement d'un ensemble de tâches partageant des données, mais il reste un cas auquel nous n'avons pas répondu. En effet, nous ne connaissons pas la complexité du problème dans le cas simple où les tâches et les données sont de tailles unitaires, et où la plate-forme est composée de processeurs de vitesse unitaire, reliés au maître par un lien de bande passante également unitaire (cf. section 3.3). La complexité de ce problème reste inconnue, même dans le cas où il n'y a qu'un seul processeur.

Dans le cas où les tailles des tâches et les tailles des données sont hétérogènes, avec un seul processeur, nous pouvons remarquer qu'il est possible de trouver un ordonnancement optimal pour certains types de graphes. C'est en particulier vrai pour les graphes que nous avons appelé en *étoile* (cf. figure 3.8). En effet, on peut remarquer qu'on ne perd rien à exécuter de manière consécutive l'ensemble des tâches dépendant d'une même donnée. Ainsi, en groupant ces tâches en une seul grosse tâche, on peut se ramener à un problème que l'on sait résoudre par l'algorithme de Johnson [52]. On peut procéder d'une manière similaire pour les

graphes *deux-un* dans le cas où il n'y a qu'une seule composante connexe (il suffit d'envoyer la donnée partagée en premier). Il serait ainsi intéressant de caractériser les graphes pour lesquels nous pouvons donner un algorithme produisant un ordonnancement optimal en temps polynomial et ceux pour lesquels on ne le peut pas.

La résolution de ces problèmes pourrait aider à cerner de manière plus précise où se situe la difficulté de notre problème d'ordonnancement.

Une approche plus globale

Toutes les heuristiques qui ont été développées dans les chapitres 3 et 4, pour résoudre le problème de l'ordonnancement des tâches sur les serveurs, utilisent une vision locale du graphe d'application pour évaluer le coût du placement d'une tâche sur un serveur. Notre but initial était de trouver des heuristiques locales d'aussi bonne qualité et plus rapides que celles de Casanova *et al.* [25, 24]. Nous pensons avoir atteint notre but. Cependant, une étude des cas pathologiques où l'heuristique *randomtask* atteint de bien meilleures performances que les heuristiques plus sophistiquées nous montre les limites de notre approche. Nous remarquons en effet que l'heuristique *randomtask* place quelques fois des tâches qui, localement, apparaissent coûteuses. Ces choix peuvent, d'un point de vue global, s'avérer finalement bénéfique car la tâche apporte alors des données qui seront beaucoup réutilisées par la suite. Une approche locale ne peut pas prendre une telle décision volontairement. Seule une vision globale permettrait de faire des choix plus intelligents.

Une approche possible pour résoudre de manière globale notre problème de l'allocation des tâches sur les serveurs est d'utiliser des méthodes de partitionnement de graphe. Le problème de partitionnement de graphe (*min k-cut*) consiste à séparer les sommets d'un graphe en plusieurs partitions telles que les sommes des poids des sommets de chaque partition soient égales, et que la somme des poids des arêtes coupées par le partitionnement soit minimale. C'est une technique couramment utilisée pour la conception de circuits VLSI [4] ou bien le partitionnement d'applications parallèles [48, 59, 91]. Ce problème est NP-complet, mais un certain nombre de bibliothèques existent, implémentant différentes heuristiques pour le résoudre [26, 72, 75, 85].

Intuitivement, nous voudrions construire un graphe dont les sommets correspondraient aux tâches, et les arêtes représenteraient les données partagées entre les tâches. Il faudrait alors partitionner ce graphe en autant de partitions qu'il y a de serveurs, tout en minimisant le nombre d'arêtes coupées par cette partition. Une arête coupée représente en effet la réplication d'une donnée sur plusieurs serveurs (c'est l'idée qui avait été utilisée dans la preuve de complexité de TSFDR-MoveDec au chapitre 4). En réalité, le problème est bien plus compliqué. En particulier,

le partitionnement doit être contraint par le placement originel des données. Les partitions ne doivent pas être de même taille en terme de somme des tailles des tâches, car les différents serveurs ne sont pas de même puissance. De la même manière, le réseau n'est pas complet et il n'est pas homogène. La mesure de la taille de la coupe doit donc être adaptée.

L'allocation des tâches sur les serveurs étant faite, il restera à régler le problème de leur ordonnancement au sein de chaque serveur.

Modèle de communication

Dans le chapitre 4, nous avons étendu le modèle de plate-forme. Le modèle de communication est cependant resté très restrictif : à un instant donné, chaque serveur peut être impliqué dans au plus une émission et une réception (contrainte un-port) et chacun des liens réseau peut être utilisé par au plus une communication. Ces contraintes ont l'inconvénient de provoquer des goulots d'étranglement pour les communications. Il serait intéressant d'étendre ce modèle afin de limiter ces contraintes. Pour cela, un modèle multi-port peut être envisagé, autorisant les serveurs à effectuer plusieurs communications en même temps. Conjointement à ça, plusieurs communications peuvent être autorisées à avoir lieu en parallèle sur un même lien. On pourra par exemple, pour modéliser le partage de la bande passante entre ces communications, se référer au modèle proposé par Casanova dans [23] ou bien à l'approche utilisée par Legrand, Renard, Robert et Vivien dans [65] .

Dans le chapitre 4, nous faisions l'hypothèse que le routage au sein du graphe de plate-forme était fixé. Cette hypothèse a elle aussi le risque de générer des goulots d'étranglements pour les communications. Un routage plus dynamique capable de s'adapter aux congestions de la plate-forme pourrait être imaginé.

Robustesse

Pour toutes nos heuristiques, nous avons supposé que nous avions une connaissance parfaite de tous les paramètres du problème. Si, en pratique, la taille d'une donnée est un paramètre qui peut s'obtenir facilement ce n'est pas le cas pour les durées des tâches. On peut supposer n'obtenir au mieux qu'une estimation grossière de la durée d'une tâche. Il serait ainsi intéressant d'étudier la robustesse de nos heuristiques face à des erreurs de prédiction et, le cas échéant, de les modifier afin de les rendre moins sensibles à ces erreurs.

Nous avons également supposé avoir une plate-forme parfaite, dont les performances obtenues seraient exactement celles prédites. En situation réelle, ces conditions idéales ne peuvent en général pas être garanties : les réseaux utilisés sont souvent partagés avec d'autres utilisateurs, de même que les machines. Il serait donc également intéressant d'étudier le comportement des différentes heu-

ristiques dans des environnements où les prédictions ne sont qu'approximatives. Pour cela, un simulateur comme SimGrid [62] pourrait nous être utile. En particulier, SimGrid permet de simuler des variations des capacités de la plate-forme, en s'appuyant sur des traces du comportement de plates-formes réelles.

Bibliographie

[1] Ammar H. Alhusaini, Viktor K. Prasanna et Cauligi S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. Dans HCW 1999 [45], p. 156–165.

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel et Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5): 749–771, mai 2002.

[3] Francisco Almeida, Daniel González et Luz Marina Moreno. The master-slave paradigm on heterogeneous systems: A dynamic programming approach for the optimal mapping. Dans *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, p. 266–272. IEEE Computer Society Press, La Corogne, Espagne, février 2004.

[4] Charles J. Alpert et Andrew B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19(1-2):1–81, août 1995.

[5] Deniz Turgay Altilar et Yakup Paker. Optimal scheduling algorithms for communication constrained parallel processing. Dans *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference*, tome 2400 de *Lecture Notes in Computer Science*, p. 197–206. Springer-Verlag, Paderborn, Allemagne, août 2002.

[6] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Meyers et David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, octobre 1990.

[7] Alessandro Amoroso, Keith Marzullo et Aleta Ricciardi. 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, Amsterdam, Pays-Bas, mai 1998.

[8] Cyril Banino, Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand et Yves Robert. Scheduling strategies for master-slave tas-

king on heterogeneous processor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, avril 2004.

[9] Jorge G. Barbosa, João Tavares et Armando Jorge Padilha. Linear algebra algorithms in heterogeneous cluster of personal computers. Dans HCW 2000 [46], p. 147–159.

[10] Gerassimos Barlas et Bharadwaj Veeravalli. Quantized load distribution for tree and bus-connected processors. *Parallel Computing*, 30(7):841–865, juillet 2004.

[11] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand et Yves Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. Dans IPDPS 2002 [51], p. 67 (6 pages).

[12] Olivier Beaumont, Arnaud Legrand et Yves Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. Dans *17th International Symposium on Computer and Information Sciences (ISCIS XVII)*, p. 115–119. CRC Press, Orlando, FL, USA, octobre 2002.

[13] Olivier Beaumont, Arnaud Legrand et Yves Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems. Dans HCW 2003 [47], p. 98b (14 pages). Workshop tenu conjointement avec IPDPS.

[14] Olivier Beaumont, Arnaud Legrand et Yves Robert. Scheduling divisible workloads on heterogeneous platforms. *Parallel Computing*, 29(9): 1121–1152, septembre 2003.

[15] Francine Berman. High-performance schedulers. Dans Foster et Kesselman [37], chapitre 12, p. 279–309.

[16] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil T. Spring, Alan Su et Dmitrii Zagorodnov. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, avril 2003.

[17] Veeravalli Bharadwaj, Debasish Ghose, Venkataraman Mani et Thomas G. Robertazzi. *Scheduling divisible loads in parallel and distributed systems*. Wiley–IEEE Computer Society Press, 1996.

[18] Veeravalli Bharadwaj, Debasish Ghose et Thomas G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, janvier 2003.

[19] Jacek Błażewicz et Maciej Drozdowski. Distributed processing of divisible jobs with communication startup costs. *Discrete Applied Mathematics,*

Second International Colloquium on Graphs and Optimization, 76(1-3):21–41, juin 1997.

[20] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest et Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, août 1973.

[21] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen et Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, juin 2001.

[22] Jacques Carlier et Philippe Chrétienne. *Problèmes d'ordonnancement : modélisation, complexité, algorithmes*. Études et recherches en informatique. Masson, 1988.

[23] Henri Casanova. Modeling large-scale platforms for the analysis and the simulation of scheduling strategies. Dans *6th Workshop on Advances in Parallel and Distributed Computational Models (APDCM'04)*, p. 170a (8 pages). IEEE Computer Society Press, Santa Fe, NM, USA, avril 2004. Workshop tenu conjointement avec IPDPS.

[24] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov et Francine Berman. 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.

[25] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov et Francine Berman. Heuristics for scheduling parameter sweep applications in Grid environments. Dans HCW 2000 [46], p. 349–363.

[26] Chaco: Software for partitioning graphs.
URL `http://www.cs.sandia.gov/~bahendr/chaco.html`

[27] Hyeong-Ah Choi et S. Louis Hakimi. Scheduling file transfers for trees and odd cycles. *SIAM Journal on Computing*, 16(1):162–168, février 1987.

[28] Philippe Chrétienne, Edward G. Coffman, Jr., Jan Karel Lenstra et Zhen Liu (éds.). *Scheduling Theory and Its Applications*. John Wiley & Sons, 1995.

[29] Michał Cierniak, Mohammed Javeed Zaki et Wei Li. Compile-time scheduling algorithms for heterogeneous network of workstations. *The Computer Journal, special issue on Automatic Loop Parallelization*, 40(6):356–372, décembre 1997.

[30] Edward G. Coffman, Jr., Michael R. Garey, David S. Johnson et Andrea S. LaPaugh. Scheduling file transfers. *SIAM Journal on Computing*, 14(3):744–780, août 1985.

[31] Thomas H. Cormen, Charles E. Leiserson et Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[32] Fabrício Alves Barbosa da Silva, Sílvia Carvalho et Eduardo Raul Hruschka. A scheduling algorithm for running bag-of-tasks data mining applications on the grid. Dans Euro-Par 2004 [34], p. 254–262.

[33] Aaron E. Darling, Lucas Carey et Wu-chun Feng. The design, implementation, and evaluation of mpiBLAST. Dans *4th International Conference on Linux Clusters: The HPC Revolution 2003 (LCI 03)*. Linux Cluster Institute, San Jose, CA, USA, juin 2003.

[34] *Euro-Par 2004, Parallel Processing, 10th International Euro-Par Conference*, tome 3149 de *Lecture Notes in Computer Science*. Springer-Verlag, Pise, Italie, août/septembre 2004.

[35] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, septembre 1988.

[36] Ian Foster et Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, été 1997.

[37] Ian Foster et Carl Kesselman (éds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[38] Michael R. Garey et David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[39] Michael R. Garey, David S. Johnson et Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, mai 1976.

[40] Stéphane Genaud, Arnaud Giersch et Frédéric Vivien. Load-balancing scatter operations for grid computing. Dans HCW 2003 [47], p. 101a (10 pages). Workshop tenu conjointement avec IPDPS.

[41] William L. George. Dynamic load-balancing for data-parallel MPI programs. Dans *Message Passing Interface Developer's and User's Conference (MPIDC'99)*. Atlanta, GA, USA, mars 1999.

[42] Jean-Pierre Goux, Sanjeev R. Kulkarni, Michael Yoder et Jeff Linderoth. Master-Worker: An enabling framework for applications on the computational Grid. *Cluster Computing*, 4(1):63–70, mars 2001.

[43] Marc Grunberg, Stéphane Genaud et Catherine Mongenet. Seismic ray-tracing and Earth mesh modeling on various parallel architectures. *The Journal of Supercomputing*, 29(1):27–44, juillet 2004.

[44] Thomas J. Hacker, Brian D. Athey et Brian Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. Dans IPDPS 2002 [51], p. 46b (10 pages).

[45] *8th Heterogeneous Computing Workshop (HCW'1999)*. IEEE Computer Society Press, San Juan, Puerto Rico, avril 1999.

[46] *9th Heterogeneous Computing Workshop (HCW'2000)*. IEEE Computer Society Press, Cancún, Mexique, mai 2000.

[47] *12th Heterogeneous Computing Workshop (HCW'2003)*. IEEE Computer Society Press, Nice, France, avril 2003. Workshop tenu conjointement avec IPDPS.

[48] Bruce Hendrickson et Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing, special issue on Graph Partitioning and Parallel Computing*, 26(12):1519–1534, novembre 2000.

[49] Elisa Heymann, Miquel A. Senar, Emilio Luque et Miron Livny. Adaptive scheduling for master-worker applications on the computational Grid. Dans *1st IEEE/ACM International Workshop on Grid Computing (GRID 2000)*, tome 1971 de *Lecture Notes in Computer Science*, p. 214–227. Springer-Verlag, Bangalore, Inde, décembre 2000.

[50] Parry Husbands et James C. Hoe. MPI-StarT: Delivering network performance to numerical applications. Dans *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC'98)*, p. 17. IEEE Computer Society Press, Orlando, FL, USA, novembre 1998.

[51] *16th International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society Press, Fort Lauderdale, FL, USA, avril 2002.

[52] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, mars 1954.

[53] Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William Gropp, Ewing Lusk et John Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. Dans *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, p. 377–384. IEEE Computer Society Press, Cancún, Mexique, mai 2000.

[54] Nicholas T. Karonis, Brian Toonen et Ian Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel*

and Distributed Computing, special issue on Computational Grids, 63(5): 551–563, mai 2003.

[55] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat et Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, août 1999.

[56] Hyoung Joong Kim, Gyu-In Jee et Jang Gyu Lee. Optimal load distribution for tree network processors. *IEEE Transactions on Aerospace and Electronic Systems*, 32(2):607–612, avril 1996.

[57] Dénes König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen*, 77:453–465, 1916.

[58] Barbara Kreaseck, Larry Carter, Henri Casanova et Jeanne Ferrante. On the interference of communication on computation in Java. Dans *3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 2004)*, p. 246a (8 pages). IEEE Computer Society Press, Santa Fe, NM, USA, avril 2004. Workshop tenu conjointement avec IPDPS.

[59] Shailendra Kumar, Sajal K. Das et Rupak Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. Dans IPDPS 2002 [51], p. 66 (7 pages).

[60] Sébastien Lacour, Christian Pérez et Thierry Priol. A network topology description model for grid application deployment. Dans *5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, p. 61–68. IEEE Computer Society Press, Pittsburgh, PA, USA, novembre 2004.

[61] Arnaud Legrand. *Algorithmique parallèle hétérogène et techniques d'ordonnancement : approches statiques et dynamiques.* Thèse de doctorat, LIP, École Normale Supérieure de Lyon, France, décembre 2003.

[62] Arnaud Legrand, Loris Marchal et Henri Casanova. Scheduling distributed applications: the SimGrid simulation framework. Dans *3rd IEEE/ ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, p. 138–145. IEEE Computer Society Press, Tokyo, Japon, mai 2003.

[63] Arnaud Legrand, Frédéric Mazoit et Martin Quinson. An Application-Level Network Mapper. Rapport de recherche n° 2003-09, LIP, École Normale Supérieure de Lyon, France, février 2003.

[64] Arnaud Legrand et Martin Quinson. Automatic deployment of the Network Weather Service using the Effective Network View. Dans *High-Performance Grid Computing Workshop*, p. 272b (8 pages). IEEE Computer

Society Press, Santa Fe, NM, USA, avril 2004. Workshop tenu conjointement avec IPDPS.

[65] Arnaud Legrand, Hélène Renard, Yves Robert et Frédéric Vivien. Mapping and load-balancing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):546–558, juin 2004.

[66] Yaohang Li et Michael Mascagni. 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, Baltimore, MD, USA, novembre 2002.

[67] Elder Magalhães Macambira et Cid Carvalho de Souza. The edge-weighted clique problem: Valid inequalities, facets and polyhedral computations. *European Journal of Operational Research*, 123(2):346–371, juin 2000.

[68] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen et Richard F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. Dans HCW 1999 [45], p. 30–44.

[69] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen et Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, novembre 1999.

[70] Weizhen Mao. Directed file transfer scheduling. Dans *ACM 31st Annual Southeast Conference (ACM-SE 1993)*, p. 199–203. ACM Press, Birmingham, AL, USA, avril 1993.

[71] Weizhen Mao et Rahul Simha. Routing and scheduling file transfers in packet-switched networks. *Journal of Computing and Information, special issue: Proceedings of the 6th International Conference on Computing and Information (ICCI'94)*, 1(1):559–574, avril 1995. CD-ROM.

[72] MeTiS: Family of multilevel partitioning algorithms.
URL `http://www-users.cs.umn.edu/~karypis/metis/index.html`

[73] MPI Forum. *MPI: A Message Passing Interface Standard, Version 1.1.* University of Tennessee, Knoxville, TN, USA, juin 1995.

[74] Ekow J. Otoo et Arie Shoshani. Accurate modeling of cache replacement policies in a data grid. Dans *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, p. 10–19. IEEE Computer Society Press, San Diego, CA, USA, avril 2003.

[75] PARTY partitioning library.
URL `http://wwwcs.upb.de/fachbereich/AG/monien/RESEARCH/PART/party.html`

[76] PIP/PipLib.
URL `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`

[77] Martin Quinson. *Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué*. Thèse de doctorat, LIP, École Normale Supérieure de Lyon, France, décembre 2003.

[78] Kavitha Ranganathan et Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. Dans *11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, p. 352–358. IEEE Computer Society Press, Edimbourg, Écosse, juillet 2002.

[79] Kavitha Ranganathan et Ian Foster. Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1(1):53–62, 2003.

[80] Pedro I. Rivera Vega, Ravi Varadarajan et Shamkant B. Navathe. Scheduling file transfers in fully connected networks. *Networks: an International Journal*, 22(6):563–588, 1992.

[81] Thomas G. Robertazzi. Processor equivalence for a linear daisy chain of load sharing processors. *IEEE Transactions on Aerospace and Electronic Systems*, 29(4):1216–1221, octobre 1993.

[82] Taher Saif et Manish Parashar. Understanding the behavior and performance of non-blocking communications in MPI. Dans Euro-Par 2004 [34], p. 173–182.

[83] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro et Aliandro Lima. 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, New-York, NY, USA, juin 2004.

[84] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, tome 24 de *Algorithms and Combinatorics*. Springer-Verlag, 2003.

[85] Scotch: Static mapping, graph partitioning, and sparse matrix block ordering package.
URL `http://www.labri.fr/Perso/~pelegrin/scotch/`

[86] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, Dept. of Computer Science and Engineering, University of California, San Diego, CA, USA, 2001.

[87] Gary Shao, Francine Berman et Richard Wolski. Master/slave computing on the Grid. Dans HCW 2000 [46], p. 3–16.

[88] Behrooz A. Shirazi, Ali R. Hurson et Krishna M. Kavi (éds.). *Scheduling and Load Balancing in Parallel and Distributed Systems*. Wiley–IEEE Computer Society Press, 1995.

[89] Joel R. Stiles, Thomas M. Bartol, Jr., Edwin E. Salpeter et Miriam M. Salpeter. Monte Carlo simulation of neuro-transmitter release using MCell, a general simulator of cellular physiological processes. Dans *Computational Neuroscience: Trends in Research, 1998*, p. 279–284. Plenum Press, 1998.

[90] Bjarne Stroustrup. *Le langage C++*. CampusPress France, 3e édition, 1999.

[91] Bora Uçar et Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[92] Larry Wall, Tom Christiansen et Jon Orwant. *Programmation en Perl*. O'Reilly, 3e édition, 2001.

[93] Jennifer Whitehead. The complexity of file transfer scheduling with forwarding. *SIAM Journal on Computing*, 19(2):222–245, avril 1990.

[94] Richard Wolski, Neil T. Spring et Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, octobre 1999.

[95] Yang Yang et Henri Casanova. A multi-round algorithm for scheduling divisible workload applications: Analysis and experimental evaluation. Rapport technique nº CS2002-0721, Dept. of Computer Science and Engineering, University of California, San Diego, CA, USA, septembre 2002.

Annexe A

Notations

Nous avons essayé, dans la mesure du possible, de conserver les mêmes notations pour l'ensemble de la thèse. Ces notations sont résumées ici en indiquant, à chaque fois, ce qu'elle représentent.

Chapitre 2

- n représente le nombre de tâches/données
- p représente le nombre de processeurs
- $\{P_1, \ldots, P_p\}$ représente l'ensemble des processeurs
- $T_{comm}(i, x)$ représente le temps nécessaire au processeur P_i pour recevoir x données ; dans seconde partie, $T_{comm}(i, x) = \lambda_i \cdot x$
- $T_{calc}(i, x)$ représente temps nécessaire au processeur P_i pour traiter x données ; dans la seconde partie, $T_{calc}(i, x) = \mu_i \cdot x$

Chapitre 3

- n représente le nombre de tâches
- $\mathcal{T} = \{T_1, \ldots, T_n\}$ représente l'ensemble des tâches
- t_i représente la taille de la tâche T_i
- m représente le nombre de données
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ représente l'ensemble des données
- d_j représente la taille de la donnée D_j
- $\mathcal{G} = (\mathcal{V} = \mathcal{T} \cup \mathcal{D}, \mathcal{E})$ représente le graphe d'application
- p représente le nombre de processeurs
- $\{P_1, \ldots, P_p\}$ représente l'ensemble des processeurs
- c_i représente l'inverse de la bande passante vers le processeur P_i
- w_i représente le temps de cycle du processeur P_i

Chapitre 4

- pour les tâches, les données et le graphe d'application, les mêmes notations que dans le chapitre 3 sont utilisée
- s représente le nombre de serveurs
- $\mathcal{S} = \{S_1, \ldots, S_s\}$ représente l'ensemble des serveurs
- $\mathcal{R} = \{R_1, \ldots, R_r\}$ représente l'ensemble des routeurs
- $\mathcal{L} = \{(u, v)\}$ avec $u \in \mathcal{S} \cup \mathcal{R}$ et $v \in \mathcal{S} \cup \mathcal{R}$ représente l'ensemble des liens réseau
- $\mathcal{P} = (\mathcal{S} \cup \mathcal{R}, \mathcal{L})$ représente le graphe de plate-forme
- b_k représente la bande passante du lien l_k
- $S_i = (E_i, C_i)$ représente le serveur S_i, avec E_i l'entrepôt et C_i la grappe associés
- p_i représente le nombre de processeurs de la grappe C_i
- $\{P_{i,1}, \ldots P_{i,p_i}\}$ représente l'ensemble des processeur de la grappe C_i
- $s_{i,k}$ représente la vitesse du processeur $P_{i,k}$

Annexe B

Résultats expérimentaux pour le chapitre 3

Dans cette section sont rassemblés les résultats des simulations menées pour le chapitre 3. Le tableau B.1 regroupe les coûts relatifs des différentes heuristiques. Sur les pages suivantes, les graphiques représentent les performances relatives des heuristiques, pour chaque type de graphe d'application et pour les trois ratios des coûts de communication et de calcul.

Tableau B.1 – Coûts relatifs des heuristiques : moyenne et détail par graphe.

	étoile	deux-un	partitionné	aléatoire	moyenne
duration	1,942	1,048	1,060	1,026	1,269
duration+shared	2,092	1,057	1,076	1,842	1,517
duration+readiness	1,991	1,152	1,244	1,395	1,446
duration+locality	2,204	1,085	1,129	1,292	1,427
payoff	1,010	1,032	1,057	1,182	1,070
payoff+shared	1,018	1,040	1,073	2,151	1,321
payoff+readiness	1,062	1,134	1,243	1,545	1,246
payoff+locality	1,346	1,085	1,138	1,428	1,250
advance	1,957	1,056	1,121	1,163	1,324
advance+shared	2,110	1,064	1,116	1,937	1,557
advance+readiness	2,015	1,159	1,281	1,405	1,465
advance+locality	2,219	1,092	1,186	1,432	1,482
johnson	2,382	1,138	1,187	1,357	1,516
johnson+shared	2,160	1,148	1,191	2,277	1,694
johnson+readiness	2,379	1,207	1,395	1,861	1,710
johnson+locality	2,474	1,157	1,283	1,654	1,642
communication	1,639	1,010	1,046	1,186	1,220
communication+shared	1,647	1,021	1,062	2,152	1,471
communication+readiness	1,791	1,111	1,232	1,466	1,400
communication+locality	1,970	1,065	1,129	1,43	1,398
computation	2,160	1,019	1,035	1,099	1,328
computation+readiness	2,119	1,123	1,269	1,763	1,569
computation+locality	2,304	1,072	1,128	1,389	1,473
min-min	339,2	220,3	388,4	728,8	419,2
max-min	329,3	214,0	330,4	697,7	392,8
sufferage	366,2	227,1	317,0	596,5	376,7
sufferage II	974,2	476,2	532,4	736,9	679,9
sufferage X	990,1	483,9	511,4	603,8	647,3

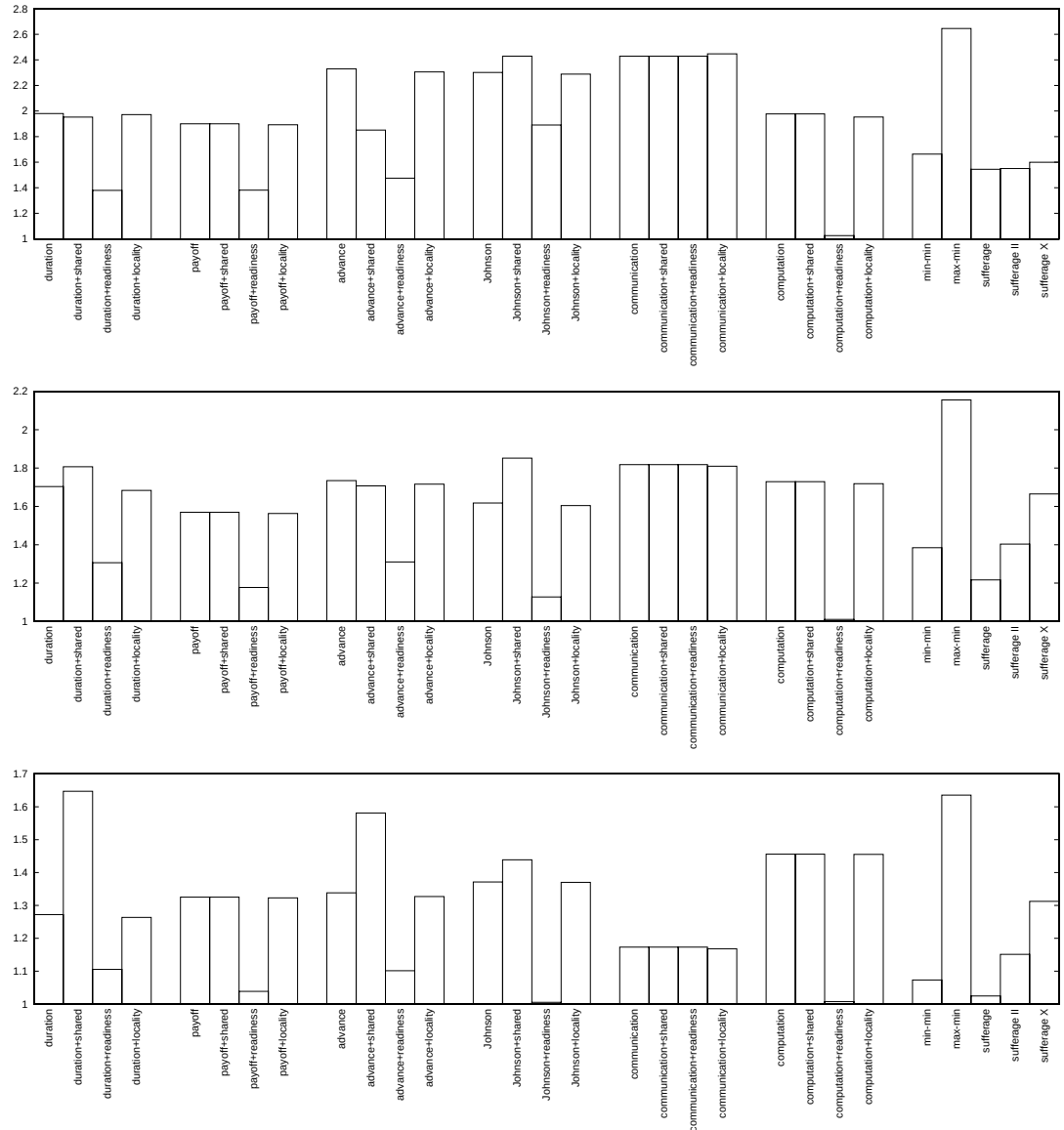


Figure B.1 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les graphes en *étoile* avec un ratio des coûts de communication et de calcul égal, de haut en bas, à : 1/10, 1 et 10.

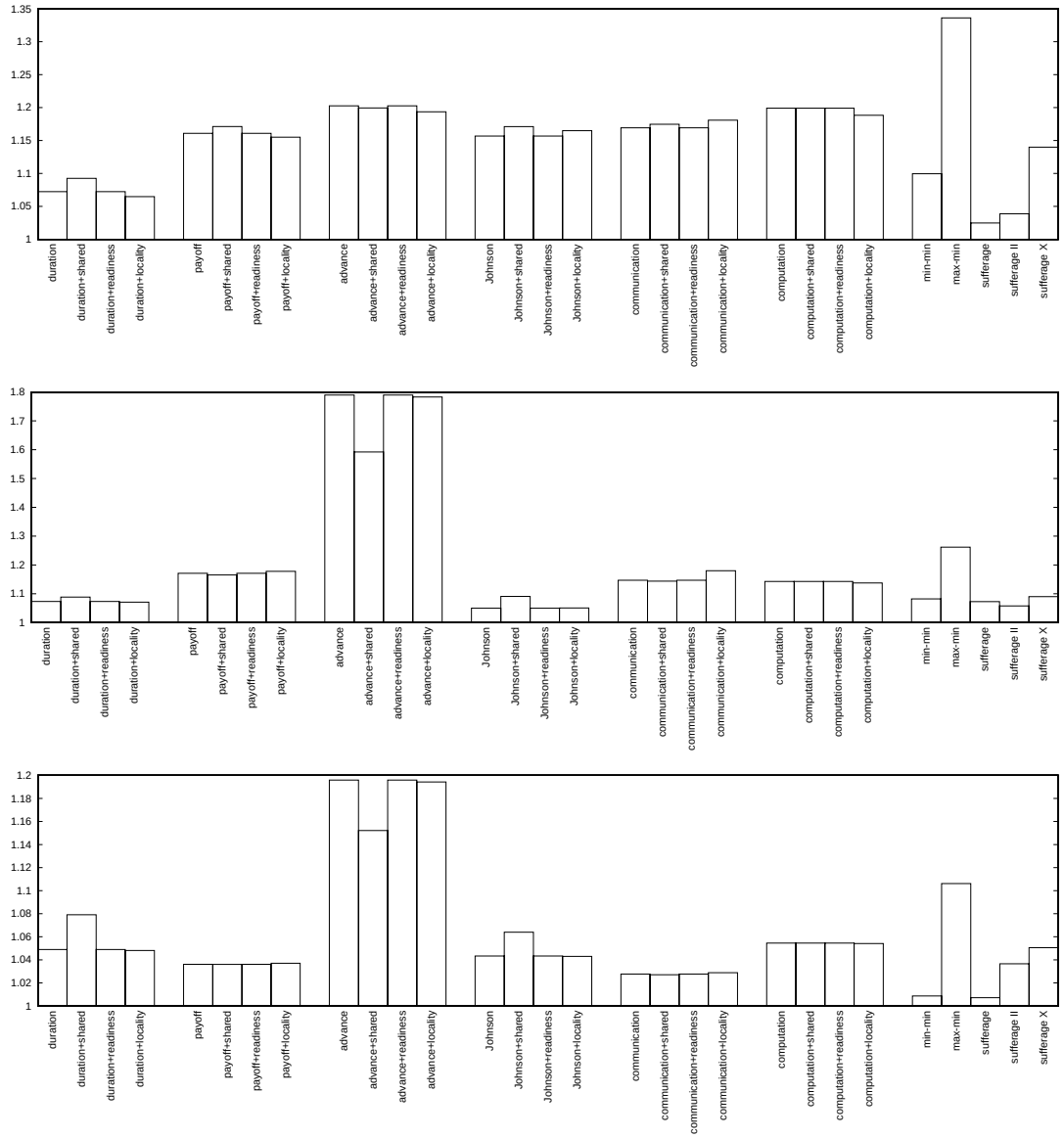


Figure B.2 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les graphes *deux-un* avec un ratio des coûts de communication et de calcul égal, de haut en bas, à : 1/10, 1 et 10.

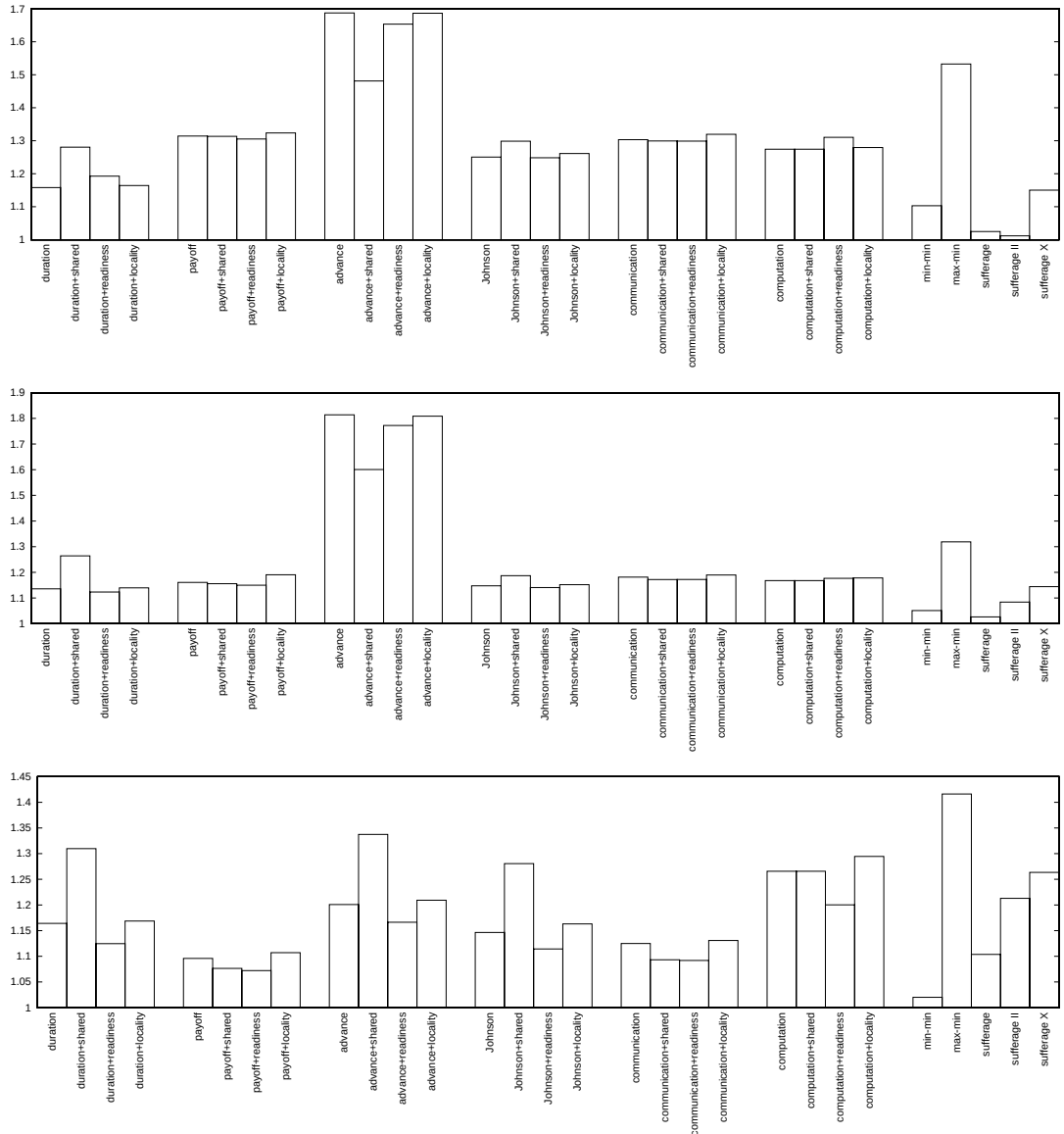


Figure B.3 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les graphes *partitionnés* avec un ratio des coûts de communication et de calcul égal, de haut en bas, à : 1/10, 1 et 10.

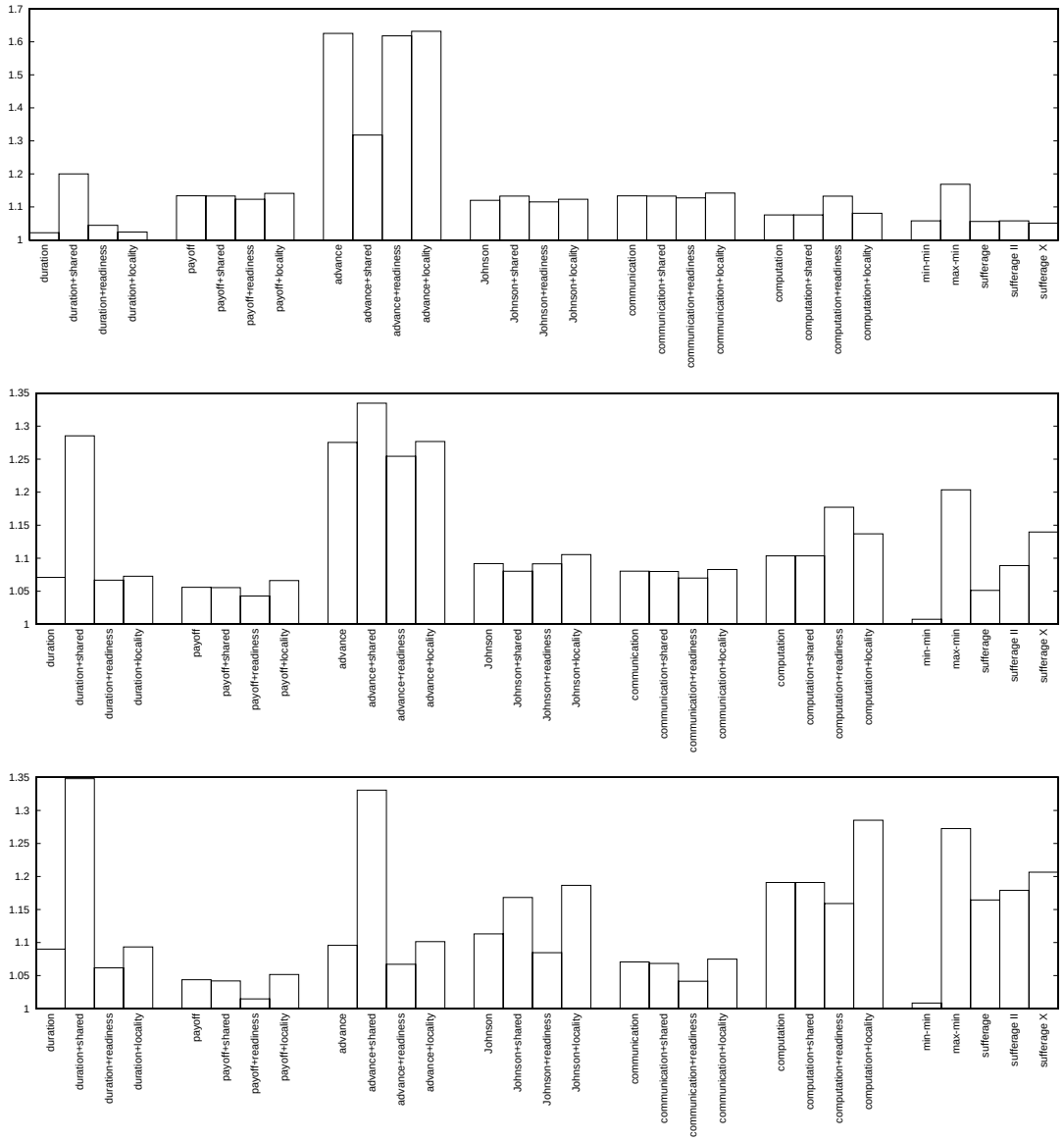


Figure B.4 – Performances relatives des ordonnancements produits par les différentes heuristiques ; moyenne sur les graphes *aléatoires* avec un ratio des coûts de communication et de calcul égal, de haut en bas, à : 1/10, 1 et 10.

Annexe C

Résultats expérimentaux pour le chapitre 4

Sur les 16 pages suivantes sont rassemblés les résultats des simulations menées pour le chapitre 4. Les moyennes sont rapportées pour toutes les heuristiques qui ont été évaluées. Les heuristiques ont été évaluées sur trois tailles de plates-formes : 7, 50 et 100 serveurs. Pour chacune, en plus des moyennes générales, des moyennes plus détaillées sont données, ventilées par :

- types de graphes d'application ;
- types de graphes de plate-forme ;
- ratios des coûts de communication et de calcul.

Tableau C.1 – **Plates-formes avec 7 serveurs** : moyennes sur toutes les configurations.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,14 (±8%)	5 196 (±77%)			1,07 (±7%)	18 002 (±104%)		
min-min+critic	2,15 (±31%)	5 194 (±77%)						
sufferage	1,17 (±15%)	5 494 (±80%)			1,06 (±10%)	21 984 (±99%)		
static	2,29 (±34%)	2 (±107%)			1,56 (±28%)	4 (±91%)		
static+critic	2,67 (±41%)	3 (±87%)			1,61 (±29%)	5 (±79%)		
static+max	2,57 (±45%)	3 (±106%)			1,82 (±37%)	4 (±91%)		
static+max+critic	2,81 (±46%)	3 (±84%)			1,86 (±37%)	6 (±78%)		
static+readiness	2,19 (±33%)	3 (±104%)	1,48 (±24%)	7 (±92%)	1,53 (±29%)	4 (±92%)	1,19 (±13%)	13 (±65%)
static+readiness+critic	2,54 (±39%)	3 (±86%)	2,30 (±40%)	8 (±85%)	1,57 (±30%)	5 (±79%)	1,29 (±16%)	14 (±63%)
static+readiness+max	2,53 (±44%)	3 (±101%)	1,50 (±25%)	7 (±87%)	1,81 (±36%)	4 (±88%)	1,22 (±15%)	14 (±65%)
static+readiness+max+critic	2,79 (±46%)	3 (±81%)	2,28 (±40%)	8 (±81%)	1,85 (±36%)	6 (±78%)	1,32 (±17%)	16 (±64%)
dynamic1	2,41 (±38%)	10 (±80%)	1,34 (±17%)	14 (±79%)	1,67 (±40%)	11 (±77%)	1,10 (±9%)	19 (±72%)
dynamic1+critic	2,65 (±41%)	10 (±79%)	2,07 (±33%)	14 (±77%)	1,70 (±39%)	13 (±77%)	1,19 (±11%)	21 (±71%)
dynamic1+max	2,84 (±45%)	14 (±85%)	1,43 (±23%)	18 (±74%)	2,09 (±45%)	16 (±86%)	1,17 (±13%)	26 (±73%)
dynamic1+max+critic	3,46 (±48%)	15 (±88%)	2,24 (±39%)	18 (±75%)	2,12 (±44%)	19 (±89%)	1,27 (±16%)	28 (±73%)
dynamic2+10	2,58 (±41%)	42 (±117%)	2,03 (±41%)	21 (±108%)	1,66 (±39%)	43 (±112%)	1,44 (±31%)	28 (±93%)
dynamic2+10+critic	2,65 (±41%)	43 (±114%)	2,55 (±43%)	22 (±105%)	1,69 (±38%)	45 (±107%)	1,50 (±30%)	30 (±91%)
dynamic2+10+max	3,61 (±53%)	46 (±104%)	3,32 (±54%)	38 (±108%)	2,09 (±45%)	47 (±99%)	1,92 (±38%)	50 (±103%)
dynamic2+10+max+critic	3,47 (±49%)	47 (±102%)	3,73 (±48%)	39 (±106%)	2,12 (±44%)	50 (±94%)	1,99 (±37%)	53 (±102%)
dynamic2+100	2,99 (±56%)	8 (±104%)	2,83 (±49%)	10 (±87%)	1,72 (±38%)	9 (±88%)	1,69 (±39%)	19 (±66%)
dynamic2+100+critic	2,78 (±43%)	8 (±95%)	2,86 (±42%)	11 (±82%)	1,74 (±37%)	11 (±77%)	1,74 (±38%)	21 (±66%)
dynamic2+100+max	4,26 (±60%)	10 (±78%)	4,65 (±51%)	17 (±87%)	2,14 (±45%)	12 (±72%)	2,09 (±33%)	29 (±94%)
dynamic2+100+max+critic	3,55 (±50%)	11 (±76%)	4,01 (±42%)	18 (±87%)	2,16 (±44%)	14 (±71%)	2,16 (±32%)	31 (±96%)
randomtask	129,94 (±318%)	2 (±92%)	1,37 (±22%)	7 (±80%)	85,92 (±336%)	5 (±84%)	1,10 (±10%)	14 (±67%)

Tableau C.2 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes d'applications en étoile**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,09 (±8%)	10 249 (±25%)			1,09 (±8%)	18 328 (±25%)		
min-min+critic	1,34 (±30%)	10 204 (±25%)						
sufferage	1,05 (±8%)	11 359 (±25%)			1,04 (±8%)	22 396 (±29%)		
static	1,49 (±35%)	6 (±39%)			1,49 (±35%)	6 (±70%)		
static+critic	1,49 (±35%)	6 (±38%)			1,49 (±35%)	7 (±65%)		
static+max	1,49 (±35%)	6 (±39%)			1,49 (±35%)	6 (±70%)		
static+max+critic	1,49 (±35%)	6 (±38%)			1,49 (±35%)	7 (±64%)		
static+readiness	1,52 (±38%)	6 (±38%)	1,07 (±7%)	15 (±25%)	1,52 (±38%)	6 (±67%)	1,06 (±7%)	18 (±32%)
static+readiness+critic	1,52 (±38%)	6 (±36%)	1,11 (±13%)	16 (±25%)	1,52 (±38%)	7 (±61%)	1,09 (±10%)	18 (±31%)
static+readiness+max	1,52 (±38%)	6 (±38%)	1,07 (±7%)	15 (±26%)	1,52 (±38%)	6 (±67%)	1,06 (±7%)	18 (±32%)
static+readiness+max+critic	1,52 (±38%)	6 (±36%)	1,11 (±13%)	16 (±25%)	1,52 (±38%)	7 (±60%)	1,09 (±10%)	18 (±31%)
dynamic1	1,84 (±47%)	14 (±40%)	1,07 (±8%)	22 (±30%)	1,80 (±46%)	14 (±40%)	1,06 (±7%)	24 (±28%)
dynamic1+critic	1,88 (±47%)	14 (±40%)	1,12 (±13%)	23 (±30%)	1,82 (±46%)	14 (±40%)	1,10 (±11%)	25 (±28%)
dynamic1+max	1,84 (±47%)	14 (±26%)	1,07 (±8%)	22 (±23%)	1,80 (±46%)	14 (±40%)	1,06 (±7%)	25 (±29%)
dynamic1+max+critic	1,88 (±47%)	14 (±26%)	1,12 (±13%)	23 (±23%)	1,82 (±46%)	15 (±39%)	1,10 (±11%)	25 (±29%)
dynamic2+10	1,78 (±46%)	114 (±21%)	1,43 (±31%)	27 (±25%)	1,74 (±45%)	114 (±21%)	1,43 (±31%)	28 (±24%)
dynamic2+10+critic	1,81 (±46%)	114 (±21%)	1,46 (±31%)	28 (±24%)	1,75 (±45%)	114 (±21%)	1,45 (±31%)	28 (±24%)
dynamic2+10+max	1,78 (±46%)	115 (±21%)	1,52 (±32%)	25 (±38%)	1,74 (±45%)	114 (±21%)	1,51 (±32%)	26 (±39%)
dynamic2+10+max+critic	1,81 (±46%)	116 (±21%)	1,54 (±32%)	26 (±39%)	1,75 (±45%)	115 (±21%)	1,53 (±32%)	27 (±40%)
dynamic2+100	1,67 (±51%)	19 (±25%)	1,77 (±55%)	19 (±24%)	1,62 (±49%)	18 (±25%)	1,84 (±55%)	22 (±30%)
dynamic2+100+critic	1,66 (±50%)	19 (±25%)	1,86 (±55%)	19 (±25%)	1,63 (±49%)	19 (±25%)	1,89 (±55%)	22 (±30%)
dynamic2+100+max	1,67 (±51%)	19 (±22%)	1,85 (±53%)	16 (±39%)	1,62 (±49%)	19 (±25%)	2,01 (±53%)	18 (±42%)
dynamic2+100+max+critic	1,66 (±50%)	19 (±22%)	1,95 (±53%)	17 (±39%)	1,63 (±49%)	19 (±25%)	2,08 (±53%)	19 (±43%)
randomtask	504,36 (±140%)	4 (±54%)	1,06 (±7%)	14 (±27%)	335,06 (±150%)	5 (±89%)	1,05 (±6%)	16 (±37%)

Tableau C.3 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes d'applications deux-un**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,16 (±6%)	3 950 (±94%)			1,12 (±6%)	9 132 (±99%)		
min-min+critic	2,38 (±16%)	3 956 (±94%)						
sufferage	1,13 (±6%)	4 185 (±94%)			1,03 (±5%)	11 641 (±90%)		
static	2,43 (±15%)	2 (±136%)			1,47 (±18%)	4 (±107%)		
static+critic	2,64 (±16%)	2 (±125%)			1,50 (±21%)	4 (±105%)		
static+max	2,58 (±18%)	2 (±133%)			1,84 (±26%)	4 (±109%)		
static+max+critic	2,74 (±18%)	2 (±121%)			1,90 (±29%)	5 (±107%)		
static+readiness	2,39 (±15%)	2 (±129%)	1,63 (±10%)	6 (±103%)	1,44 (±17%)	4 (±108%)	1,24 (±9%)	12 (±93%)
static+readiness+critic	2,59 (±16%)	2 (±119%)	2,48 (±16%)	7 (±104%)	1,46 (±19%)	5 (±104%)	1,29 (±9%)	13 (±93%)
static+readiness+max	2,53 (±18%)	2 (±128%)	1,61 (±9%)	6 (±104%)	1,79 (±25%)	4 (±108%)	1,28 (±10%)	13 (±94%)
static+readiness+max+critic	2,68 (±17%)	3 (±115%)	2,43 (±14%)	7 (±105%)	1,85 (±27%)	5 (±106%)	1,34 (±9%)	14 (±95%)
dynamic1	2,37 (±20%)	9 (±112%)	1,53 (±10%)	14 (±109%)	1,59 (±27%)	11 (±109%)	1,19 (±9%)	20 (±102%)
dynamic1+critic	2,60 (±22%)	10 (±113%)	2,37 (±16%)	14 (±109%)	1,61 (±27%)	12 (±107%)	1,23 (±9%)	21 (±102%)
dynamic1+max	2,29 (±22%)	10 (±110%)	1,46 (±11%)	15 (±109%)	1,63 (±29%)	11 (±109%)	1,17 (±11%)	22 (±103%)
dynamic1+max+critic	2,66 (±23%)	10 (±111%)	2,35 (±17%)	15 (±110%)	1,67 (±29%)	13 (±109%)	1,23 (±10%)	23 (±103%)
dynamic2+10	2,75 (±25%)	34 (±112%)	1,91 (±18%)	35 (±111%)	1,59 (±27%)	36 (±111%)	1,39 (±21%)	41 (±106%)
dynamic2+10+critic	2,63 (±22%)	35 (±112%)	2,59 (±19%)	36 (±111%)	1,63 (±27%)	37 (±111%)	1,45 (±20%)	43 (±107%)
dynamic2+10+max	2,93 (±28%)	35 (±111%)	2,23 (±22%)	82 (±64%)	1,65 (±29%)	36 (±111%)	1,65 (±28%)	97 (±64%)
dynamic2+10+max+critic	2,71 (±23%)	35 (±112%)	3,02 (±19%)	84 (±64%)	1,70 (±29%)	37 (±111%)	1,73 (±26%)	100 (±64%)
dynamic2+100	3,38 (±37%)	6 (±113%)	2,69 (±27%)	11 (±106%)	1,71 (±26%)	8 (±108%)	1,56 (±20%)	19 (±96%)
dynamic2+100+critic	2,83 (±21%)	7 (±114%)	2,82 (±22%)	12 (±106%)	1,74 (±26%)	9 (±106%)	1,63 (±20%)	20 (±96%)
dynamic2+100+max	3,81 (±39%)	7 (±112%)	3,60 (±18%)	26 (±65%)	1,81 (±29%)	8 (±109%)	1,93 (±16%)	41 (±64%)
dynamic2+100+max+critic	2,99 (±24%)	7 (±114%)	3,52 (±16%)	27 (±64%)	1,85 (±29%)	9 (±107%)	2,03 (±16%)	43 (±64%)
randomtask	6,09 (±23%)	2 (±118%)	1,46 (±8%)	6 (±97%)	3,02 (±18%)	5 (±94%)	1,08 (±7%)	13 (±95%)

Tableau C.4 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes d'applications partitionnés**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,18 (±7%)	2 842 (±69%)			1,06 (±5%)	11 107 (±72%)		
min-min+critic	2,54 (±22%)	2 844 (±68%)						
sufferage	1,25 (±11%)	2 992 (±61%)			1,04 (±8%)	15 530 (±69%)		
static	2,66 (±27%)	1 (±73%)			1,60 (±27%)	3 (±67%)		
static+critic	3,18 (±28%)	2 (±73%)			1,68 (±28%)	5 (±65%)		
static+max	2,95 (±33%)	1 (±80%)			1,92 (±34%)	3 (±73%)		
static+max+critic	3,31 (±31%)	2 (±73%)			1,98 (±34%)	5 (±69%)		
static+readiness	2,59 (±28%)	1 (±81%)	1,79 (±20%)	3 (±71%)	1,59 (±28%)	3 (±67%)	1,30 (±14%)	10 (±63%)
static+readiness+critic	3,10 (±29%)	2 (±73%)	3,01 (±29%)	4 (±70%)	1,67 (±29%)	5 (±66%)	1,43 (±15%)	12 (±64%)
static+readiness+max	2,93 (±33%)	1 (±75%)	1,84 (±21%)	4 (±68%)	1,91 (±34%)	3 (±68%)	1,36 (±16%)	13 (±62%)
static+readiness+max+critic	3,29 (±31%)	2 (±72%)	3,05 (±28%)	5 (±70%)	1,97 (±33%)	5 (±71%)	1,49 (±16%)	15 (±63%)
dynamic1	2,40 (±23%)	5 (±77%)	1,46 (±11%)	8 (±77%)	1,35 (±21%)	7 (±74%)	1,07 (±6%)	13 (±71%)
dynamic1+critic	2,59 (±19%)	6 (±75%)	2,46 (±22%)	8 (±76%)	1,40 (±20%)	8 (±72%)	1,18 (±8%)	15 (±71%)
dynamic1+max	2,97 (±23%)	7 (±75%)	1,76 (±21%)	10 (±76%)	2,01 (±27%)	10 (±72%)	1,31 (±14%)	20 (±68%)
dynamic1+max+critic	3,83 (±23%)	8 (±75%)	3,01 (±28%)	11 (±77%)	2,05 (±26%)	12 (±72%)	1,43 (±15%)	22 (±68%)
dynamic2+10	2,60 (±28%)	12 (±79%)	2,08 (±20%)	14 (±77%)	1,38 (±21%)	14 (±76%)	1,26 (±17%)	23 (±71%)
dynamic2+10+critic	2,65 (±20%)	13 (±79%)	2,79 (±22%)	15 (±75%)	1,43 (±21%)	15 (±75%)	1,34 (±16%)	25 (±70%)
dynamic2+10+max	4,19 (±36%)	13 (±77%)	3,82 (±16%)	20 (±117%)	2,03 (±27%)	16 (±77%)	1,87 (±20%)	34 (±118%)
dynamic2+10+max+critic	3,86 (±24%)	14 (±76%)	4,46 (±13%)	20 (±116%)	2,07 (±26%)	18 (±75%)	1,99 (±19%)	36 (±117%)
dynamic2+100	3,32 (±51%)	3 (±74%)	3,48 (±40%)	6 (±70%)	1,59 (±27%)	5 (±73%)	1,53 (±20%)	17 (±66%)
dynamic2+100+critic	3,05 (±26%)	3 (±75%)	3,27 (±31%)	6 (±72%)	1,63 (±25%)	6 (±73%)	1,59 (±20%)	19 (±66%)
dynamic2+100+max	5,03 (±44%)	4 (±76%)	6,06 (±18%)	8 (±119%)	2,11 (±27%)	6 (±74%)	1,98 (±14%)	20 (±117%)
dynamic2+100+max+critic	3,98 (±25%)	5 (±76%)	4,77 (±13%)	9 (±116%)	2,13 (±26%)	9 (±72%)	2,06 (±12%)	23 (±118%)
randomtask	4,47 (±28%)	1 (±63%)	1,62 (±19%)	4 (±63%)	2,50 (±23%)	4 (±69%)	1,16 (±11%)	13 (±64%)

Tableau C.5 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes d'applications aléatoires**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,12 (±6%)	3 704 (±60%)			1,02 (±4%)	33 443 (±88%)		
min-min+critic	2,34 (±23%)	3 713 (±60%)						
sufferage	1,27 (±19%)	3 439 (±55%)			1,11 (±15%)	38 366 (±91%)		
static	2,59 (±31%)	1 (±37%)			1,69 (±26%)	3 (±75%)		
static+critic	3,37 (±35%)	3 (±63%)			1,76 (±27%)	5 (±76%)		
static+max	3,24 (±45%)	1 (±61%)			2,04 (±41%)	3 (±72%)		
static+max+critic	3,71 (±42%)	3 (±60%)			2,08 (±40%)	6 (±72%)		
static+readiness	2,25 (±30%)	1 (±56%)	1,43 (±15%)	3 (±58%)	1,56 (±28%)	2 (±75%)	1,17 (±11%)	11 (±59%)
static+readiness+critic	2,93 (±35%)	2 (±61%)	2,59 (±24%)	4 (±57%)	1,64 (±29%)	4 (±76%)	1,34 (±13%)	14 (±58%)
static+readiness+max	3,16 (±44%)	1 (±54%)	1,49 (±16%)	3 (±57%)	2,01 (±40%)	3 (±71%)	1,19 (±11%)	12 (±63%)
static+readiness+max+critic	3,66 (±42%)	3 (±58%)	2,54 (±23%)	5 (±57%)	2,06 (±39%)	6 (±73%)	1,37 (±13%)	15 (±63%)
dynamic1	3,01 (±40%)	10 (±65%)	1,29 (±10%)	11 (±61%)	1,94 (±41%)	13 (±66%)	1,08 (±8%)	19 (±62%)
dynamic1+critic	3,53 (±40%)	12 (±64%)	2,32 (±21%)	13 (±62%)	1,97 (±39%)	16 (±69%)	1,25 (±10%)	22 (±62%)
dynamic1+max	4,27 (±30%)	26 (±58%)	1,44 (±12%)	24 (±61%)	2,92 (±39%)	30 (±58%)	1,16 (±10%)	37 (±60%)
dynamic1+max+critic	5,48 (±25%)	28 (±59%)	2,49 (±21%)	25 (±62%)	2,94 (±38%)	36 (±58%)	1,33 (±12%)	41 (±59%)
dynamic2+10	3,18 (±42%)	7 (±60%)	2,71 (±44%)	9 (±59%)	1,94 (±40%)	9 (±63%)	1,69 (±37%)	20 (±64%)
dynamic2+10+critic	3,53 (±40%)	9 (±61%)	3,37 (±41%)	11 (±60%)	1,97 (±39%)	13 (±68%)	1,75 (±35%)	23 (±66%)
dynamic2+10+max	5,54 (±31%)	19 (±58%)	5,70 (±23%)	24 (±101%)	2,94 (±39%)	22 (±58%)	2,66 (±33%)	43 (±100%)
dynamic2+10+max+critic	5,51 (±25%)	21 (±58%)	5,90 (±19%)	27 (±98%)	2,96 (±38%)	28 (±58%)	2,69 (±32%)	49 (±99%)
dynamic2+100	3,61 (±54%)	2 (±59%)	3,38 (±45%)	5 (±59%)	1,94 (±41%)	4 (±67%)	1,82 (±35%)	18 (±62%)
dynamic2+100+critic	3,57 (±41%)	4 (±63%)	3,47 (±38%)	6 (±62%)	1,97 (±39%)	8 (±72%)	1,84 (±33%)	22 (±62%)
dynamic2+100+max	6,54 (±35%)	11 (±59%)	7,07 (±25%)	16 (±101%)	3,01 (±38%)	14 (±59%)	2,44 (±28%)	35 (±100%)
dynamic2+100+max+critic	5,57 (±26%)	14 (±58%)	5,79 (±21%)	18 (±100%)	3,02 (±38%)	20 (±59%)	2,46 (±27%)	40 (±100%)
randomtask	4,85 (±34%)	1 (±50%)	1,34 (±20%)	4 (±51%)	3,12 (±39%)	4 (±66%)	1,11 (±10%)	14 (±66%)

Tableau C.6 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes de plate-forme clique-temps**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,14 (±7%)	4 610 (±74%)			1,06 (±7%)	17 025 (±68%)		
min-min+critic	2,24 (±28%)	4 597 (±73%)						
sufferage	1,16 (±12%)	4 920 (±78%)			1,04 (±7%)	20 016 (±59%)		
static	2,30 (±30%)	2 (±94%)			1,63 (±25%)	3 (±85%)		
static+critic	2,66 (±34%)	3 (±74%)			1,69 (±26%)	5 (±61%)		
static+max	2,64 (±39%)	2 (±92%)			1,98 (±33%)	4 (±84%)		
static+max+critic	2,86 (±38%)	3 (±70%)			2,05 (±33%)	5 (±61%)		
static+readiness	2,19 (±29%)	2 (±93%)	1,47 (±22%)	6 (±89%)	1,59 (±27%)	3 (±87%)	1,18 (±13%)	12 (±48%)
static+readiness+critic	2,53 (±33%)	3 (±74%)	2,30 (±36%)	7 (±80%)	1,65 (±27%)	5 (±64%)	1,30 (±15%)	13 (±43%)
static+readiness+max	2,59 (±39%)	2 (±90%)	1,51 (±23%)	7 (±83%)	1,96 (±33%)	4 (±82%)	1,22 (±15%)	13 (±44%)
static+readiness+max+critic	2,81 (±38%)	3 (±70%)	2,30 (±36%)	7 (±74%)	2,02 (±33%)	5 (±61%)	1,34 (±16%)	14 (±40%)
dynamic1	2,53 (±39%)	9 (±59%)	1,39 (±17%)	12 (±62%)	1,78 (±41%)	10 (±53%)	1,11 (±10%)	17 (±44%)
dynamic1+critic	2,79 (±41%)	9 (±58%)	2,11 (±31%)	13 (±59%)	1,82 (±40%)	11 (±52%)	1,22 (±12%)	19 (±42%)
dynamic1+max	3,12 (±47%)	13 (±64%)	1,49 (±23%)	16 (±50%)	2,35 (±47%)	15 (±64%)	1,20 (±14%)	24 (±47%)
dynamic1+max+critic	3,74 (±48%)	14 (±65%)	2,27 (±37%)	17 (±50%)	2,38 (±46%)	17 (±67%)	1,32 (±16%)	25 (±47%)
dynamic2+10	2,77 (±40%)	39 (±121%)	2,15 (±40%)	17 (±60%)	1,77 (±40%)	40 (±116%)	1,51 (±34%)	23 (±43%)
dynamic2+10+critic	2,79 (±41%)	40 (±118%)	2,67 (±41%)	18 (±57%)	1,81 (±39%)	42 (±110%)	1,58 (±32%)	25 (±41%)
dynamic2+10+max	4,03 (±51%)	43 (±107%)	2,88 (±51%)	21 (±30%)	2,34 (±48%)	44 (±101%)	1,93 (±53%)	29 (±21%)
dynamic2+10+max+critic	3,75 (±49%)	44 (±105%)	3,39 (±50%)	22 (±30%)	2,37 (±47%)	46 (±95%)	1,98 (±51%)	31 (±27%)
dynamic2+100	3,22 (±51%)	7 (±105%)	2,96 (±44%)	9 (±74%)	1,83 (±38%)	8 (±85%)	1,76 (±38%)	17 (±40%)
dynamic2+100+critic	2,92 (±41%)	8 (±93%)	2,96 (±38%)	10 (±66%)	1,87 (±37%)	10 (±67%)	1,82 (±36%)	18 (±38%)
dynamic2+100+max	4,71 (±54%)	10 (±71%)	3,75 (±51%)	12 (±50%)	2,41 (±47%)	11 (±60%)	1,91 (±38%)	19 (±39%)
dynamic2+100+max+critic	3,85 (±48%)	10 (±67%)	3,55 (±46%)	13 (±51%)	2,44 (±46%)	13 (±55%)	1,97 (±36%)	22 (±41%)
randomtask	86,37 (±297%)	2 (±63%)	1,38 (±19%)	7 (±73%)	61,38 (±305%)	4 (±66%)	1,09 (±8%)	13 (±45%)

Tableau C.7 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes de plate-forme clique-distance**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,16 (±8%)	5 715 (±80%)			1,07 (±7%)	24 714 (±132%)		
min-min+critic	2,44 (±28%)	5 738 (±80%)						
sufferage	1,23 (±17%)	5 845 (±82%)			1,05 (±7%)	29 402 (±132%)		
static	2,64 (±32%)	3 (±118%)			1,86 (±26%)	5 (±101%)		
static+critic	3,00 (±37%)	4 (±101%)			1,93 (±28%)	6 (±102%)		
static+max	3,23 (±42%)	3 (±116%)			2,28 (±34%)	5 (±98%)		
static+max+critic	3,39 (±42%)	4 (±95%)			2,33 (±34%)	7 (±95%)		
static+readiness	2,52 (±31%)	3 (±115%)	1,58 (±25%)	8 (±93%)	1,84 (±27%)	5 (±102%)	1,27 (±16%)	15 (±89%)
static+readiness+critic	2,87 (±36%)	4 (±98%)	2,66 (±40%)	9 (±89%)	1,90 (±29%)	6 (±101%)	1,40 (±19%)	17 (±89%)
static+readiness+max	3,18 (±41%)	3 (±111%)	1,66 (±27%)	9 (±91%)	2,26 (±33%)	5 (±96%)	1,31 (±19%)	17 (±90%)
static+readiness+max+critic	3,35 (±42%)	4 (±93%)	2,65 (±40%)	10 (±88%)	2,31 (±33%)	7 (±96%)	1,44 (±20%)	19 (±91%)
dynamic1	2,71 (±39%)	13 (±93%)	1,33 (±14%)	19 (±89%)	1,85 (±36%)	15 (±94%)	1,09 (±10%)	26 (±92%)
dynamic1+critic	3,01 (±40%)	14 (±92%)	2,28 (±32%)	20 (±88%)	1,89 (±35%)	17 (±97%)	1,22 (±10%)	28 (±92%)
dynamic1+max	3,22 (±43%)	18 (±105%)	1,54 (±25%)	23 (±92%)	2,32 (±39%)	21 (±108%)	1,21 (±17%)	33 (±97%)
dynamic1+max+critic	3,97 (±46%)	19 (±108%)	2,49 (±39%)	25 (±92%)	2,36 (±38%)	24 (±110%)	1,33 (±18%)	35 (±98%)
dynamic2+10	3,11 (±39%)	52 (±102%)	2,40 (±44%)	34 (±120%)	1,85 (±35%)	54 (±99%)	1,59 (±31%)	42 (±112%)
dynamic2+10+critic	3,03 (±40%)	53 (±100%)	2,97 (±41%)	35 (±117%)	1,89 (±34%)	56 (±96%)	1,66 (±29%)	45 (±110%)
dynamic2+10+max	4,53 (±49%)	56 (±92%)	3,33 (±54%)	38 (±107%)	2,33 (±39%)	58 (±90%)	1,93 (±38%)	50 (±103%)
dynamic2+10+max+critic	4,01 (±46%)	57 (±90%)	3,74 (±48%)	40 (±106%)	2,37 (±38%)	61 (±86%)	1,99 (±36%)	53 (±102%)
dynamic2+100	3,97 (±55%)	9 (±95%)	3,78 (±45%)	13 (±96%)	1,96 (±34%)	11 (±88%)	1,95 (±34%)	24 (±91%)
dynamic2+100+critic	3,27 (±42%)	10 (±89%)	3,48 (±38%)	14 (±92%)	2,00 (±33%)	13 (±85%)	2,02 (±33%)	26 (±92%)
dynamic2+100+max	5,78 (±53%)	12 (±84%)	4,67 (±51%)	17 (±87%)	2,46 (±39%)	15 (±83%)	2,09 (±33%)	29 (±94%)
dynamic2+100+max+critic	4,24 (±46%)	14 (±85%)	4,02 (±42%)	18 (±87%)	2,48 (±38%)	18 (±88%)	2,16 (±32%)	32 (±97%)
randomtask	96,50 (±289%)	3 (±127%)	1,55 (±23%)	8 (±90%)	61,60 (±297%)	6 (±103%)	1,18 (±11%)	17 (±93%)

Tableau C.8 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes de plate-forme en arbre**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,12 (±8%)	5 400 (±73%)			1,09 (±7%)	14 892 (±51%)		
min-min+critic	1,62 (±22%)	5 384 (±73%)						
sufferage	1,13 (±14%)	5 777 (±76%)			1,08 (±14%)	19 156 (±49%)		
static	1,70 (±24%)	2 (±95%)			1,29 (±20%)	3 (±85%)		
static+critic	1,87 (±26%)	3 (±74%)			1,31 (±20%)	5 (±59%)		
static+max	1,72 (±25%)	2 (±94%)			1,39 (±22%)	3 (±86%)		
static+max+critic	1,85 (±26%)	3 (±72%)			1,41 (±22%)	4 (±60%)		
static+readiness	1,64 (±25%)	2 (±93%)	1,29 (±16%)	7 (±86%)	1,26 (±21%)	3 (±85%)	1,12 (±9%)	12 (±43%)
static+readiness+critic	1,79 (±26%)	3 (±73%)	1,70 (±26%)	7 (±77%)	1,29 (±20%)	5 (±60%)	1,18 (±9%)	14 (±37%)
static+readiness+max	1,71 (±25%)	2 (±92%)	1,30 (±17%)	7 (±79%)	1,39 (±23%)	4 (±84%)	1,15 (±10%)	13 (±38%)
static+readiness+max+critic	1,84 (±26%)	3 (±71%)	1,75 (±27%)	8 (±70%)	1,41 (±23%)	5 (±59%)	1,21 (±10%)	15 (±33%)
dynamic1	2,02 (±36%)	9 (±55%)	1,22 (±13%)	13 (±57%)	1,54 (±42%)	10 (±45%)	1,09 (±7%)	18 (±36%)
dynamic1+critic	2,13 (±38%)	9 (±54%)	1,61 (±24%)	13 (±55%)	1,56 (±41%)	12 (±45%)	1,14 (±8%)	19 (±34%)
dynamic1+max	2,46 (±43%)	14 (±66%)	1,25 (±13%)	17 (±49%)	1,96 (±46%)	16 (±65%)	1,11 (±7%)	24 (±41%)
dynamic1+max+critic	2,71 (±44%)	14 (±67%)	1,71 (±26%)	17 (±49%)	1,97 (±45%)	18 (±67%)	1,17 (±9%)	26 (±41%)
dynamic2+10	2,01 (±36%)	39 (±121%)	1,69 (±31%)	17 (±43%)	1,53 (±41%)	40 (±115%)	1,34 (±29%)	25 (±23%)
dynamic2+10+critic	2,12 (±38%)	40 (±118%)	1,95 (±36%)	18 (±40%)	1,55 (±40%)	42 (±108%)	1,37 (±28%)	26 (±22%)
dynamic2+10+max	2,67 (±45%)	43 (±106%)	2,29 (±46%)	21 (±30%)	1,95 (±46%)	44 (±99%)	1,66 (±34%)	32 (±27%)
dynamic2+10+max+critic	2,71 (±44%)	44 (±103%)	2,65 (±45%)	22 (±27%)	1,96 (±45%)	47 (±92%)	1,70 (±33%)	34 (±29%)
dynamic2+100	2,09 (±41%)	7 (±104%)	1,95 (±36%)	9 (±68%)	1,54 (±40%)	8 (±80%)	1,43 (±40%)	19 (±33%)
dynamic2+100+critic	2,14 (±40%)	8 (±92%)	2,05 (±36%)	10 (±62%)	1,55 (±39%)	10 (±63%)	1,47 (±40%)	20 (±32%)
dynamic2+100+max	2,81 (±49%)	10 (±69%)	2,47 (±41%)	11 (±48%)	1,93 (±46%)	11 (±58%)	1,72 (±39%)	25 (±44%)
dynamic2+100+max+critic	2,69 (±45%)	11 (±65%)	2,64 (±40%)	13 (±47%)	1,94 (±45%)	14 (±53%)	1,76 (±38%)	26 (±47%)
randomtask	197,71 (±304%)	2 (±56%)	1,20 (±12%)	7 (±71%)	147,98 (±312%)	4 (±56%)	1,03 (±5%)	13 (±43%)

Tableau C.9 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les **graphes de plate-forme en anneau**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,14 (±8%)	5 027 (±78%)			1,07 (±6%)	15 379 (±53%)		
min-min+critic	2,27 (±30%)	5 025 (±77%)						
sufferage	1,17 (±14%)	5 432 (±81%)			1,05 (±11%)	19 361 (±46%)		
static	2,53 (±30%)	2 (±95%)			1,47 (±23%)	4 (±75%)		
static+critic	3,15 (±40%)	3 (±71%)			1,51 (±23%)	5 (±52%)		
static+max	2,68 (±38%)	2 (±93%)			1,64 (±28%)	4 (±73%)		
static+max+critic	3,16 (±44%)	3 (±66%)			1,66 (±28%)	6 (±50%)		
static+readiness	2,41 (±28%)	2 (±93%)	1,58 (±23%)	6 (±91%)	1,41 (±21%)	4 (±76%)	1,19 (±10%)	12 (±43%)
static+readiness+critic	2,95 (±35%)	3 (±72%)	2,53 (±36%)	7 (±82%)	1,45 (±21%)	5 (±52%)	1,27 (±12%)	14 (±37%)
static+readiness+max	2,66 (±38%)	2 (±88%)	1,53 (±22%)	7 (±86%)	1,63 (±28%)	4 (±70%)	1,21 (±10%)	13 (±38%)
static+readiness+max+critic	3,14 (±44%)	3 (±65%)	2,43 (±36%)	7 (±77%)	1,65 (±28%)	6 (±49%)	1,30 (±13%)	15 (±34%)
dynamic1	2,36 (±31%)	8 (±63%)	1,41 (±18%)	12 (±65%)	1,51 (±35%)	9 (±51%)	1,10 (±8%)	17 (±40%)
dynamic1+critic	2,68 (±36%)	8 (±59%)	2,26 (±32%)	12 (±61%)	1,54 (±35%)	10 (±47%)	1,17 (±10%)	18 (±37%)
dynamic1+max	2,58 (±37%)	12 (±56%)	1,45 (±19%)	15 (±48%)	1,74 (±34%)	13 (±57%)	1,18 (±11%)	23 (±40%)
dynamic1+max+critic	3,41 (±44%)	12 (±57%)	2,48 (±37%)	15 (±47%)	1,78 (±33%)	15 (±58%)	1,26 (±14%)	25 (±41%)
dynamic2+10	2,41 (±32%)	37 (±124%)	1,89 (±29%)	16 (±54%)	1,50 (±34%)	38 (±119%)	1,32 (±25%)	22 (±32%)
dynamic2+10+critic	2,68 (±36%)	38 (±121%)	2,61 (±38%)	17 (±50%)	1,53 (±33%)	40 (±114%)	1,37 (±24%)	23 (±28%)
dynamic2+10+max	3,21 (±42%)	41 (±111%)	2,28 (±33%)	21 (±70%)	1,73 (±33%)	42 (±105%)	1,41 (±19%)	29 (±57%)
dynamic2+10+max+critic	3,41 (±45%)	41 (±109%)	3,19 (±41%)	22 (±67%)	1,77 (±33%)	44 (±98%)	1,48 (±20%)	30 (±57%)
dynamic2+100	2,69 (±41%)	7 (±109%)	2,63 (±35%)	9 (±76%)	1,53 (±32%)	8 (±88%)	1,61 (±37%)	16 (±36%)
dynamic2+100+critic	2,77 (±38%)	7 (±98%)	2,94 (±36%)	9 (±70%)	1,55 (±31%)	9 (±71%)	1,63 (±37%)	17 (±32%)
dynamic2+100+max	3,73 (±48%)	9 (±75%)	3,11 (±33%)	11 (±60%)	1,75 (±34%)	10 (±61%)	1,65 (±30%)	19 (±47%)
dynamic2+100+max+critic	3,41 (±45%)	10 (±70%)	3,44 (±34%)	12 (±56%)	1,77 (±33%)	12 (±53%)	1,69 (±30%)	22 (±43%)
randomtask	139,19 (±297%)	2 (±64%)	1,37 (±19%)	7 (±76%)	72,74 (±300%)	4 (±62%)	1,09 (±7%)	13 (±39%)

Tableau C.10 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les configurations avec un **ratio des coûts de communication et de calcul de 0.1** (intensives en calcul).

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,15 (±8%)	5 040 (±80%)			1,07 (±8%)	18 074 (±110%)		
min-min+critic	2,27 (±25%)	5 052 (±80%)						
sufferage	1,20 (±14%)	5 327 (±82%)			1,08 (±10%)	21 840 (±110%)		
static	2,35 (±25%)	2 (±107%)			1,74 (±27%)	4 (±96%)		
static+critic	2,82 (±33%)	3 (±90%)			1,83 (±28%)	5 (±85%)		
static+max	2,59 (±34%)	3 (±106%)			1,98 (±31%)	4 (±95%)		
static+max+critic	2,91 (±35%)	3 (±86%)			2,06 (±31%)	5 (±80%)		
static+readiness	2,29 (±26%)	3 (±104%)	1,48 (±23%)	7 (±93%)	1,73 (±30%)	4 (±96%)	1,17 (±12%)	12 (±67%)
static+readiness+critic	2,70 (±31%)	3 (±87%)	2,32 (±39%)	7 (±86%)	1,82 (±31%)	5 (±83%)	1,29 (±14%)	14 (±66%)
static+readiness+max	2,54 (±33%)	3 (±102%)	1,52 (±24%)	7 (±88%)	1,96 (±31%)	4 (±92%)	1,20 (±14%)	13 (±66%)
static+readiness+max+critic	2,87 (±35%)	3 (±83%)	2,34 (±39%)	8 (±81%)	2,04 (±31%)	5 (±80%)	1,33 (±16%)	15 (±67%)
dynamic1	2,45 (±33%)	10 (±84%)	1,35 (±16%)	14 (±82%)	1,75 (±37%)	12 (±81%)	1,08 (±8%)	19 (±75%)
dynamic1+critic	2,96 (±41%)	11 (±83%)	2,09 (±32%)	14 (±81%)	1,78 (±36%)	13 (±82%)	1,20 (±11%)	20 (±74%)
dynamic1+max	2,63 (±36%)	14 (±93%)	1,44 (±22%)	17 (±80%)	1,96 (±38%)	16 (±93%)	1,15 (±12%)	25 (±79%)
dynamic1+max+critic	3,36 (±44%)	15 (±96%)	2,26 (±37%)	18 (±81%)	1,99 (±37%)	19 (±97%)	1,28 (±15%)	27 (±80%)
dynamic2+10	2,95 (±37%)	41 (±118%)	2,34 (±42%)	21 (±107%)	1,75 (±37%)	43 (±113%)	1,62 (±33%)	28 (±90%)
dynamic2+10+critic	2,98 (±41%)	42 (±115%)	2,95 (±42%)	22 (±103%)	1,78 (±36%)	45 (±107%)	1,67 (±31%)	30 (±87%)
dynamic2+10+max	3,85 (±51%)	45 (±107%)	3,48 (±53%)	37 (±109%)	1,97 (±38%)	46 (±102%)	1,80 (±32%)	47 (±106%)
dynamic2+10+max+critic	3,39 (±44%)	46 (±104%)	3,63 (±45%)	38 (±107%)	2,00 (±37%)	48 (±97%)	1,85 (±30%)	50 (±105%)
dynamic2+100	4,04 (±50%)	7 (±104%)	3,25 (±49%)	10 (±86%)	1,99 (±35%)	9 (±87%)	1,84 (±33%)	19 (±69%)
dynamic2+100+critic	3,35 (±40%)	8 (±93%)	3,32 (±38%)	11 (±81%)	2,00 (±35%)	11 (±76%)	1,91 (±32%)	21 (±70%)
dynamic2+100+max	4,76 (±54%)	10 (±82%)	4,89 (±52%)	16 (±93%)	2,15 (±37%)	11 (±77%)	2,11 (±28%)	28 (±105%)
dynamic2+100+max+critic	3,63 (±43%)	11 (±81%)	4,06 (±39%)	18 (±95%)	2,16 (±37%)	14 (±77%)	2,21 (±27%)	31 (±108%)
randomtask	9,61 (±104%)	2 (±95%)	1,39 (±20%)	7 (±81%)	6,03 (±120%)	4 (±88%)	1,11 (±9%)	13 (±71%)

Tableau C.11 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les configurations avec un **ratio des coûts de communication et de calcul de 1**.

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,14 (±7%)	5 249 (±76%)			1,08 (±6%)	17 993 (±109%)		
min-min+critic	2,11 (±33%)	5 247 (±76%)						
sufferage	1,17 (±15%)	5 553 (±79%)			1,05 (±10%)	22 062 (±102%)		
static	2,29 (±35%)	2 (±107%)			1,51 (±25%)	4 (±90%)		
static+critic	2,64 (±42%)	3 (±87%)			1,53 (±25%)	5 (±79%)		
static+max	2,57 (±47%)	3 (±105%)			1,79 (±37%)	4 (±89%)		
static+max+critic	2,81 (±49%)	3 (±84%)			1,80 (±37%)	6 (±78%)		
static+readiness	2,17 (±34%)	3 (±104%)	1,48 (±24%)	7 (±91%)	1,46 (±23%)	4 (±92%)	1,20 (±13%)	13 (±65%)
static+readiness+critic	2,50 (±40%)	3 (±85%)	2,31 (±40%)	8 (±84%)	1,48 (±23%)	5 (±80%)	1,29 (±16%)	15 (±63%)
static+readiness+max	2,54 (±46%)	3 (±101%)	1,50 (±25%)	7 (±87%)	1,77 (±36%)	4 (±88%)	1,24 (±15%)	14 (±63%)
static+readiness+max+critic	2,78 (±48%)	3 (±81%)	2,28 (±39%)	8 (±81%)	1,78 (±36%)	6 (±78%)	1,32 (±17%)	16 (±63%)
dynamic1	2,77 (±38%)	10 (±78%)	1,34 (±17%)	14 (±78%)	1,90 (±40%)	11 (±75%)	1,11 (±9%)	20 (±71%)
dynamic1+critic	2,89 (±36%)	10 (±77%)	2,07 (±34%)	15 (±76%)	1,93 (±39%)	13 (±76%)	1,19 (±10%)	21 (±71%)
dynamic1+max	3,06 (±41%)	14 (±84%)	1,43 (±23%)	18 (±74%)	2,29 (±42%)	16 (±87%)	1,19 (±14%)	26 (±72%)
dynamic1+max+critic	3,71 (±45%)	15 (±88%)	2,25 (±39%)	19 (±75%)	2,33 (±41%)	19 (±89%)	1,27 (±16%)	28 (±73%)
dynamic2+10	2,82 (±38%)	42 (±116%)	2,12 (±39%)	22 (±107%)	1,88 (±39%)	44 (±111%)	1,50 (±30%)	28 (±91%)
dynamic2+10+critic	2,86 (±37%)	43 (±114%)	2,61 (±40%)	22 (±104%)	1,90 (±38%)	45 (±106%)	1,56 (±29%)	30 (±89%)
dynamic2+10+max	3,84 (±49%)	46 (±103%)	3,47 (±52%)	38 (±108%)	2,27 (±42%)	48 (±98%)	2,11 (±35%)	50 (±103%)
dynamic2+10+max+critic	3,70 (±46%)	47 (±101%)	3,99 (±48%)	40 (±106%)	2,31 (±41%)	50 (±93%)	2,19 (±34%)	53 (±101%)
dynamic2+100	2,97 (±46%)	8 (±103%)	2,88 (±44%)	10 (±86%)	1,80 (±37%)	9 (±87%)	1,72 (±41%)	19 (±63%)
dynamic2+100+critic	2,83 (±39%)	8 (±94%)	2,88 (±39%)	11 (±81%)	1,83 (±37%)	11 (±75%)	1,77 (±40%)	21 (±64%)
dynamic2+100+max	4,62 (±58%)	10 (±78%)	5,04 (±48%)	16 (±86%)	2,25 (±44%)	12 (±72%)	2,26 (±32%)	28 (±92%)
dynamic2+100+max+critic	3,71 (±49%)	11 (±76%)	4,34 (±41%)	18 (±86%)	2,27 (±44%)	14 (±71%)	2,33 (±31%)	31 (±95%)
randomtask	45,15 (±175%)	2 (±91%)	1,37 (±22%)	7 (±79%)	29,54 (±188%)	5 (±83%)	1,10 (±10%)	14 (±66%)

Tableau C.12 – **Plates-formes avec 7 serveurs** : moyennes seulement sur les configurations avec un **ratio des coûts de communication et de calcul de 10** (intensives en communication).

Heuristique	Version de base		Variante *mct*		Variante *insert*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
min-min	1,13 (±8%)	5 299 (±75%)			1,07 (±6%)	17 940 (±91%)		
min-min+critic	2,05 (±35%)	5 282 (±75%)						
sufferage	1,16 (±16%)	5 601 (±78%)			1,04 (±10%)	22 049 (±84%)		
static	2,24 (±40%)	2 (±106%)			1,44 (±28%)	4 (±88%)		
static+critic	2,56 (±47%)	3 (±84%)			1,47 (±29%)	5 (±72%)		
static+max	2,53 (±52%)	3 (±105%)			1,71 (±42%)	4 (±88%)		
static+max+critic	2,72 (±54%)	3 (±82%)			1,73 (±42%)	6 (±76%)		
static+readiness	2,11 (±39%)	3 (±104%)	1,47 (±25%)	7 (±92%)	1,39 (±26%)	4 (±89%)	1,20 (±14%)	13 (±65%)
static+readiness+critic	2,41 (±45%)	3 (±84%)	2,27 (±41%)	8 (±84%)	1,41 (±26%)	5 (±73%)	1,29 (±18%)	15 (±61%)
static+readiness+max	2,52 (±52%)	3 (±100%)	1,49 (±26%)	7 (±88%)	1,70 (±41%)	4 (±85%)	1,23 (±16%)	14 (±64%)
static+readiness+max+critic	2,71 (±53%)	4 (±79%)	2,23 (±40%)	8 (±81%)	1,72 (±41%)	6 (±75%)	1,32 (±19%)	16 (±63%)
dynamic1	2,00 (±36%)	9 (±77%)	1,33 (±18%)	14 (±76%)	1,36 (±30%)	10 (±74%)	1,10 (±10%)	20 (±69%)
dynamic1+critic	2,10 (±35%)	10 (±75%)	2,04 (±35%)	15 (±74%)	1,40 (±30%)	12 (±71%)	1,18 (±11%)	21 (±68%)
dynamic1+max	2,84 (±53%)	14 (±77%)	1,42 (±24%)	18 (±67%)	2,02 (±51%)	16 (±79%)	1,18 (±15%)	26 (±66%)
dynamic1+max+critic	3,31 (±55%)	15 (±79%)	2,21 (±41%)	19 (±67%)	2,04 (±50%)	19 (±80%)	1,26 (±17%)	28 (±67%)
dynamic2+10	1,96 (±35%)	42 (±116%)	1,64 (±26%)	22 (±110%)	1,36 (±29%)	43 (±112%)	1,20 (±16%)	28 (±98%)
dynamic2+10+critic	2,11 (±36%)	43 (±114%)	2,10 (±35%)	22 (±107%)	1,40 (±29%)	45 (±109%)	1,27 (±17%)	29 (±95%)
dynamic2+10+max	3,13 (±57%)	46 (±102%)	3,01 (±56%)	39 (±106%)	2,02 (±52%)	48 (±97%)	1,86 (±44%)	52 (±100%)
dynamic2+10+max+critic	3,32 (±55%)	47 (±100%)	3,57 (±50%)	40 (±104%)	2,05 (±51%)	50 (±92%)	1,93 (±42%)	55 (±98%)
dynamic2+100	1,97 (±36%)	8 (±105%)	2,37 (±46%)	10 (±89%)	1,35 (±26%)	9 (±91%)	1,50 (±41%)	18 (±66%)
dynamic2+100+critic	2,15 (±37%)	8 (±97%)	2,37 (±42%)	11 (±84%)	1,39 (±27%)	10 (±80%)	1,53 (±39%)	20 (±64%)
dynamic2+100+max	3,40 (±61%)	10 (±74%)	4,01 (±50%)	17 (±80%)	2,01 (±53%)	12 (±67%)	1,88 (±38%)	29 (±85%)
dynamic2+100+max+critic	3,30 (±57%)	11 (±72%)	3,61 (±46%)	18 (±79%)	2,03 (±52%)	14 (±65%)	1,94 (±37%)	32 (±86%)
randomtask	335,07 (±198%)	2 (±91%)	1,36 (±23%)	7 (±80%)	222,20 (±211%)	5 (±81%)	1,09 (±10%)	14 (±63%)

Tableau C.13 – **Plates-formes avec 50 serveurs** : moyennes par types de **graphes d'application**.

Heuristique	Version de base		Variante *mct*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût
Graphes en étoile						
static+readiness	1.86 (±48%)	1 (±11%)	1.04 (±5%)	19 (±10%)	1.02 (±3%)	20 (±10%)
dynamic1	4.63 (±58%)	96 (±20%)	1.06 (±6%)	120 (±18%)	1.02 (±2%)	109 (±17%)
randomtask	5,481.95 (±192%)	1 (±2%)	1.05 (±5%)	19 (±7%)	1.02 (±3%)	20 (±8%)
Graphes deux-un						
static+readiness	5.18 (±52%)	1 (±22%)	1.94 (±28%)	8 (±10%)	1.20 (±11%)	366 (±92%)
dynamic1	4.43 (±63%)	593 (±30%)	1.77 (±32%)	603 (±30%)	1.10 (±10%)	866 (±27%)
randomtask	14.46 (±54%)	1 (±15%)	2.16 (±40%)	9 (±13%)	1.05 (±8%)	636 (±102%)
Graphes partitionnés						
static+readiness	5.72 (±44%)	1 (±11%)	2.47 (±31%)	7 (±11%)	1.26 (±13%)	1,106 (±98%)
dynamic1	4.67 (±36%)	338 (±33%)	1.97 (±28%)	345 (±32%)	1.03 (±5%)	1,102 (±65%)
randomtask	8.29 (±46%)	1 (±15%)	2.29 (±30%)	8 (±18%)	1.10 (±12%)	1,791 (±114%)
Graphes aléatoires						
static+readiness	4.73 (±48%)	2 (±38%)	2.07 (±28%)	6 (±22%)	1.23 (±15%)	1,881 (±133%)
dynamic1	5.45 (±36%)	123 (±20%)	1.68 (±19%)	126 (±20%)	1.06 (±12%)	1,412 (±115%)
randomtask	6.68 (±33%)	1 (±2%)	1.88 (±32%)	7 (±37%)	1.03 (±5%)	2,977 (±154%)
Moyennes sur tous les graphes d'application						
static+readiness	4.37 (±61%)	1 (±41%)	1.88 (±40%)	10 (±53%)	1.17 (±14%)	843 (±184%)
dynamic1	4.79 (±49%)	288 (±79%)	1.62 (±34%)	299 (±75%)	1.05 (±9%)	873 (±117%)
randomtask	1,377.84 (±419%)	1 (±12%)	1.84 (±43%)	11 (±48%)	1.05 (±8%)	1,356 (±204%)

Tableau C.14 – **Plates-formes avec 50 serveurs** : moyennes par types de **graphes de plate-forme**.

Heuristique	Version de base		Variante *mct*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût
Graphes clique-temps						
static+readiness	3.90 (±32%)	1 (±40%)	1.77 (±27%)	10 (±54%)	1.15 (±11%)	204 (±100%)
dynamic1	4.35 (±50%)	324 (±77%)	1.61 (±21%)	336 (±73%)	1.07 (±10%)	420 (±47%)
randomtask	100.33 (±263%)	1 (±8%)	1.76 (±25%)	10 (±53%)	1.01 (±2%)	223 (±90%)
Graphes clique-distance						
static+readiness	5.41 (±44%)	1 (±22%)	2.19 (±39%)	10 (±57%)	1.30 (±15%)	119 (±94%)
dynamic1	4.63 (±55%)	333 (±79%)	1.56 (±25%)	347 (±75%)	1.00 (±2%)	449 (±53%)
randomtask	204.88 (±262%)	1 (±1%)	2.06 (±34%)	10 (±58%)	1.16 (±10%)	136 (±100%)
Graphes en arbre						
static+readiness	1.89 (±34%)	1 (±53%)	1.28 (±16%)	10 (±57%)	1.08 (±7%)	732 (±79%)
dynamic1	3.87 (±45%)	301 (±69%)	1.24 (±13%)	307 (±67%)	1.06 (±6%)	795 (±65%)
randomtask	1,808.30 (±294%)	1 (±16%)	1.23 (±17%)	11 (±47%)	1.00 (±1%)	843 (±83%)
Graphes en anneau						
static+readiness	6.30 (±49%)	1 (±45%)	2.27 (±37%)	10 (±39%)	1.17 (±14%)	2,318 (±106%)
dynamic1	6.34 (±34%)	193 (±68%)	2.07 (±35%)	205 (±64%)	1.08 (±12%)	1,827 (±87%)
randomtask	3,397.87 (±290%)	1 (±13%)	2.32 (±46%)	12 (±34%)	1.03 (±5%)	4,222 (±103%)
Moyennes sur tous les graphes de plate-forme						
static+readiness	4.37 (±61%)	1 (±41%)	1.88 (±40%)	10 (±53%)	1.17 (±14%)	843 (±184%)
dynamic1	4.79 (±49%)	288 (±79%)	1.62 (±34%)	299 (±75%)	1.05 (±9%)	873 (±117%)
randomtask	1,377.84 (±419%)	1 (±12%)	1.84 (±43%)	11 (±48%)	1.05 (±8%)	1,356 (±204%)

Tableau C.15 – **Plates-formes avec 50 serveurs** : moyennes par **ratios des coûts de communication et de calcul**.

Heuristique	Version de base		Variante *mct*		Variante *mct+insert*	
	Performance	Coût	Performance	Coût	Performance	Coût
Ratio 0.1 (intensives en calcul)						
static+readiness	4.58 (±53%)	1 (±52%)	1.86 (±42%)	10 (±53%)	1.15 (±13%)	846 (±179%)
dynamic1	3.76 (±59%)	285 (±78%)	1.64 (±34%)	296 (±75%)	1.04 (±7%)	887 (±118%)
randomtask	77.39 (±236%)	1 (±11%)	1.84 (±43%)	10 (±47%)	1.05 (±7%)	1,379 (±199%)
Ratio 1						
static+readiness	4.34 (±62%)	1 (±32%)	1.90 (±39%)	10 (±53%)	1.19 (±15%)	840 (±186%)
dynamic1	5.17 (±38%)	289 (±78%)	1.62 (±33%)	301 (±75%)	1.06 (±10%)	869 (±118%)
randomtask	446.20 (±255%)	1 (±13%)	1.85 (±43%)	11 (±49%)	1.05 (±9%)	1,334 (±206%)
Ratio 10 (intensives en communication)						
static+readiness	4.20 (±68%)	1 (±32%)	1.87 (±40%)	10 (±53%)	1.19 (±15%)	843 (±186%)
dynamic1	5.46 (±46%)	289 (±79%)	1.59 (±34%)	299 (±76%)	1.05 (±10%)	861 (±115%)
randomtask	3,609.95 (±264%)	1 (±13%)	1.83 (±44%)	11 (±49%)	1.05 (±9%)	1,355 (±208%)
Moyennes sur tous les ratios						
static+readiness	4.37 (±61%)	1 (±41%)	1.88 (±40%)	10 (±53%)	1.17 (±14%)	843 (±184%)
dynamic1	4.79 (±49%)	288 (±79%)	1.62 (±34%)	299 (±75%)	1.05 (±9%)	873 (±117%)
randomtask	1,377.84 (±419%)	1 (±12%)	1.84 (±43%)	11 (±48%)	1.05 (±8%)	1,356 (±204%)

Tableau C.16 – **Plates-formes avec 100 serveurs** : toutes les heuristiques sont présentées avec leur variante *mct*, mais sans la variante *insert*.

Configuration	static		dynamic1		dynamic2+10		dynamic+100		randomtask	
	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût	Performance	Coût
Graphes d'application										
étoile	1.01 (±2%)	36 (±2%)	1.04 (±4%)	255 (±13%)	1.43 (±34%)	684 (±20%)	1.66 (±51%)	104 (±12%)	1.02 (±3%)	37 (±11%)
deux-un	1.11 (±11%)	15 (±10%)	1.01 (±3%)	1,220 (±29%)	1.26 (±14%)	949 (±27%)	2.86 (±71%)	104 (±29%)	1.28 (±17%)	15 (±9%)
partitionné	1.26 (±22%)	11 (±12%)	1.01 (±3%)	587 (±34%)	1.64 (±22%)	418 (±32%)	4.04 (±52%)	56 (±30%)	1.20 (±15%)	10 (±10%)
aléatoireé	1.37 (±19%)	8 (±20%)	1.09 (±13%)	211 (±22%)	3.33 (±31%)	148 (±19%)	7.21 (±61%)	22 (±19%)	1.16 (±14%)	8 (±32%)
Graphes de plate-forme										
clique-temps	1.11 (±7%)	18 (±57%)	1.03 (±5%)	701 (±77%)	1.98 (±52%)	616 (±57%)	4.83 (±67%)	81 (±48%)	1.14 (±8%)	17 (±62%)
clique-distance	1.44 (±22%)	18 (±59%)	1.01 (±2%)	642 (±77%)	2.27 (±57%)	601 (±61%)	6.63 (±64%)	82 (±51%)	1.29 (±15%)	16 (±68%)
arbre	1.08 (±6%)	16 (±73%)	1.04 (±5%)	568 (±74%)	1.91 (±46%)	562 (±58%)	2.40 (±53%)	71 (±50%)	1.02 (±3%)	17 (±70%)
anneau	1.14 (±19%)	17 (±64%)	1.07 (±13%)	362 (±59%)	1.50 (±41%)	419 (±64%)	1.91 (±44%)	53 (±64%)	1.21 (±20%)	21 (±65%)
Ratios des coûts de communication et de calcul										
ratio 0.1	1.16 (±18%)	17 (±63%)	1.04 (±9%)	565 (±80%)	2.17 (±51%)	549 (±62%)	4.45 (±88%)	71 (±54%)	1.15 (±14%)	17 (±65%)
ratio 1	1.21 (±21%)	17 (±63%)	1.04 (±8%)	568 (±80%)	1.90 (±50%)	548 (±62%)	4.04 (±84%)	71 (±55%)	1.17 (±17%)	18 (±68%)
ratio 10	1.20 (±22%)	17 (±63%)	1.03 (±7%)	571 (±80%)	1.68 (±56%)	552 (±61%)	3.35 (±78%)	72 (±56%)	1.17 (±18%)	18 (±68%)
Moyennes générales										
	1.19 (±21%)	17 (±63%)	1.04 (±8%)	568 (±80%)	1.92 (±53%)	550 (±62%)	3.94 (±86%)	71 (±55%)	1.16 (±16%)	18 (±67%)

Annexe D

Autre résultat de complexité

Nous avons vu dans le chapitre 4 (section 4.4.2) que, dans le cas où il y a des périodes d'indisponibilité pour les liens de communication, le problème de décision associé au problème de l'ordonnancement d'un ensemble de communications ayant une même destination est NP-complet, même si le routage est fixé.

Nous montrons ici la complexité du problème dual : dans le modèle un-port, si le routage dans le graphe de plate-forme est libre, alors le simple problème d'ordonnancer les communications est NP-difficile, même si tous les liens sont entièrement disponibles. Nous définissons formellement le problème de décision puis nous prouvons sa NP-complétude.

Définition (CommSchedFreeRoute$(\mathcal{P}, \mathcal{M}, D, T)$**).** Étant donnés un graphe de plate-forme $\mathcal{P} = (\mathcal{S}, \mathcal{L})$, un ensemble fini $\mathcal{M}$ de communications de même destination D et une borne de temps T, existe-t-il un ordonnancement valide des communications dont le temps total ne dépasse pas T ? Chaque élément de $\mathcal{M}$ est un couple (S_i, s) représentant la communication d'une donnée de taille s du serveur S_i à la destination D.

Theorème. CommSchedFreeRoute$(\mathcal{P}, \mathcal{M}, D, T)$ *est NP-complet.*

Démonstration. Il est évident que CommSchedFreeRoute$(\mathcal{P}, \mathcal{M}, D, T)$ appartient à NP. Pour prouver sa complétude, nous utilisons une réduction de 2-Partition qui est NP-complet [38, problème SP12]. Considérons une instance arbitraire $\{a_1, a_2, \ldots, a_n\}$ de 2-Partition où les a_i sont des entiers strictement positifs : existe-t-il un sous-ensemble I de $\{1, \ldots, n\}$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

Nous construisons, à partir de cette instance de 2-Partition, la plate-forme $\mathcal{P} = (\mathcal{S}, \mathcal{L})$ représentée sur la figure D.1. Il y a, dans cette plate-forme, $n + 5$ serveurs :

- $A_1, \ldots, A_n$: le serveur A_i stocke une donnée D_i de taille a_i.
- X_l et X_r : à partir de chacun des n serveurs $A_1, \ldots, A_n$, il y a un lien de bande passante 1 vers le serveur X_l, ainsi que vers le serveur X_r.

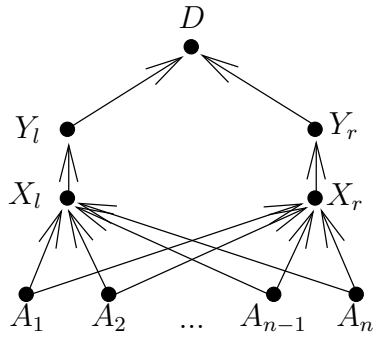


Figure D.1 – Le graphe utilisé dans la réduction de 2-Partition en CommSchedFreeRoute.

- Y_l et Y_r : il y a un lien de bande passante $1/2N$ du serveur X_l vers le serveur Y_l, et un autre de X_r vers Y_r, avec $N = \sum_{1 \leq i \leq n} a_i$.
- D : il y a un lien de bande passante 1 des serveurs Y_l et Y_r vers le serveur D.

L'ensemble des communications $\mathcal{M}$ est défini par les communication des serveurs A vers D : $\mathcal{M} = \{(A_i, a_i) \mid 1 \leq i \leq n\}$. La borne de temps est $T = N^2 + N + \min_{1 \leq i \leq n} a_i$. La taille de l'instance de CommSchedFreeRoute que nous venons de construire est clairement polynomiale en la taille de l'instance originale du problème 2-Partition.

Supposons que l'instance originale de 2-Partition admet une solution : soit $(\mathcal{I}_1, \mathcal{I}_2)$ une partition de $\{1, \ldots, n\}$ telle que $\sum_{i \in \mathcal{I}_1} a_i = \sum_{i \in \mathcal{I}_2} a_i = N/2$. Nous supposons, sans perte de généralité, que $\mathcal{I}_1$ contient un élément a_j qui est minimal : $a_j = \min_{1 \leq i \leq n} a_i$. Nous dérivons un ordonnancement pour l'instance de CommSchedFreeRoute de la manière suivante :

1. À la date 0, A_j envoie sa donnée de taille a_j à X_l. Elle est reçue à la date a_j.
2. Entre les dates 0 et $N/2$, les serveurs correspondants à $\mathcal{I}_2$ ($\{A_i \mid i \in \mathcal{I}_2\}$) envoient dans n'importe quel ordre leurs données à X_r.
3. Entre les dates a_j et $a_j + 2N \cdot a_j$, X_l envoie à Y_l la donnée reçue de A_j.
4. Entre les dates a_j et $N/2$, les serveurs $\{A_i \mid i \in \mathcal{I}_1, i \neq j\}$ envoient dans n'importe quel ordre leurs données à X_l.
5. De $a_j + 2N \cdot a_j$ à $a_j + 2N \cdot N/2$, X_l envoie dans n'importe quel ordre à Y_l les données reçues des serveurs de $\{A_i \mid i \in \mathcal{I}_1, i \neq j\}$.
6. Entre les dates $N/2$ et $N/2 + 2N \cdot N/2$, X_r envoie dans n'importe quel ordre à Y_r les données reçues des serveurs correspondant à $\mathcal{I}_2$.
7. De $a_j + 2N \cdot N/2$ à $a_j + 2N \cdot N/2 + N/2$, Y_l envoie dans n'importe quel ordre à D toutes les données qu'il détient.
8. De $a_j + 2N \cdot N/2 + N/2$ à $a_j + 2N \cdot N/2 + N$, Y_r envoie dans n'importe quel ordre à D toutes les données qu'il détient.

Ainsi, nous avons produit un ordonnancement valide correspondant à la borne de temps et donc une solution à l'instance de CommSchedFreeRoute.

Réciproquement, supposons que l'instance CommSchedFreeRoute admet une solution, c'est-à-dire un ordonnancement valide satisfaisant la borne de temps T. Alors, les communications provenant d'un A_i doivent passer soit par X_l, soit par X_r. Nous définissons $\mathcal{I}_1$ (resp. $\mathcal{I}_2$) comme l'ensemble des serveurs A_i dont les données sont envoyées à D en passant par X_l (resp. X_r). Ainsi, $\mathcal{I}_1$ et $\mathcal{I}_2$ définissent une partition de $\{1, \ldots, n\}$. Supposons que ce ne soit pas une solution à 2-Partition. Alors, sans perte de généralité,

$$\sum_{i \in \mathcal{I}_1} a_i > \sum_{i \in \mathcal{I}_2} a_i$$

et donc

$$\sum_{i \in \mathcal{I}_i} a_i \geq 1 + \frac{N}{2} > \frac{N}{2}.$$

Le temps nécessaire à X_l pour envoyer à Y_l toutes les données qu'il a reçu des A_i est alors égal à

$$2N \cdot \sum_{i \in \mathcal{I}_1} a_i \geq 2N \cdot \frac{N}{2} + 2N > \min_{1 \leq i \leq n} a_i + 2N \cdot \frac{N}{2} + N = T,$$

ce qui est absurde. □

Annexe E

Liste des publications

Revues internationales avec comité de lecture

[1] Arnaud Giersch, Yves Robert et Frédéric Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. *Journal of Systems Architecture, special issue on Parallel, Distributed and Network-based Processing: selected papers from the 12th Euromicro Conference.* À paraître.

[2] Stéphane Genaud, Arnaud Giersch et Frédéric Vivien. Load-balancing scatter operations for grid computing. *Parallel Computing*, 30(8):923–946, août 2004.

Conférences internationales avec comité de lecture

[3] Arnaud Giersch, Yves Robert et Frédéric Vivien. 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, Pise, Italie, août/septembre 2004.

[4] Arnaud Giersch, Yves Robert et Frédéric Vivien. 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, La Corogne, Espagne, février 2004.

[5] Arnaud Giersch, Yves Robert et Frédéric Vivien. Scheduling tasks sharing files on heterogeneous clusters. Dans *10th European PVM/MPI Users' Group Conference (EuroPVM/MPI 2003)*, tome 2840 de *Lecture Notes in Computer Science*, p. 657–660. Springer-Verlag, Venise, Italie, septembre 2003. Version courte de [4].

[6] Stéphane Genaud, Arnaud Giersch et Frédéric Vivien. Load-balancing scatter operations for grid computing. Dans *12th Heterogeneous Computing*

Workshop (HCW'2003), p. 101a (10 pages). IEEE Computer Society Press, Nice, France, avril 2003. Workshop tenu conjointement avec IPDPS.

[7] Romaric David, Stéphane Genaud, Arnaud Giersch, Benjamin Schwarz et Éric Violard. Source code transformations strategies to load-balance grid applications. Dans *3rd International Workshop on Grid Computing (GRID 2002)*, tome 2536 de *Lecture Notes in Computer Science*, p. 82–87. Springer-Verlag, Baltimore, MD, USA, novembre 2002.

Rapports de recherche

[8] Arnaud Giersch, Yves Robert et Frédéric Vivien. Scheduling tasks sharing files from distributed repositories (revised version). Rapport de recherche n° 2004-04, LIP, École Normale Supérieure de Lyon, France, février 2004. Également disponible comme rapport de recherche n° 5124 de l'INRIA.

[9] Arnaud Giersch, Yves Robert et Frédéric Vivien. Scheduling tasks sharing files from distributed repositories. Rapport de recherche n° 2003-49, LIP, École Normale Supérieure de Lyon, France, octobre 2003. Également disponible comme rapport de recherche n° 4976 de l'INRIA.

[10] Arnaud Giersch, Yves Robert et Frédéric Vivien. Scheduling tasks sharing files on heterogeneous clusters. Rapport de recherche n° 2003-28, LIP, École Normale Supérieure de Lyon, France, mai 2003. Également disponible comme rapport de recherche n° 4819 de l'INRIA.

[11] Stéphane Genaud, Arnaud Giersch et Frédéric Vivien. Load-balancing scatter operations for grid computing. Rapport de recherche n° 2003-17, LIP, École Normale Supérieure de Lyon, France, mars 2003. Également disponible comme rapport de recherche n° 4770 de l'INRIA.

[12] Romaric David, Stéphane Genaud, Arnaud Giersch, Benjamin Schwarz et Éric Violard. Source code transformations strategies to load-balance grid applications. Rapport de recherche n° 02-09, ICPS-LSIIT, Université Louis Pasteur, Strasbourg, France, août 2002.

Résumé

Nous étudions des stratégies d'ordonnancement et d'équilibrage de charge pour des plates-formes hétérogènes distribuées. Notre problème est d'ordonnancer un ensemble de tâches indépendantes afin d'en réduire le temps total d'exécution. Ces tâches utilisent des données d'entrée qui peuvent être partagées : chaque tâche peut utiliser plusieurs données, et chaque donnée peut être utilisée par plusieurs tâches. Les tâches ont des durées d'exécution différentes, et les données ont des tailles différentes. Toute la difficulté est de réussir à placer sur un même processeur des tâches partageant des données, tout en conservant un bon équilibrage de la charge des différents processeurs.

Notre étude comporte trois parties généralisant progressivement le problème. Nous nous limitons dans un premier temps au cas simple où il n'y a pas de partage de données, où les tailles des tâches et des données sont homogènes, et où la plate-forme est de type maître-esclave. Le partage des données est introduit dans la deuxième partie, ainsi que l'hétérogénéité pour les tailles des tâches et des données. Dans la dernière partie nous généralisons le modèle de plate-forme à un ensemble décentralisé de serveurs reliés entre eux par un réseau d'interconnexion quelconque.

La complexité théorique du problème est étudiée. Pour les cas simples, des algorithmes calculant une solution optimale sont proposés, puis validés par des résultats expérimentaux avec une application scientifique réelle. Pour les cas plus complexes, nous proposons de nouvelles heuristiques pour résoudre le problème d'ordonnancement. Ces nouvelles heuristiques, ainsi que des heuristiques classiques comme min-min et sufferage sont comparées entre elles à l'aide de nombreuses simulations. Nous montrons ainsi que nos nouvelles heuristiques réussissent à obtenir des performances aussi bonnes que les heuristiques classiques, tout en ayant une complexité algorithmique d'un ordre de grandeur plus faible.

Mots-clefs : ordonnancement, clusters hétérogènes, grilles de calcul, tâches indépendantes, données partagées, NP-complétude, heuristiques, expérimentations, simulations.

Abstract

We study scheduling and load-balancing strategies for distributed heterogeneous platforms. Our problem is to schedule a set of independent tasks in order to reduce the overall execution time. These tasks can use some input data that may be shared: each task can use several data, and each datum can be used by several tasks. Tasks have different execution durations, and data have different sizes. The difficulty is to map on a same processor tasks sharing some data, while keeping a good load-balance across the processors.

Our study comprises three parts, progressively generalizing the problem. First, we restrict ourselves to the simple case where there is no data sharing, with homogeneous sizes for tasks and data, and where the platform is a master-slave platform. Data sharing is introduced in the second part, along with heterogeneity for the tasks and data sizes. In the last part, we generalize the platform model to a decentralized set of servers, that are linked through an arbitrary interconnection network.

The theoretical complexity of the problem is studied. For simple cases, algorithms to compute an optimal solution are given and validated by experimental results with a real scientific application. For more complicated cases, we propose new heuristics to solve the scheduling problem. These new heuristics, and classic ones like min-min and sufferage are compared through extensive simulations. Thus, we show that our new heuristics perform as efficiently as the classic ones although their algorithmic complexity is an order of magnitude lower.

Keywords: scheduling, heterogeneous clusters, grid computing, independant tasks, shared data, NP-completeness, heuristics, experimentations, simulations.