Solutions parallèles efficaces sur le modèle CGM d'une classe de problèmes issus de la programmation dynamique

Vianney Kengne Tchendji

Université de Picardie Jules Verne

Université de Yaoundé I

Thèse de Doctorat

Spécialité Informatique

présentée à

L'Université de Picardie Jules Verne

Kengne Tchendji Vianney

pour obtenir le grade de Docteur de l'Université de Picardie Jules Verne

Solutions parallèles efficaces sur le modèle CGM d'une classe de problèmes issus de la programmation dynamique

Soutenue le 6 juin 2014, après avis des rapporteurs, devant le jury d'examen :

M. Gilles Dequen,	**Professeur**	**Président**
M. Christophe Cérin,	**Professeur**	**Rapporteur**
M. Eddy Caron,	**Maître de conférences, HDR**	**Rapporteur**
M. Michaël Krajecki,	**Professeur**	**Examinateur**
M. Devan Sohier,	**Maître de conférences, HDR**	**Examinateur**
M. Jean Frédéric Myoupo,	**Professeur**	**Directeur**
M^me^ Laure Pauline Fotso,	**Maître de conférences, HDR**	**Co-encadrant**

Laboratoire de Modélisation, Information et Systèmes

Dédicaces

À la mémoire de mes défunts parents

Papa TCHENDJI Raymond
et
Maman WOBINWO Jeanne, Épouse TCHENDJI

Remerciements

Le travail présenté dans ce rapport est la concrétisation des efforts de toutes sortes engagés et soutenus par bien des personnes. Sachant ne pas pouvoir trouver des mots assez justes pour leur exprimer toute ma gratitude, je voudrais ici mentionner ces personnes et le rôle qu'elles ont joué dans le cadre de ce travail. Je remercie donc très chaleureusement :

- Monsieur Jean-Fédéric MYOUPO, Professeur à l'Université de Picardie Jules Verne et Madame Laure Pauline FOTSO, Maître de Conférences à l'Université de Yaoundé I, pour avoir été plus que des directeurs de thèse ;
- Les membres du jury qui ont accepté de juger ce travail, à qui je présente mes sentiments de haute considération auxquels je joins mon plus profond respect ;
- Mes parents Raymond et Jeanne pour toute la peine qu'ils se sont donné depuis mes tous premiers jours pour assurer mon éducation ;
- Mes frères et sœurs (Gaétan, Rachel, Maurice, Mayeul, Thomas, Céline) pour leur soutien inconditionnel ;
- Les enseignants des Universités de Dschang et de Yaoundé I qui ont énormément contribué à ma formation ;
- Vous tous qui de prêt ou de loin, d'une manière ou d'une autre avez contribué à l'édification de l'être que je suis aujourd'hui ; Soyez rassuré de ma reconnaissance. Notre Père du Ciel saura seul vous combler bien au-delà de vos mérites et désirs.

Table des matières

Table des figures

Liste des Algorithmes

Résumé

Plusieurs facteurs technologiques, logiciels et économiques amènent les concepteurs d'architectures parallèles à converger vers des systèmes multi-processeurs gros-grain. Pourtant, la plupart des logiciels parallèles ont été conçu pour des systèmes parallèles à grains fins et pour des systèmes à mémoire partagée.
Dans cette thèse, nous utilisons le modèle de calcul parallèle BSP/CGM (*Bulk Synchronous Parallel / Coarse-Grained Multicomputer*), conçu dans le but de diminuer l'écart entre le logiciel et le matériel, pour apporter des solutions parallèles efficaces, à une classe de problèmes issus de la programmation dynamique. Il s'agit d'une classe de problèmes de type *polyadique non-serial* qui se caractérise par une très forte dépendance entre les calculs. Cette classe compte par exemple le problème d'ordonnancement de produit de chaine de matrices, le problème de triangulation optimale d'un polygone convexe, le problème de recherche de l'arbre binaire de recherche optimal, etc.
Pour ce faire, nous commençons par mener une étude de l'outil de conception de nos solutions, c'est-à-dire le modèle de calcul parallèle BSP/CGM. Ensuite, nous présentons quelques uns des problèmes de la classe étudiée ainsi que quelques algorithmes séquentiels permettant de les résoudre. Après, nous proposons un mécanisme de rééquilibrage de charge des processeurs pour un algorithme BSP/CGM existant. Il permet de résoudre tous les problèmes de la classe abordée. En s'inspirant de cet algorithme, nous proposons une nouvelle solution générique ayant de meilleures performances. Enfin, nous proposons deux algorithmes BSP/CGM typiques à certains problèmes de la classe. Ces algorithmes sont basés sur des solutions séquentielles dites accélérées. Ils permettent d'obtenir de meilleures performances que la première.

Mots clés : Algorithmique parallèle, Modèle de calcul parallèle, Modèle BSP, Modèle CGM, Programmation dynamique.

Abstract

Several factors lead designers of parallel architectures to converge to coarse-grained multi-processor systems. However, most parallel software has been designed for fine-grained parallel systems and for systems with shared memory.
In this thesis, we use the BSP/CGM (*Bulk Synchronous Parallel / Coarse-Grained Multicomputer*) parallel computing model, designed to close the gap between software and hardware, to provide parallels solutions to a class of dynamic programming problems. This is a *polyadique non-serial dynamic programming* class of problems, which is characterized by very high dependence of calculations. This class includes, for example, Matrix Chain Ordering Problem, Triangulation of Convexe Polygon problem and Optimal Binay Search Tree problem.
To do this, we start by carry out a detailed study of the design tool of our solutions, i.e. the BSP/CGM parallel computing model. Then, we present some of the problems of the class studied and some sequential algorithms to solve them. After that, we propose a load balancing mechanism of the processors, for an existing generic BSP/CGM algorithm which solves all the problems of the class discussed. From this algorithm, we propose a new generic solution with better performance. Finally, we propose two BSP/CGM algorithms for typical problems of the class. These algorithms are based on sequential solutions said accelerated. They perform better than the first.

Keywords : Parallel computing model, BSP model, CGM model, Dynamic programming.

Introduction

Sommaire

Calcul parallèle

Beaucoup de problèmes dans des domaines divers (recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov, etc.), expriment un besoin de haute-performance qui s'intensifie au fil du temps. Il s'agit des applications consommateurs en temps d'exécution et en espace mémoire. Parmi eux, on peut citer par exemple : la clôture transitive d'une relation, le plus court chemin dans un graphe, la grande famille des problèmes d'alignement de séquences, l'analyse syntaxique de phrases, les arbres binaires optimaux, etc. Bien que toujours en pleine évolution, les technologies de machines séquentielles n'arrivent plus à suivre le rythme. Ainsi, le parallélisme s'est présenté comme l'issue la plus naturelle pour palier à ces besoins. C'est pourquoi le développement de solutions parallèles logicielles et matérielles est devenu un axe de recherche très important de l'ingénierie informatique.
Cependant, malgré l'incapacité des plates-formes séquentielles à répondre aux besoins de puissance de calcul et de stockage des grandes compagnies, ceux-ci n'adoptent que très rarement les plates-formes parallèles. Deux raisons fondamentales justifient ce choix : le compromis Efficacité/Portabilité et l'écart logiciel-matériel.

Efficacité versus Portabilité

La plupart des solutions logicielles proposées sont soit efficaces soit portables. Ainsi, très souvent, si l'on change la machine parallèle, et que l'on souhaite optimiser les per-

formances, on doit aussi changer le logiciel parallèle. Inversement, si l'on veut changer le logiciel parallèle, le nouveau doit être conçu pour la machine parallèle existante, sinon il faudra aussi changer la machine.
La raison vient du fait que pour concevoir les algorithmes parallèles, on doit avoir un certain nombre d'informations sur la machine sur laquelle l'algorithme doit être implanté. Dans le domaine du calcul parallèle, cette description est appelée *« modèle de calcul parallèle »*. La diversité des machines parallèles à conduit à l'émergence d'une multitude de modèles de calcul parallèle.

L'inefficacité des algorithmes vient des *modèles parallèles trop abstraits* comme le modèle PRAM, qui pour davantage de facilité et de simplicité, masquent les détails de bas niveau des machines qu'ils décrivent, au point de cacher aux concepteurs quelques-unes qui sont indispensables pour la production d'algorithmes efficaces pour ces machines. Par contre, les*modèles trop réalistes* comme le modèle systolique, le modèle de grille à deux dimensions et l'hypercube, en vue d'augmenter l'efficacité des algorithmes, fournissent aux concepteurs les moindres détails (topologies d'interconnexion, techniques de routage, etc.) des machines qu'ils décrivent, au point où ces détails deviennent intrinsèques à la validité et l'efficacité des algorithmes développés. Ainsi, exécuter ces algorithmes sur d'autres machines que celles décrites par ces modèles dégrade grandement leur efficacité. Le modèle idéal est donc un modèle qui donne la même importance à l'efficacité et à la portabilité des algorithmes. Il doit abstraire les détails de bas niveau des machines parallèles (pour augmenter la portabilité), sans en abuser au point d'amener les concepteurs à fournir des algorithmes inefficaces, en leurs cachant des caractéristiques cruciales des machines. C'est pour atteindre ce but que Valiant a proposé le modèle BSP (*Bulk Synchronous Parallel model*) [Val89, Val90].

Écart entre le logiciel et le matériel

On assiste ces dernières années à une migration du matériel parallèle des systèmes à grain fin vers des systèmes multi-processeurs gros-grain [Ola08]. Ces architectures sont constituées d'un ensemble d'ordinateurs, reliés par un réseau de communication : les *« Clusters de stations de travail »* constituent un exemple typique de cette classe. En novembre 2008, 82% des 500 super-calculateurs les plus rapides du monde sont des clusters [MSDS10]. Ce qui pose problème est que le plus souvent, les logiciels parallèles sont conçus pour les machines à grain-fin et pour des machines à mémoire partagée, et donc, ces logiciels ne sont pas adaptés aux machines multi-processeurs gros-grain. Ceci est appelé dans la littérature : *l'écart (gap) entre le logiciel et le matériel* [Son01] ou encore *la barrière du logiciel* (*parallel software barrier*) [Deh06]. Dans l'éditorial consacré au calcul

scalable (pratique et expérience), du journal scientifique international du calcul parallèle et distribué, Song [Son01] conclut que : *« Dans la prochaine décennie, le défi des chercheurs en conception d'algorithmes parallèles sera de réduire cet écart ».*

Des expérimentations montrent que les modèles de calcul parallèle à gros-grain permettent de produire des algorithmes qui sont bien adaptés aux clusters [CDBL08, DFC+02, H.96]. C'est dans ce contexte que Dehné et al [DFRC93] ont proposés le modèle CGM (*Coarse-Grained Multicomputer*) qui est une version complètement gros-grain du modèle BSP [Deh99, Deh06]. Ainsi, ils contribuent à la réduction de l'écart logiciel-matériel dans le calcul parallèle. C'est pour ces raisons que de plus en plus de travaux sont menés sur la conception d'algorithmes parallèles suivant le modèle CGM.

Programmation dynamique

La programmation dynamique peut être définie comme une technique de recherche opérationnelle employée pour résoudre une large variété de problèmes d'optimisation discrets [Gen96, SL06]. C'est une méthode d'optimisation procédant par énumération implicite des solutions.
L'idée de la programmation dynamique est de subdiviser le problème à résoudre en sous-problèmes et organiser leur résolution de sorte que chacun d'eux soit évalué une seule fois. Pour ce faire, le problème est subdivisé en étapes (niveaux). Chaque étape est composée de plusieurs sous-problèmes, et a une stratégie de résolution et un certain nombre d'états associés. La solution de chacun des sous-problèmes d'une étape donnée ne dépend que de celles des sous-problèmes appartenant aux étapes précédentes. Ainsi, la complexité d'un problème de programmation dynamique dépend de l'intensité de la dépendance entre les sous-problèmes des différentes étapes.

Nous nous intéressons dans cette thèse à la conception de solutions parallèles efficaces sur le modèle CGM d'une classe de problèmes issus de la programmation dynamique.
Les problèmes de programmation dynamique dont il existe une solution suivant le modèle BSP/CGM ne sont pas très nombreux [ACD+02a, ACD02b, ACDS03, ACS05, ACS06, GMS03, KT06]. Ces solutions sont conçues pour des problèmes de programmation dynamiques *monadiques à faible dépendance*. C'est-à-dire des problèmes pour lesquelles il y a peu de dépendances entre les différents niveaux de sous-problèmes, et donc, ayant peu de contraintes de parallélisation.

La classe de problèmes qui nous intéresse est une classe de problèmes de programmation dynamique de type *polyadique non-serial* qui se caractérise par une très forte dépendance

entre les calculs. D'après M. Gengler [Gen96], *les algorithmes parallèles pour les problèmes de la classe polyadique non-serial sont plus difficiles à concevoir.*
Cette classe regroupe beaucoup de problèmes, parmi lesquels : le problème de recherche de l'arbre binaire de recherche optimal, le problème de triangulation optimale d'un polygone convexe, la grande famille des problèmes de parenthésage minimal dont le plus connu est le problème d'ordonnancement de produit de chaine de matrices, etc.

Travaux réalisés

Nous partons d'une solution BSP/CGM générique existante [Kec09], basée sur la solution générique séquentielle. Elle est conçue pour résoudre tous les problèmes de la classe abordée. Elle met plus l'accent sur la minimisation des communications par rapport à l'équilibrage de la charge de travail des processeurs. Nous proposons donc une méthode de rééquilibrage des charges de ceux-ci. Ensuite, nous proposons une nouvelle solution, qui utilise moins d'étapes, qui permet à un maximum de processeurs de participer le plus longtemps possible à la résolution du problème, et qui s'intéresse de manière égale à la minimisation des communications et à l'équilibrage des charges des processeurs.

Dans un second temps, nous nous intéressons aux solutions séquentielles dites « accélérés ». Ces solutions utilisent des spécificités propres à certains des problèmes de la classe considérée pour arriver plus rapidement à la solution recherchée. À partir de ces solutions, nous proposons deux nouveaux algorithmes BSP/CGM qui héritent de tous les avantages du précédent, mais avec de meilleures performances. Le premier entre ces deux algorithmes permet de résoudre le problème de recherche de l'arbre binaire de recherche optimal, et le second permet de résoudre les problèmes d'ordonnancement de produit de chaine de matrices et de triangulation optimale d'un polygone convexe.

Plan du document

La suite de ce document comporte cinq chapitres organisés comme suit :

Chapitre 1 : Modèles de calcul parallèle. Ce chapitre porte sur l'étude de l'outil de conception de nos solutions. Après avoir présentés les différentes caractéristiques des machines parallèles, nous présentons quelques modèles de calculs parallèles ainsi que leurs avantages et inconvénients. Nous y dégageons enfin les contraintes liées à la conception d'algorithmes parallèles.

Chapitre 2 : BSP et ses raffinements parallèles. Ce chapitre est dédié à la description du modèle BSP ainsi que ses raffinements parallèles et ses variantes. Nous présentons aussi les raisons qui nous ont poussés à opter pour l'utilisation de sa version simplifiée et complêtement gros-grain : le modèle CGM.

Chapitre 3 : Classe de problèmes étudiée. Dans ce chapitre, après avoir présenté comment les problèmes sont résolus via la technique de programmation dynamique, nous présentons la classe de problèmes de programmation dynamique dont nous voulons proposer des solutions parallèles efficaces et portables. Ensuite, nous présentons quelques problèmes appartenant à cette classe ainsi que quelques algorithmes séquentiels permettant de les résoudres. Pour l'un de ces algorithmes, nous proposons un moyen d'accroître son efficacité sans changer sa complexité.

Chapitre 4 : Algorithmes BSP/CGM génériques pour la classe de problèmes étudiée. Dans ce chapitre nous proposons tout d'abord, pour un algorithme parallèle existant, une méthode de rééquilibrage de charge de calcul des processeurs utilisés. Ensuite, nous proposons une solution parallèle générique pour tous les problèmes de la classe étudiée, qui minimise le nombre de rondes de communication et équilibre les charges de calcul des processeurs.

Chapitre 5 : Accélération en programmation dynamique. Nous proposons dans ce chapitre deux algorithmes parallèles basés sur des solutions séquentielles dites *accélérées*. Ils permettent de résoudre plus rapidement trois des problèmes de la classe étudiée. Le problème de recherche de l'arbre binaire de recherche optimal, le problème d'ordonnancement de produit de chaine de matrices et le problème de triangulation d'un polygone convexe.

Conclusions et perspectives. Nous y dressons un bilan de notre travail et présentons quelques pistes d'extensions relatives aux résultats obtenus.

Chapitre 1

Modèles de calcul parallèle

Sommaire

1.1 Introduction

Un adage ancien dit *« deux bras valent mieux qu'un seul »*. Cet adage, très utilisé dans la région de l'Ouest Cameroun, est bien mis en œuvre dans diverses situations de la vie comme lors des travaux manuels. Par exemple, dans cette région, les travailleurs se mettent le plus souvent ensemble pour effectuer les récoltes de café. Ceci vient du fait que la répartition de la tâche à accomplir entre plusieurs personnes rend le travail moins pénible et semble être un bon moyen d'aller plus vite. Une fois que les travailleurs (hommes et/ou bêtes) et le matériel (machettes, bottes, etc.) sont présents, deux tâches très importantes doivent être réalisées afin d'obtenir un maximum d'efficacité : *la subdivision du travail global à effectuer en petits travaux* et *l'organisation du matériel et des travailleurs*. Dans le cas où le travail à effectuer est la mise en œuvre d'une solution informatique (un logiciel), les travailleurs sont les composants de calcul d'un ordinateur parallèle, et le matériel, tous les outils qu'ils utilisent (mémoires, réseau d'interconnexion, etc.). L'organisation de cet ensemble est appelée *« la conception d'algorithmes parallèles »*. La performance d'une

architecture parallèle est la combinaison des performances de ses ressources et de leur agencement.

Dans ce chapitre, nous présentons plusieurs classifications d'ordinateurs parallèles, ensuite nous abordons le concept de conception d'algorithmes parallèles et les contraintes associées. Enfin nous présentons un ensemble de modèles de calcul parallèles avec leurs avantages et inconvénients.

1.2 Taxonomie des machines parallèles

L'évolution galopante de la technologie des machines séquentielles a longtemps mis sous la touche l'essor des machines parallèles. De nos jours, les architectures séquentielles sont équipées de processeurs ayant une fréquence dépassant 3 Ghz, mais malgré cela, elles ne répondent toujours pas aux besoins de haute performance d'un grand nombre d'applications. Ceci est vrai pour plusieurs raisons :

- *Le besoin des applications en puissance de traitement* : latence et débit du traitement ;
- *Le besoin toujours grandissant de mémoire* ;
- *La limite de l'approche microprocesseur* : d'une part dû à l'inadéquation du format de données et des opérations (standardisés) des microprocesseurs aux caractéristiques de certaines applications (traitement d'images, analyse numériques, etc.), et d'autre part à l'exécution séquentielle d'où découle des limites des microprocesseurs (capacités d'accès à la mémoire, tolérance aux pannes, etc.).

Il est donc nécessaire de concevoir des architectures parallèles comportant un nombre important de processeurs (jusqu'à quelques centaines pour des calculateurs à usage général et jusqu'à plusieurs millions pour des machines spécialisés) [Fos95a, Lei92].

Un ordinateur parallèle est une machine constituée d'un ensemble de processeurs capable de coopérer (effectuer des traitements, communiquer, se synchroniser, etc.) afin de résoudre plus rapidement (par rapport à un seul processeur) un problème. Cette large définition inclut les superordinateurs parallèles ayant des centaines ou des milliers de processeurs, les grappes (traduction de « clusters » en anglais), c'est-à-dire un ensemble de machines (stations de travail ou PCs) reliées par un réseau d'interconnexion, ou encore les stations de travail multiprocesseurs.

L'idée de la conception d'ordinateurs parallèles naquit depuis bien longtemps. En effet, dès la naissance des ordinateurs, Von Neumann dans les années 1940, proposa le concept d'automate cellulaire comme un modèle de calcul [Asp90]. C'est un modèle massivement

parallèle[1]. Mais à l'époque, il n'y avait ni la technologie, ni le savoir-faire suffisant pour réaliser ce parallélisme et pour l'exploiter. Il fallait donc trouver plus simple, et Von Newman a proposé l'architecture séquentielle (par opposition à « parallèle ») qui est toujours à la base des architectures actuelles. C'est une décennie plus tard que l'on commence effectivement à s'intéresser au traitement parallèle pour améliorer les calculs : Zuse en 1956, Holland en 1959 qui proposa « les machines capables d'exécuter simultanément un nombre arbitraire de sous-programmes » et Conway en 1963 qui décrit la conception d'un ordinateur parallèle et sa programmation. En 1962 la compagnie Burroughs a annoncé la « D825 », la première machine parallèle (équipée de 4 processeurs). En 1967 la machine « ILLIAC IV » (équipé de 64 processeurs), souvent considérée comme « le premier Supercalculateur » a vu le jour. Depuis lors, le parallélisme n'a cessé d'évoluer (en logiciels et matériels). En effet, d'après H. Meuer et al [MSDS10], en novembre 2012, Le CRAY XK7 : 17.6 Pflops avec 8.2 Mwatt (encore dénommée Titan), classé au premier rang du top500 des machines parallèles, est composé de 18.688 nœuds de calculs (299.008 CPU cores + 18.688 GPU K20, 60.640 cores pour exécuter le benchmark) et une mémoire de 700 téraoctets.

Selon la structure des machines parallèles, différentes classifications sont possibles. Ces classifications permettent de mettre en évidence les différents aspects qui distinguent les machines parallèles et leurs éventuelles conséquences sur le logiciel. Nous présentons ici les plus importantes.

1.2.1 Classification de Flyn

La classification de Flynn [Fly72] est sans doute la plus connu. Elle est fondée sur deux critères :

1. la machine a-t-elle un ou plusieurs flux de données ? (*Single Data stream* ou *Multiple Data stream*) ;
2. la machine a-t-elle un ou plusieurs flux d'instructions ? (*Single Instruction stream* ou *Multiple Instruction stream*).

Ainsi, en fonction du nombre de flux de données et du nombre de programmes utilisé dans les machines parallèles, Flynn les répartit en quatre classes, comme illustré dans le tableau 1.1.

Voici une interprétation des différents sigles de ce tableau :

1. Modèle dont les composants de traitement sont étroitement reliés par un réseau rapide.

Flux	Flux de données simples	Flux de données multiples
Flux d'instructions simples	SISD (Von Neumann)	SIMD
Flux d'instructions multiples	MISD (pipeline)	MIMD

TABLE 1.1 – Classification de Flynn

SISD (*Single Instruction stream, Single Data stream*) : Il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme, tant au niveau des instructions qu'au niveau de la mémoire (figure 1.1). Cette catégorie correspond à l'architecture de Von Neumann ;

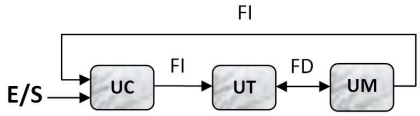
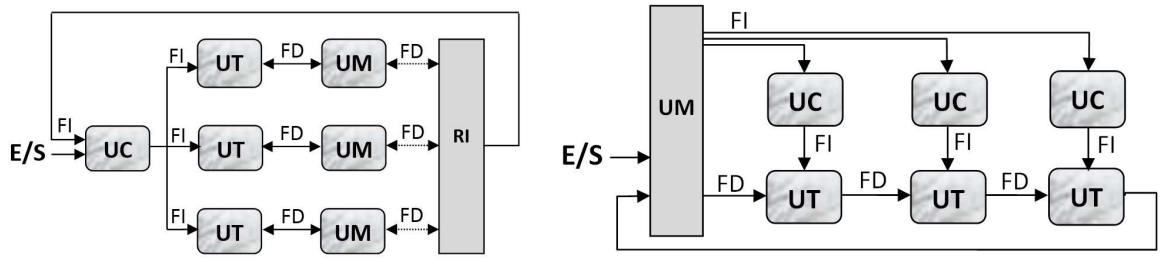


FIGURE 1.1 – Architecture SISD. L'Unité de Contrôle (UC), recevant son Flot d'Instructions (FI) de l'Unité Mémoire (UM), envoie les instructions à l'Unité de Traitement (UT), qui effectue ses opérations sur le Flot de Données (FD) provenant de l'unité mémoire.

SIMD (*Single Instruction stream, Multiple Data stream*) : Dans cette classe d'architectures, les processeurs sont fortement synchronisés, et exécutent au même instant la même instruction, chacun sur des données différentes (figure 1.2a). Ces machines sont adaptées aux traitements réguliers, comme le calcul matriciel sur matrices pleines ou le traitement d'images. Elles perdent en revanche toute efficacité lorsque les traitements à effectuer sont irréguliers et dépendent fortement des données locales, car dans ce cas, les processeurs sont inactifs la majorité du temps. Cette catégorie correspond à l'architecture des processeurs vectoriels ;

MISD (*Multiple Instructions stream, Single Data stream*) : Il s'agit d'un ordinateur dans lequel une même donnée est traitée par plusieurs processeurs en parallèle. Il existe peu d'implémentations en pratique. Cette catégorie peut être utilisée dans le filtrage numérique et la vérification de redondance dans les systèmes critiques. Elle correspond aux calculateurs systoliques et les machines utilisant le mode pipeline (figure 1.2b) ;

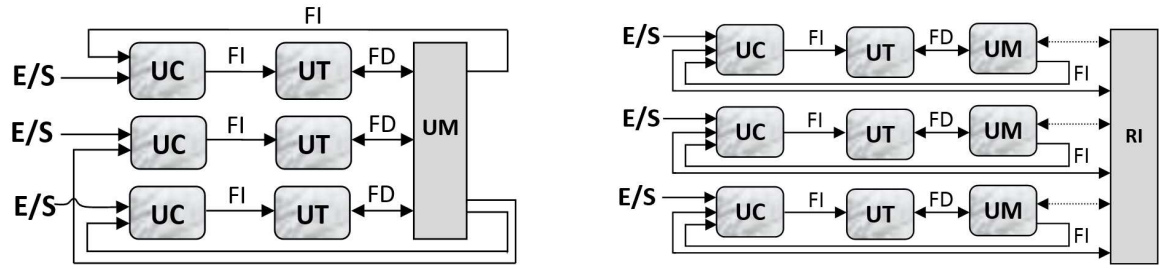
MIMD (*Multiple Instructions stream, Multiple Data stream*) : Cette classe comprend les machines multiprocesseurs, où chaque processeur exécute son propre code de manière asynchrone et indépendante. On distingue habituellement deux sous-classes (figure 1.3a et figure 1.3b), selon que les processeurs de la machine ont accès à une mémoire commune (on parle alors de MIMD à mémoire partagée ou



(a) Architecture SIMD. (b) Architecture MISD

FIGURE 1.2 – Architectures SIMD et MISD, (RI *désigne le réseau d'interconnexion*).

« multiprocessor »), ou disposent chacun d'une mémoire propre (MIMD à mémoire distribuée ou « multicomputer »). Dans ce dernier cas, un réseau d'interconnexion est nécessaire pour échanger les informations entre les processeurs. Il existe aussi les architectures SPMD (*Single Program Multiple Data*) dans le cas où le même programme est exécuté sur tous les processeurs et MPMD (*Multiple Program Multiple Data*) dans le cas contraire.



(a) Mémoire partagée (b) Mémoire distribuée

FIGURE 1.3 – Architecture MIMD

1.2.2 Classification de Raina

Une sous-classification étendue des machines MIMD, due à Raina [Rai92], et illustrée en figure 1.5, permet de prendre en compte de manière fine les architectures mémoire selon deux critères :

L'organisation de l'espace d'adressage

- **Machine à mémoire partagée ou SASM** (*Single Address space, Shared Memory*) : Les processeurs ont accès à la mémoire comme un espace d'adressage global. Tout changement dans une case mémoire est vu par les autres processeurs.

La communication et la synchronisation inter-processeurs sont effectuées via la mémoire globale (variables partagées) ;

- **Machine à mémoire distribuée ou DADM** (*Distributed Address space, Distributed Memory*) : Chaque processeur a sa propre mémoire et son propre système d'exploitation. L'échange de données entre processeurs s'effectue nécessairement par passage de messages. Ce second cas de figure nécessite un middleware pour la synchronisation et la communication ;
- **Architecture hybride ou SADM** (*Single Address space, Distributed Memory*) : C'est l'architecture la plus utilisée par les superordinateurs. Elle combine les caractéristiques des deux classes précédentes. La mémoire est physiquement distribuée sur les processeurs, cependant l'utilisateur à l'impression d'avoir un seul espace d'adressage global. Les systèmes hybrides possèdent l'avantage d'être très extensibles, performants et à faible coût.

Le type d'accès mémoire mis en œuvre

- **NORMA** (*No Remote Memory Access*) : il n'y a pas de moyen d'accès aux données distantes. La communication inter-processeurs se fait par passage de messages ;
- **UMA** (*Uniform Memory Access*) : les processeurs accèdent à la mémoire de manière symétrique. Le coût d'accès à la mémoire est identique pour tous les processeurs ;
- **NUMA** (*Non-Uniform Memory Access*) : les performances d'accès dépendent de la localisation des données. La majorité des machines parallèles appartiennent à cette classe car l'accès en temps constant à la mémoire est une contrainte technologique difficile à réaliser ;
- **CC-NUMA** (*Cache-Coherent NUMA*) : type d'architecture NUMA intégrant des caches ;
- **COMA** (*Cache Only Memory Access*) : les mémoires locales se comportent comme des caches, de telle sorte qu'une donnée n'a pas de processeur propriétaire ni d'emplacement déterminé en mémoire ;
- **OSMA** (*Operating System Memory Access*) : les accès aux données distantes sont gérés par le système d'exploitation, qui traite les défauts de page au niveau logiciel et gère les requêtes d'envoi/copie de pages distantes.

La classification de Raina est résumé sur la figure 1.4. Des exemples de machines existantes sont proposés pour chaque classe de machine.

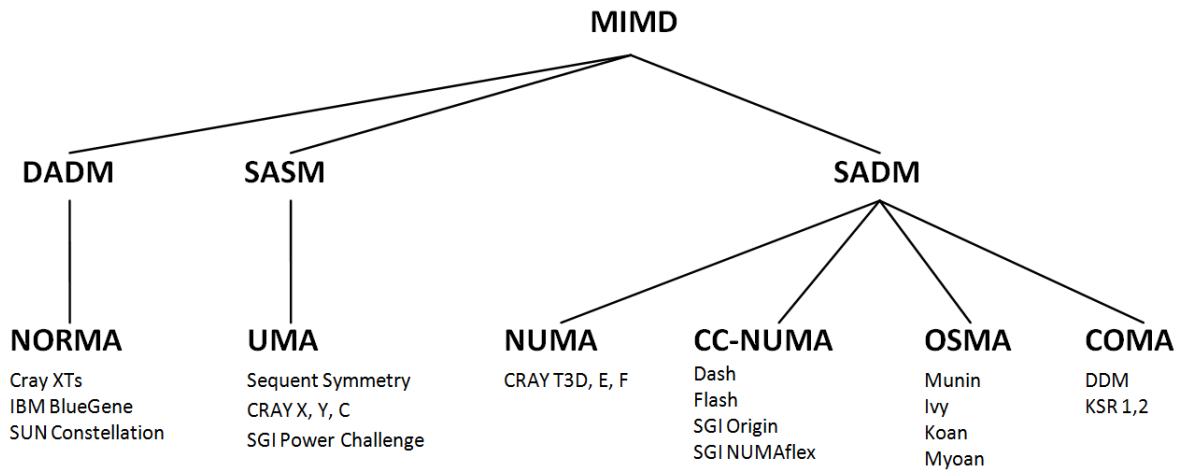
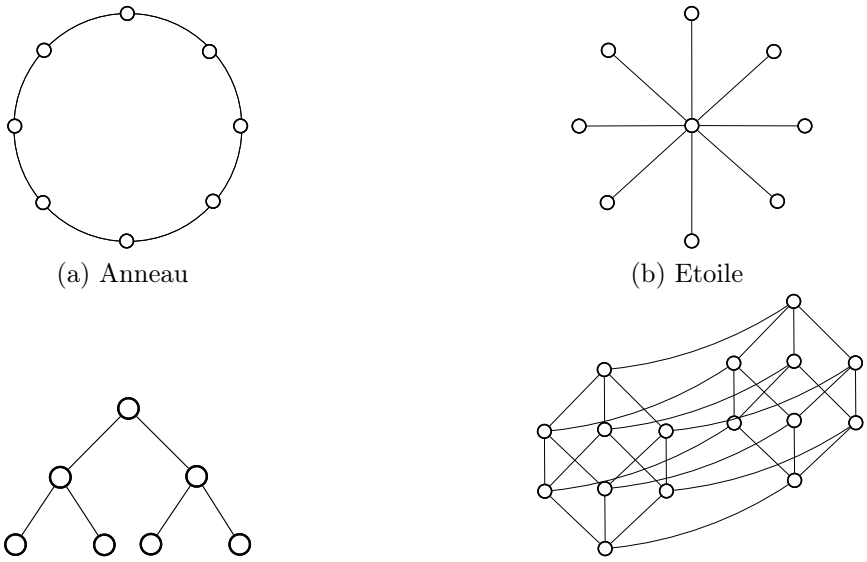


FIGURE 1.4 – Classification MIMD de Raina

1.2.3 Classification suivant la topologie du réseau d'interconnexion

Ce qui distingue les machines parallèles dans cette classification est la topologie du réseau d'interconnexion utilisés pour relier les différents nœuds de calculs avec les unités de mémoires. Cette topologie est souvent prise en compte dans la conception d'algorithmes parallèles car elle peut diriger la tâche de conception ou bien influer sur le temps d'exécution globale des programmes. Elle est donc cruciale dans le choix du type d'algorithmes convenables pour chaque type de machine parallèle [Was94]. On distingue principalement deux classes de réseaux d'interconnexion :

- **Les réseaux à topologie statique :** Ce sont des réseaux dont la topologie est définie une fois pour toute lors de la construction de la machine parallèle. Cette topologie représente un type de graphe qui caractérise le réseau. Les réseaux statiques sont utilisés essentiellement dans les machines à passage de messages, les machines cellulaires, etc. Dans ces machines, on demande moins de performances au réseau d'interconnexion. Ce sont des machines à processeurs faiblement couplés. Par exemple, dans la topologie en anneau (figure 1.5a), un nœud n'est relié qu'à deux voisins, et dans l'hypercube à quatre dimensions (figure 1.5d), chacun est relié à quatre voisins. Ainsi, si un nœud veut communiquer avec un nœud non-voisin, il doit obligatoirement passer par tous les nœuds les séparant suivant un chemin donné.
- **Les réseaux à topologies dynamiques :** Ce sont des réseaux dont la topologie (topologie logique ou comportement du réseau) peut varier (via des « switches ») au cours de l'exécution d'un programme parallèle ou entre deux exécutions de pro-

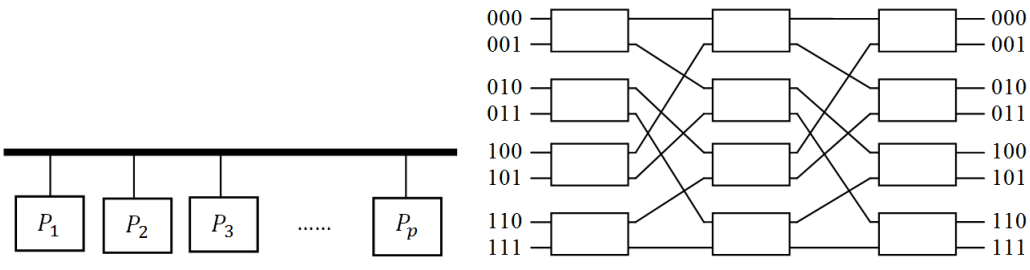


(a) Anneau (b) Etoile

(c) Arbre (d) 4-D Hypercube

FIGURE 1.5 – Quelques topologies de réseaux d'interconnexion statiques

grammes. Les réseaux dynamiques sont souvent utilisés dans les multiprocesseurs à mémoire partagée. Dans ce cas, le réseau d'interconnexion relie les processeurs aux bancs de la mémoire centrale partagée. Ce sont des machines à processeurs fortement couplés. Dans ces machines, le réseau doit être extrêmement performant pour ne pas trop ralentir les instructions machines les plus sensibles. Généralement, les réseaux dynamiques se distinguent les uns des autres par leurs stratégies de Switch et leur nombre de niveaux de switch. La figure 1.6a présente le cas le plus simple, à un seul niveau de switch (un BUS) et la figure 1.6b représente une topologie dynamique à trois niveaux de switch (le réseau Oméga).



(a) Bus (b) 8×8 Omega network

FIGURE 1.6 – Quelques topologies de réseaux d'interconnexion dynamiques

1.2.4 Classification suivant la granularité de calcul

La taille du grain d'une machine parallèle peut se définir par la longueur typique d'un morceau de code exécutable sans interruption par l'unité de calcul [Was94]. Une interruption peut être externe, provoquée par exemple par l'ordonnanceur du système d'exploitation, ou bien interne, provoquée par le processus en cour lui-même, suite à un besoin de communication avec d'autres processus (ou processeurs). Beaucoup de machines parallèles ne permettent que des interruptions internes. Selon la taille du grain de calcul, on rencontre trois classes de machines parallèles :

- **Les machines à grains fins :** Pour ces modèles, on suppose que le nombre de processeurs est sensiblement égal au nombre de données en entrée. Les processeurs exécutent de très petits morceaux de code entre deux communications successives. Le coût des communications est négligé au profit des calculs. Cette classe correspond aux machines systoliques ;
- **Les machines à gros grains :** Pour ces modèles, la taille de chaque mémoire locale est beaucoup plus grande que la taille d'une donnée. Et donc, chaque processeur est capable d'exécuter sans interruption des procédures voir des programmes entiers. Ces modèles reconnaissent et prennent en compte le coût des communications sans pour autant spécifier la topologie du support de communication. Les clusters de stations de travail sont un exemple typique de cette classe ;
- **Les machines à grains moyens :** Ce sont les machines qui sont entre les deux classes précédentes.

1.2.5 Classification suivant la connectivité des unités de calcul

La classification peut être aussi faite selon la vitesse du réseau d'interconnexion. Dans ce cas, deux classes se distinguent :

- **Les machines à processeurs fortement couplés :** il s'agit des machines dont les processeurs sont étroitement connectés par un réseau rapide. Ce sont par exemple les machines massivement parallèles telle que la T3E de SCI/CRAY ;
- **Les machines à processeurs faiblement couplés :** Ce sont des machines dont les processeurs sont reliés par un réseau plutôt lent. C'est le cas des clusters de stations de travail qui utilisent des réseaux locaux standard [2].

Les machines parallèles sont aussi classifiées selon leurs performances, selon leur nombre de processeurs, selon leur taille mémoire et selon leur extensibilité (possibilité d'augmen-

2. Pas ceux utilisant les LAN rapides tel que Gigabit Ethernet.

tation du nombre de processeurs sans perte significative de performance). Pour plus de détails sur les types de machines parallèles et leurs classifications, voir [Was94, GGKK03].

La connaissance des ordinateurs parallèles ne suffit pas pour obtenir un bon algorithme parallèle, il faut en plus une bonne gestion de tout le matériel informatique utilisé.

1.3 Conception d'algorithmes parallèles

1.3.1 Source du parallélisme

La mise en œuvre des architectures parallèles repose sur deux courants de pensées : celui issu de l'étude des systèmes concurrents et celui issu de l'étude de l'algorithmique parallèle. Les systèmes concurrents se sont attaqués surtout à l'exploitation du *parallélisme de contrôle* qui met l'accent sur les actions à réaliser, et l'algorithmique parallèle s'est attaqué surtout à l'exploitation du *parallélisme de données* qui met l'accent sur l'organisation des données. Ces deux écoles sur lesquelles découlent la classification de Flyn ont donc développé séparément des modèles de calcul et des langages concurrents ou parallèles. Ainsi, il existe deux sources (formes) de parallélisme : une issue des données (parallélisme de données) et une autre issue de la structure de l'application (parallélisme de contrôle) [CS99].

- **Parallélisme de données :** Il résulte du fait qu'une même opération peut se réaliser simultanément par plusieurs processeurs sur des données différentes ;
- **Parallélisme de contrôle** : Il résulte du fait que différentes opérations, indépendantes les unes des autres, sont réalisables simultanément. Ce type de parallélisme ne provient pas des données mais plutôt de la structure de la solution et de la dépendance entre les différentes opérations qui la composent.

1.3.2 Calcul de la complexité parallèle

Pour étudier les performances d'un algorithme parallèle, plusieurs mesures sont utilisées. La plus naturelle est le *temps d'exécution* de l'algorithme. Il correspond au temps écoulé entre le moment où le premier processeur commence l'exécution de l'algorithme et le moment où le dernier processeur finit cette exécution. Dans la suite de cette section, le temps d'exécution de l'algorithme parallèle sur p processeurs sera noté T_p. Ce temps peut être aussi défini comme étant la somme du temps de calculs, du temps de communication et du temps d'attente d'un processeur arbitraire j $\left(T_p = T^j_{calculs} + T^j_{coms} + T^j_{attente}\right)$,

ou comme étant la somme de ces trois temps sur tous les processeurs divisé par le nombre de processeurs $\left(T_p = \frac{1}{p}\left(\sum_{i=1}^{p} T^i_{calculs} + \sum_{i=1}^{p} T^i_{coms} + \sum_{i=1}^{p} T^i_{attente}\right)\right)$.

Une manière simple d'évaluer les communications est de donner le nombre de messages transmis et leur taille en octets. Ainsi, envoyer un message ayant k octets prendra un temps $T_{mesg} = T^j_{init} + k \times T^j_{trans}$, où T^j_{init} est le *temps d'initialisation* de la communication, et T^j_{trans} est le *temps pour transférer* un octet entre deux processeurs. La réalité est souvent plus complexe car il faut prendre en compte la nature du réseau. La *latence* indique le temps nécessaire pour communiquer un message unitaire [3] entre deux processeurs. Tout envoi d'un message prendra donc un temps supérieur à celui de la latence.

Le *facteur d'accélération* (« speedup ») correspond au rapport entre le temps d'exécution de l'algorithme séquentiel optimal T_s et le temps d'exécution de l'algorithme parallèle sur p processeurs : $A_p = \frac{T_s}{T_p}$. Aussi, p est le meilleur facteur d'accélération que l'on puisse obtenir.

L'efficacité est le facteur d'accélération divisé par le nombre de processeurs : $E_p = \frac{A_p}{p}$. Il représente la fraction de temps où les processeurs travaillent. Il caractérise aussi l'efficacité avec laquelle un algorithme utilise les ressources de calcul d'une machine.

On s'intéresse aussi à *l'extensibilité* (« scalability ») lorsqu'on regarde l'évolution du temps d'exécution et du facteur d'accélération en fonction du nombre de processeurs.

1.3.3 Modèles de calcul parallèle : définitions et terminologies

Une solution (algorithme) parallèle efficace d'un problème ne peut pas émerger uniquement à partir de la maîtrise des sources du parallélisme. Il faut en plus considérer un modèle. Ce dernier permet de formaliser en quelques paramètres le cadre de la ou des machines sur lesquelles l'algorithme devrait être implanté. Le modèle énonce aussi les règles de fonctionnement du programme informatique, règles résultant de l'implantation de l'algorithme.

Une fois ces caractéristiques données, le modèle permet d'analyser l'algorithme, c'est-à-dire de déterminer par exemple son temps d'exécution ainsi que le nombre et la taille des autres ressources occupées. Ces analyses sont précieuses car si le modèle est assez réaliste, elles indiquent si le programme associé à l'algorithme donnera ou non de bonnes performances. Elles permettent aussi de comparer entre eux des algorithmes résolvant le même problème et donc de déterminer suivant un critère bien précis (comme le temps d'exécution par exemple) quel est le meilleur algorithme obtenu pour un problème donné.

3. C'est le message de plus petite taille du réseau qui n'est pas nécessairement un octet.

On appelle *modèle de calcul parallèle*[4] *« la description d'une machine parallèle destinée à être utilisée par les concepteurs d'algorithmes parallèles »*. Le choix d'un modèle est très souvent guidé par son utilisation sous-jacente dans le but d'obtenir des résultats concluent sur une ou un ensemble de machines données.

Un bon modèle de calcul parallèle doit avoir un certain nombre de propriétés dont les plus importantes sont les suivantes [Fos95b, JáJ92] :

- *La simplicité de compréhension* : c'est-à-dire ne pas être ambigu et avoir un minimum de paramètres ;
- *La simplicité d'utilisation* : c'est-à-dire qu'il ne doit pas augmenter la complexité du problème à résoudre en rappelant par exemple les petits détails inhérents à l'architecture des machines ;
- *La bonne définition* : c'est-à-dire qu'il doit être bien et complètement défini pour pouvoir servir de plate-forme commune pour différents développeurs ;
- *La généralité* : il doit refléter les caractéristiques de la majorité des architectures existantes ;
- *La prédiction (représentation) des performances* : les prédictions théoriques des performances des algorithmes doivent être en adéquation avec leur exécution sur les machines réelles. Autrement dit, les bons algorithmes pratiques ne doivent résulter que de bons algorithmes théoriques [Sni89] ;
- *La faisabilité (réalité)* : La machine abstraite décrite dans un bon modèle doit être réalisable par les technologies en cours sans la violation d'un grand nombre des hypothèses du modèle ;
- *La robustesse dans le temps* : un bon modèle doit survivre à l'évolution des technologies utilisées par les machines parallèles réelles qu'il décrit ;
- *La portabilité* : les algorithmes conçus suivant le modèle ne doivent pas être dépendant des architectures des machines.

Il est malheureusement difficile de concevoir un modèle répondant aux critères sus exprimés. Le modèle séquentiel de Von Neumann satisfait en grande partie ces différentes caractéristiques, c'est d'ailleurs pour cette raison qu'il est utilisé sur la plupart des machines actuelles. Il modélise correctement les machines, ce qui permet de réaliser de bonnes prédictions, mais sous une restriction : *il suppose la mémoire des machines infinie*, ce qui n'est pas le cas en pratique. Lorsque la mémoire est saturée, les performances se dégradent

4. Dans la littérature on utilise différents termes : modèle de calcul parallèle, machine abstraite parallèle, modèle parallèle [JáJ92], modèle de programmation parallèle [Fos95b], modèle de machine parallèle [GGKK03].

très fortement et les prédictions effectuées ne sont plus valables. Von Neumann avait de même mis en avant la hiérarchie mémoire d'un grand nombre de plates-formes actuelles qui va des caches aux bandes en passant par la mémoire et les disques, tout en soulevant les difficultés pour proposer un modèle prenant en compte cette hiérarchie mémoire. C'est néanmoins le modèle séquentiel le plus employé et probablement le plus performant.

Dans le but de palier aux différentes insuffisances liées au modèle séquentiel, des modèles parallèles ont vu le jour mais aucun d'eux ne s'est imposé jusqu'ici de la même façon que le modèle séquentiel de Von Neumann. Beaucoup de modèles proposés sont loin de répondre aux différentes caractéristiques énoncées ci-dessus.

1.4 Quelques modèles de calcul parallèle

Nous présenterons dans cette section différentes grandes classes de modèles de calcul parallèle. L'objectif n'est pas de décrire tous les modèles existant (la liste est très longue), mais de présenter les concepts sous-jacents à chaque grand groupe de modèles. Pour une introduction aux différents modèles de ces grandes classes, voir [MA97]. Nous présenterons quelques modèles à grain fin parmi lesquels le modèle PRAM (*Parallel Random Access Memory*), le modèle systolique, le modèle de grille à deux dimensions et l'hypercube. Un chapitre sera consacré à la description des modèles plus récents à gros grain. Typiquement, il s'agira du modèle BSP (*Bulk Synchronous Parallel*) et ses raffinements parallèles.

1.4.1 Modèle PRAM

Le modèle PRAM (*Parallel Random Access Memory*) est apparu le premier à la fin des années 70. Il comprend un ensemble de processeurs possédant chacun une mémoire locale de petite taille (égale à $O(1)$), et une mémoire partagée par laquelle les processeurs peuvent communiquer. Les processeurs travaillent de manière synchrone, et chacun peut accéder à n'importe quelle place de la mémoire partagée en temps constant afin de lire ou d'écrire une donnée.

Le modèle PRAM est simple et est très utile pour le parallélisme des problèmes étudiés. Il constitue souvent une première étape pour la parallélisation. En vertu de son haut niveau d'abstraction, il permet souvent de savoir si un problème peut être parallélisé ou non et dans quel mesure. La description des algorithmes est très simple, car il suffit de décrire une séquence d'opérations parallèles exécutées par les processus sans se préoccuper des communications entre les processeurs. La plupart des questions théoriques s'expliquent

naturellement dans ce modèle.

l'inconvénient de ce modèle est qu'il est fortement éloigné des machines réelles. La plupart des machines sont à mémoire distribuée et non à mémoire partagée et les contraintes technologiques pour que beaucoup de processeurs puissent accéder en temps constant à une mémoire commune sont telles qu'aucune machine PRAM n'a encore vu le jour. Par conséquent il faut souvent réadapter l'algorithme PRAM à la structure de la machine choisie. Ce modèle reste néanmoins d'actualité et un certain nombre d'extensions ont été proposées [Val90, Gib89].

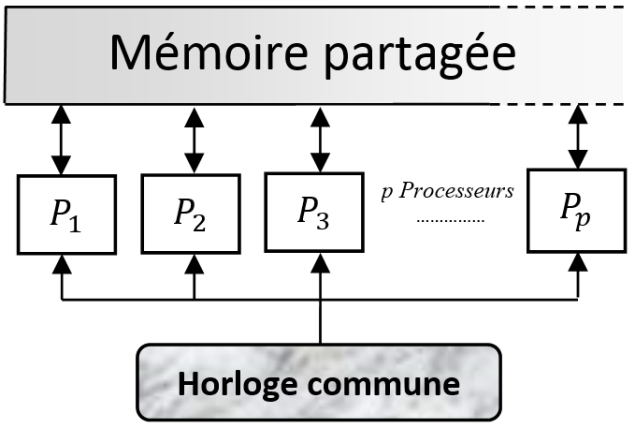


FIGURE 1.7 – Modèle PRAM

1.4.2 Modèle systolique

Le concept d'automate cellulaire est étroitement lié au concept de « réseaux systolique » [Fru92]. On peut définir un réseau systolique comme un réseau de processeurs qui calculent et échangent des données régulièrement. L'analogie est souvent faite avec la régularité de la contraction cardiaque qui propage le sang dans le système circulatoire du corps. Chaque processeur d'un réseau systolique peut être vu comme un cœur jouant le rôle de pompe sur plusieurs flots le traversant. Le rythme régulier de ces processeurs maintient un flot de données constant à travers tout le réseau.

Une donnée introduite une seule fois dans le réseau est propagée d'un processeur à un processeur voisin et peut ainsi être utilisé un grand nombre de fois. Cette propriété autorise de gros débits de calculs même si la cadence des entrées-sorties reste faible. En d'autres termes, on évite les engorgements des buffers d'entrées-sorties. Ceci rend le modèle systolique adéquat pour beaucoup de problèmes. Ceux dont le nombre de calculs sur une même donnée est largement supérieur à son nombre d'entrées-sorties.

Les caractéristiques dominantes d'un réseau systolique peuvent être définies par un parallélisme massif et décentralisé, par des communications locales et régulières, et par un mode opératoire synchrone. Pour décrire un réseau systolique, il est donc nécessaire de spécifier :

- La topologie du réseau d'interconnexion des processeurs ;
- L'architecture d'un processeur (description des registres et canaux : nom, type, sémantique, etc.) ;
- Le programme d'un processeur ;
- Le flot de données consommées par le réseau pour produire une solution.

1.4.3 Modèle grille à deux dimensions et modèle hypercube

Parmi les modèles de machines à mémoires distribués, *la grille à deux dimensions* et *l'hypercube* ont été beaucoup utilisés [Fer96, Lei92]. Dans ces modèles chaque processeur a sa propre mémoire locale de taille constante, il n'existe pas de mémoire partagée. Les processeurs peuvent communiquer uniquement grâce à un réseau d'interconnexion. Comme dans le cas de PRAM et du modèle systolique, les processeurs travaillent de manière synchrone. À chaque étape, chaque processeur peut envoyer un mot de données à un de ses voisins, recevoir un mot de données d'un de ses voisins, et effectuer un traitement local sur ces données. La complexité d'un algorithme est définie comme étant le nombre d'étapes de son exécution.

Ces modèles prennent explicitement en compte la topologie du réseau d'interconnexion. Ce dernier présente différentes caractéristiques importante comme :

- *Le degré*, qui est le nombre maximal de voisins d'un processeur, il correspond à une sorte de limitation architecturale donnée par le nombre maximum de liens physiques associés à chaque processeur ;
- *Le diamètre*, qui est la distance maximale entre deux processeurs (cette distance est donnée par le plus court chemin dans le réseau d'interconnexion entre les deux processeurs les plus éloignés). Il donne une borne inférieure sur la complexité des algorithmes dans lesquels deux processeurs arbitraires doivent communiquer ;
- *La largeur*, qui est le nombre minimum de liens à enlever afin de diviser le réseau en deux réseaux de même taille (plus ou moins un). Elle présente une borne inférieure sur le temps d'exécution des algorithmes où il existe une phase qui fait communiquer une moitié des processeurs avec une autre moitié.

La grille à deux dimensions

La grille à deux dimensions de taille p est composée de p processeurs $P_{i,j}, 1 \leq i,j \leq \sqrt{p}$, tels que le processeur $P_{i,j}$ est relié aux processeurs $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$ et $P_{i,j+1}$ pour $2 \leq i,j \leq \sqrt{p}-1$. La grille a un degré de 4, un diamètre de $O(\sqrt{p})$ et une largeur de bande de $\sqrt{p}$. Cette structure ainsi que ses dérivés, comme la grille à trois dimensions, le tore à 2 ou 3 dimensions (les processeurs à la périphérie sont connectés avec ceux se trouvant à l'opposé) ont l'avantage d'être simple et flexible. La figure 1.8 présente une grille à deux dimensions de taille 16.

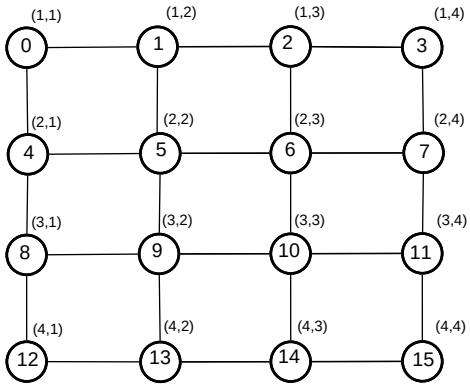


FIGURE 1.8 – Une grille 4×4

L'hypercube

L'hypercube de dimension d est composé de $p = 2d$ processeurs, numérotés de 0 à $p-1$. Deux processeurs P_i et P_j sont reliés si et seulement si la représentation binaire de i et celle de j diffèrent seulement d'un bit. Si ce bit est égal à k, on dit que P_i et P_j sont voisins suivant la dimension k et que $j = i^k$. Le degré et le diamètre sont égaux à $\log p$, et la largeur de bande est égale à $p/2$. L'hypercube est attrayant de part sa régularité et son petit diamètre, mais est difficilement réalisable en pratique (à cause du degré qui augmente avec le nombre de processeurs). La figure 1.9 présente des hypercubes de dimensions $2, 4, 8, 16$ et 32.

Pour écrire un algorithme dans un de ces modèles à mémoire distribuée, il est possible, soit de simuler un algorithme PRAM, soit d'écrire un algorithme spécifique pour l'architecture du modèle choisi. La première solution bien qu'élégante conduit souvent à des algorithmes non efficaces. En revanche, la deuxième solution permet d'obtenir des résultats efficaces mais reste très souvent compliquée. De plus, la plupart du temps, elle ne permet pas d'écrire des algorithmes portables, puisque chaque modèle à mémoire distribuée dépend fortement de la topologie du réseau choisi. Par conséquent, lorsqu'on change la topologie du réseau d'interconnexion, il faut souvent changer d'algorithme.

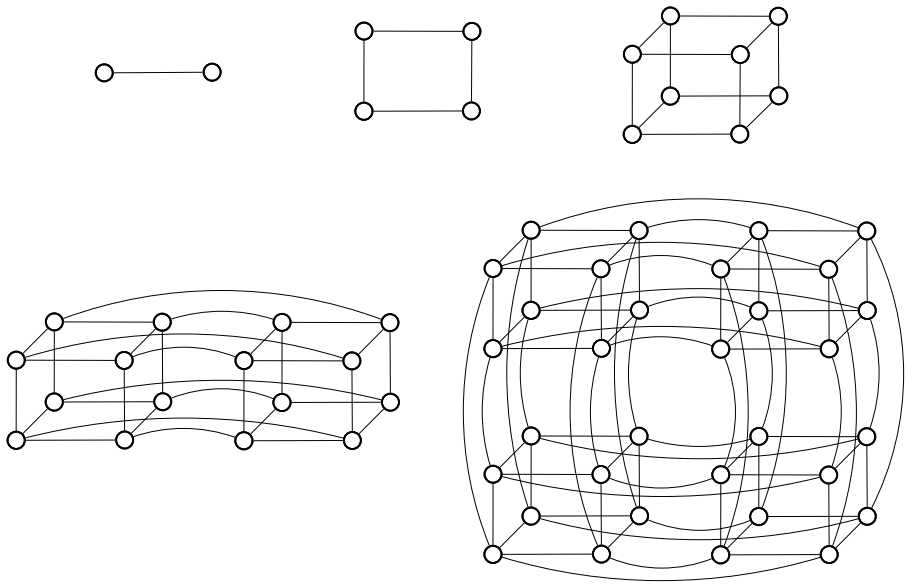


FIGURE 1.9 – Hypercubes pour $p \in \{2, 4, 8, 16, 32\}$

La mise en œuvre d'algorithmes parallèles avec les modèles parallèles existants à montré un certain nombre de compromis [Ski02, Sni89] :

- *Efficacité* versus *Portabilité* des algorithmes : modèle hypercube - modèle textscpram ;
- *Généralité* versus *Réalité* : modèle systolique - modèle PRAM ;
- *Simplicité de programmation* versus *Efficacité d'exécution* : modèle à mémoire partagée -modèle à mémoire distribuée ;
- *Simplicité d'analyse* versus *Simplicité de réalisation* : modèle synchrone - modèle asynchrone.

1.5 Compromis Efficacité - Portabilité

L'un des compromis les plus importants est celui de l'*Efficacité - Portabilité*. En effet, à la fin les années 1980, les observateurs de l'évolution du calcul parallèle se posaient la question suivante : *pourquoi malgré l'incapacité des plates-formes séquentielles à répondre aux besoins de puissance de calcul et de stockage des grandes compagnies, celles-ci n'adoptent que très rarement les plates-formes parallèles ?*

L'une des sources principales de ce problème provient d'une part de l'*inefficacité* d'une grande partie des algorithmes parallèles proposés, ceux conçus suivant des *modèles trop abstraits* comme le modèle PRAM. Et d'autres part de la *non-portabilité* de l'autre partie,

c'est-à-dire ceux conçus suivant des *modèles trop réalistes* comme le modèle systolique, le modèle de grille à deux dimensions et l'hypercube. Ainsi, les modèles de calcul parallèle utilisés à cette époque étaient soit trop abstraits, soit trop réalistes.

Modèle trop abstrait : Nous appelons modèle trop abstrait tout modèle qui, pour davantage de facilité et de simplicité, masque les détails de bas niveau des machines qu'il décrit, au point de cacher aux concepteurs quelques-unes qui sont indispensables pour la production d'algorithmes efficaces pour ces machines. Le modèle trop abstrait le plus répandu est le modèle PRAM [5] (voir section 1.4.1). Dans ce modèle, à chaque unité de temps, un processeur peut faire, soit une opération lecture/écriture sur la mémoire partagée, soit une opération de calcul interne. Ainsi, on donne le même coût (une unité de temps) à une opération de calcul et une opération de communication. Cette abstraction conduit les concepteurs à développer des algorithmes sans aucune contrainte sur le nombre de communications utilisées. Ainsi, elle leur cache le fait que dans les machines réelles, une communication est beaucoup plus coûteuse qu'une opération de calcul et donc beaucoup plus influente sur le temps d'exécution des algorithmes [Akl90, Sny86].

Modèle trop réaliste : Inversement, nous appelons modèle trop réaliste tout modèle qui, en quête d'augmenter l'efficacité des algorithmes, fournit aux concepteurs les moindres détails (topologies d'interconnexion, techniques de routage, etc.) des machines qu'il décrit, au point où ces détails deviennent intrinsèques à la validité et l'efficacité des algorithmes développés. Ainsi, exécuter ces algorithmes sur d'autres machines que celles décrites par ce modèle dégradera grandement leur efficacité. Le modèle hypercube (voir section 1.4.3) fait partie de cette classe.

1.6 Synthèse

Plusieurs critères permettent de classifier les machines parallèle, et la diversité des architectures de machines parallèle a conduit à l'émergence de plusieurs modèles de calcul parallèle. Mais contrairement au modèle séquentiel de Von Neumann, l'unification de ces modèles dans le but de faciliter la tâche aux concepteurs d'algorithmes parallèle est une tâche qui semble très ardu, à causes des différents compromis qui doivent être pris en compte. Comme nous l'avons vu, l'un des plus importants (parmi ces compromis) est

5. Même la simulation du modèle PRAM dans les machines à mémoire distribuée est mauvaise dès que le gap (temps entre deux réceptions ou deux transmissions successives) est grand, c'est le cas dans la majorité des machines parallèles actuelles (en occurrence les clusters de PC).

celui de *l'Efficacité - Portabilité*. La solution la plus intuitive à ce problème (compromis) est de s'intéresser de manière égale à l'efficacité et à la portabilité des algorithmes parallèles à produire. Ceci requiert un modèle centre qui abstrait les détails de bas niveau des machines parallèles (pour augmenter la portabilité), sans en abuser au point d'amener les concepteurs à fournir des algorithmes inefficaces, en leurs cachant des caractéristiques cruciales des machines. C'est dans cette optique que Valiant a proposé le modèle BSP (*Bulk Synchronous Parallel model*) [Val89, Val90] dont la présentation succincte fait l'objet du chapitre suivant. Les algorithmes proposés dans cette thèse reposent sur l'un des raffinements de ce modèle, en l'occurrence, le modèle CGM (*Coarse-Grained Multicomputer*) [DFRC93].

Chapitre 2

BSP et ses raffinements parallèles

Sommaire

2.1 Introduction

Le modèle de Von Neumann utilisé pour le développement des solutions séquentielles permet d'éviter le compromis entre la portabilité et l'efficacité des algorithmes. Ceci provient du fait qu'il joue le rôle de pont (« bridge ») entre les développeurs d'algorithmes et les concepteurs de machines. En effet, ce modèle a donnée à chacune des industries, logicielle et matérielle, ce dont elle avait besoin pour se mouvoir [McC96]. Pour le calcul parallèle, Valiant ([Val89, Val90]) s'est inspiré de cette caractéristique pour mettre au point *un modèle centre* qui devait offrir un consensus entre : *les concepteurs d'architectures parallèles* dont la tâche sera d'exploiter au maximum les innovations technologiques pour la fabrication des machines compatibles avec le modèle et *les concepteurs d'algorithmes parallèles* qui devront centrer leur énergie à mettre au point des solutions (logiciels) toujours plus efficaces et compatibles avec la machine abstraite décrite par le modèle. Ils s'affranchissent de ce fait, pendant la phase de conception, de l'épineuse question de la portabilité des algorithmes.

Dans ce chapitre nous présentons le modèle BSP (*Bulk Synchronous Parallel model*), ainsi que quelques uns de ses raffinements et variantes : le raffinement de McColl [McC95b], le raffinement de miller [Mil94], le raffinement de Kechid-Myoupo [Kec09] et le modèle CGM (*Coarse-Grained Multicomputer*) [DFRC93]. Tout le travail de cette thèse est basé sur le modèle CGM. Nous motivons aussi ce choix dans ce chapitre.

2.2 Modèle BSP

Depuis plusieurs décennies, on observe que l'évolution des super-ordinateurs se dirige vers des architectures parallèles (multi-processeurs), que les stations de travail multiprocesseurs se banalisent et que les performances des réseaux locaux s'améliorent. Plusieurs facteurs technologiques, logiciels et économiques amènent les concepteurs d'architectures parallèles à converger vers des systèmes constitués d'un ensemble d'ordinateurs complets (unité centrale, mémoire vive importante, cache, disque, etc.) reliés par un réseau de communication. Cette convergence suggère de modéliser le nombre de processeurs, leurs performances et les paramètres numériques des communications, plutôt que la topologie du réseau d'interconnexion ou le routage des messages, qui sont des facteurs trop variables. Ces considérations et les résultats théoriques sur le routage des messages obtenus pendant les années 1980 ont permis à L. Valiant [Val90] de décrire une machine abstraite capable de généraliser la majorité des machines parallèle existante : *la machine abstraite BSP*.

McColl [McC96] préconisait en 1996 que : *« Il y a actuellement un consensus croissant sur le fait que, pour une multitude de raisons, il va y avoir une évolution stable vers un modèle d'architecture standard pour le calcul parallèle extensible. Il va très probablement, consister en une collection de couples processeur-mémoire connectés par un réseau d'interconnexion, qui pourra être utilisé pour supporter efficacement un espace d'adressage globale. Les deux types de programmation parallèle, à mémoire partagée et à mémoire distribuée, seront donc efficacement supportés dans ces architectures »*. L'évolution (la convergence) des machines parallèles pendant les deux dernières décennies lui donne bien raison. Le modèle BSP permet dans de nombreux cas de prévoir les performances exactes d'un algorithme avant de l'implanter ou de le porter (sur une autre machine).

2.2.1 Machine abstraite BSP

Le modèle BSP [Val90] décrit la machine abstraite BSP comme un ensemble de trois composants :

- Un ensemble de nœuds de calcul (processeurs) dotés ou non de la fonction de stockage ;
- Un réseau d'interconnexion capable de transmettre *de manière point à point* des messages entre n'importe quel couple de nœuds ;
- Un mécanisme de synchronisation globale capable de synchroniser tout ou une partie des nœuds de calcul à des intervalles de temps réguliers L. La valeur de L peut être déterminée par programme.

Un algorithme BSP est constitué d'une succession de *super-étapes* qu'on nomme aussi *s-étape*. Lors d'une super-étape, chaque processeur peut réaliser des calculs locaux et effectuer un certain nombre de communications (composées d'envois et de réceptions de messages). Deux super-étapes consécutives sont séparées par une barrière de synchronisation. Cette synchronisation est décrite comme étant une opération périodique. Après chaque L unités de temps, une vérification est effectuée. Si la super-étape en cours est finie, la super-étape suivante peut être entamée. Si non, une autre vérification est faite après L unités de temps. Les calculs réalisés dans une super-étape sont asynchrones. Ainsi, le mécanisme de synchronisation est utilisé uniquement pour assurer une synchronisation des processeurs au début de chaque super-étape. La super-étape $i+1$ est entamée uniquement lorsque tous les processeurs de la super-étape i ont fini leurs tâches. Empêchant un processeur de passer à la super-étape $i+1$ tant qu'un autre est encore dans la super-étape i. La succession des super-étapes est illustrée dans la figure 2.1.

Cette machine abstraite permet de concevoir des algorithmes portables.

2.2.2 Portabilité de la machine abstraite BSP

Comme présenté dans la section 1.3.3, un bon modèle de calcul parallèle doit satisfaire le critère de portabilité. Il doit refléter le plus grand nombre d'architectures de machines possible de telle sorte que celles-ci ne constituent que des instances de la machine abstraite qu'il définit [UD97]. La machine abstraite BSP satisfait pleinement cette propriété.

En effet, l'abstraction faite sur les caractéristiques internes du réseau d'interconnexion (topologie, algorithme de routage, etc.) et sur la relation des mémoires avec les processeurs lui permet d'englober toutes les machines à mémoire partagée et toutes les machines à mémoire distribuée. La longueur contrôlable de l'intervalle entre deux synchronisations lui permet d'englober les machines SIMD et MIMD. Le calcul à l'intérieur d'une super-étape étant effectué de manière asynchrone fait de la machine abstraite BSP un compromis entre les machines synchrones et les machines asynchrones. Ainsi, un algorithme développé

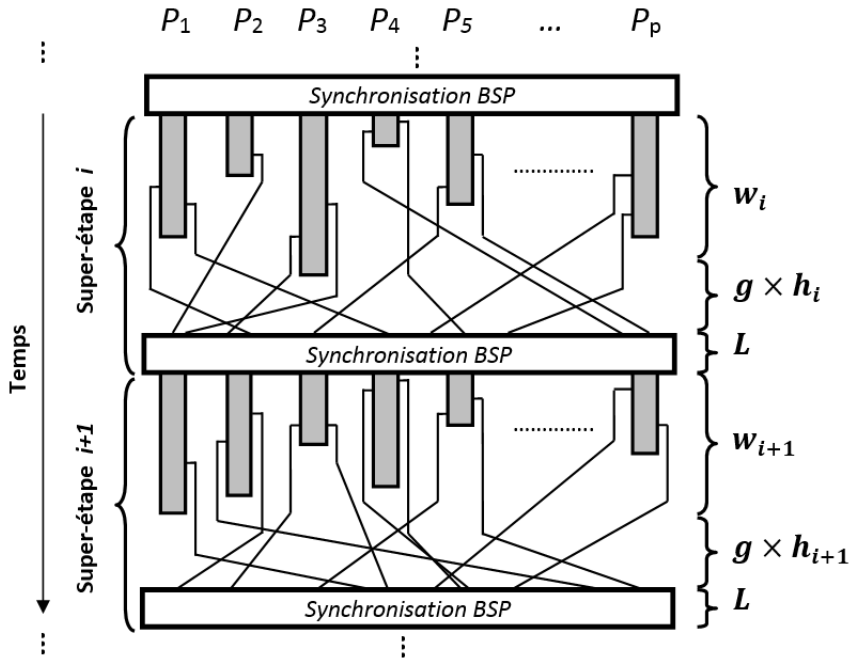


FIGURE 2.1 – Calcul BSP organisé en super-étapes

pour la machine abstraite BSP ne souffrira pas du problème d'incompatibilité ou de non portabilité sur une grande gamme des machines parallèles réelles.

2.2.3 Modèle de coût BSP

À coté de la description de la machine abstraite BSP, le modèle BSP décrit les performances d'un algorithme parallèle à l'aide de trois paramètres p, L et g de la manière suivante :

- p est le nombre de nœuds de calcul (processeurs) ;
- g est le temps d'envoie ou de réception d'un mot (message) à travers le réseau. Ce temps est la somme du temps d'initialisation de la communication et du temps de transfert proprement dit. Ces deux opérations sont séparées par certains auteurs ;
- L est le temps de synchronisation qui correspond à L unités de temps nécessaires pour synchroniser tous les processeurs.

Les paramètres L et g sont définis dans la même unité de temps appelée *« étape de temps »*, qui correspond au temps d'exécution d'une opération de calcul élémentaire par la machine BSP. Ce choix permet la comparaison des différentes machines BSP en termes de temps de livraison, tout en prenant en compte leurs vitesses de calcul. En effet, si deux machines BSP permettent la livraison du même nombre de mots (messages) par

seconde, leur comparaison à travers leurs paramètres g mesurés en seconde indique qu'elles possèdent la même qualité de livraison. Pourtant, la meilleure machine est celle qui a la plus petite vitesse de calcul puisqu'elle a un rapport *vitesse de calcul / vitesse de livraison* plus petit. Ainsi, prendre l'étape de temps, comme unité de mesure, ne cache pas de telles informations. Afin de normaliser le paramètre g avec cette unité de mesure pour toute machine parallèle, il est calculé par l'expression suivante :

$$g = \frac{\text{La vitesse de calcul globale du système [opération de calcul/seconde]}}{\text{La vitesse de livraison globale du système [mot/seconde]}}$$

Les paramètres du modèle BSP guident l'analyse et la prédiction des coûts d'exécution des algorithmes BSP. En effet, ce modèle évalue un algorithme à l'aide d'une expression arithmétique écrite à base des paramètres L et g. Dans la littérature, on appelle cette expression *« modèle de coût BSP »*. Le coût d'un algorithme BSP est l'addition des coûts de toutes les super-étapes qui le constitue. Le coût d'une super-étape dépend du temps de synchronisation, du temps de calcul et du temps de communication de chaque processeur de la machine BSP. Les temps de calcul et de communication sont réalisés comme suit :

- *Coût des calculs d'une super-étape* : il est noté w et est égale à $Max_{1\leq i\leq p}(w_i)$, où w_i est le nombre d'opérations de calcul effectuées par le processeur i durant la super-étape ;
- *Coût de communication d'une super-étape* : toutes les communications effectuées durant une super-étape sont considérées ensemble (en groupe) dans ce qu'on appelle une *h-relation*. Une h-relation est un schéma de communication globale défini sur un ensemble de processeurs où chacun peut envoyer ou recevoir au maximum h mots (la taille de sa h-relation). Ainsi, le calcul du coût de communication d'une super-étape se fait à travers le calcul de la valeur de h. Cette valeur est égale à $Max_{1\leq i\leq p}(h_i)$, où h_i est le nombre de messages (mots) envoyés ou reçus par le processeur i durant la super-étape.

  La description de la machine abstraite BSP ne spécifie pas si les envoies et les réceptions de messages d'un processeur peuvent s'effectuer en même temps ou pas. On peut donc calculer h_i de deux façons différentes : Il sera égale à $Max\{h - send_i, h - receive_i\}$ [1] si la simultanéité est possible, et à $h - send_i + h - receive_i$ dans le cas contraire. Une fois h calculé, le coût de communication de la super-étape est : $g \times h$.

1. $h - send_i$ est le nombre de mots envoyés par le processeur i durant une super-étape et $h - receive_i$ le nombre de mots qu'il reçoit.

Puisque, d'une part, la description de la machine BSP ne spécifie pas si les processeurs peuvent effectuer les opérations de calcul et les opérations de communication simultanément ou pas, et d'autre part, la valeur du paramètre L est parfois définie par la période entre deux synchronisations successives ([Val90, GV94]) et parfois par le temps d'exécution d'une opération de synchronisation ([McC94, McC95b, ST98, Ski02]) ; on peut exprimer le coût d'une super-étape de quatre façons différentes :

- $w + g \times h + L$ si les communications s'effectuent après les calculs et L est le temps d'exécution d'une opération de synchronisation ;
- $\max\{w + g \times h, L\}$ si les communications s'effectuent après les calculs et L est la période entre deux synchronisations successives ;
- $\max\{w, g \times h\} + L$ si les communications et les calculs peuvent s'effectuer en même temps, et L est le temps d'exécution d'une opération de synchronisation ;
- $\max\{w, g \times h, L\}$ si les communications et les calculs peuvent s'effectuer en même temps et L est la période entre deux synchronisations successives.

En définitive, si on appelle t_i le coût d'exécution d'une super-étape i, alors le coût de l'algorithme BSP ayant s super-étapes sera égale à : $\sum_{i=1}^{s} t_i$.

2.2.4 Efficacité des algorithmes BSP

L'efficacité des algorithmes BSP résulte principalement de :

L'exactitude du modèle de coût BSP : Elle est étudiée dans la section 2.3.

L'orientation correcte à la conception : Contrairement au modèle PRAM qui laisse comprendre que le coût d'une opération de communication est égal à celui d'une opération de calcul, le modèle de coût BSP, dont les paramètres sont déduits à partir de benchmarks réels, fournit des directives très correctes qui mènent les concepteurs à produire des algorithmes plus efficaces sur les architectures parallèles réelles. Ces directives leur apprend que :

- le coût d'un algorithme BSP provient du coût de ses opérations de calcul, de communication et de synchronisation ;
- le coût d'une opération de communication est plus grand que celui d'une opération de calcul ($g > 1$ dans toutes les machines parallèles réelles) [BSP13b, Hil98] ;
- le coût d'une opération de synchronisation est beaucoup plus grand que celui d'une opération de communication ($L > g$ dans toutes les machines parallèles réelles) [BSP13b, Hil98].

Les concepteurs d'algorithmes BSP déduisent qu'il faut suivre dans l'ordre prioritaire sous-indiqué les directives suivantes :

1. Minimiser le nombre d'opérations de synchronisation d'un algorithme (c'est-à-dire le nombre de super-étapes) ;
2. Minimiser le nombre d'opérations de communication ;
3. Minimiser le nombre d'opérations de calcul.

Cet ordre va des opérations les plus couteuses vers les opérations les moins couteuses.

2.2.5 Quelques implémentations BSP

Nous avons montré qu'en vertu de sa généralité, la machine abstraite BSP permet de développer des algorithmes portables (voir section 2.2.2). En effet, cette machine abstraite est réalisable à travers les technologies existantes. Puisque toute machine parallèle est constituée d'un ensemble de processeurs capable de communiquer (à l'aide de routeurs par exemple) et de se synchroniser (par une implantation logicielle des barrières de synchronisation par exemple) alors on peut dire que : *Toute machine parallèle est implicitement une machine BSP.*

La réalisation de machines BSP se fait à travers l'implantation d'un *système d'exécution parallèle* sur les différentes machines parallèles existantes [UD97] (Voir figure 2.2). Le rôle d'un système d'exécution consiste en l'exploitation des ressources de la machine cible afin d'offrir les routines requises pour permettre explicitement un calcul conforme au modèle BSP sur cette machine. Ces systèmes d'exécution parallèle sont appelés dans la littérature *« les Bibliothèques BSP »*. Ces Bibliothèques offrent principalement des fonctions pour la création et la destruction de processus, des fonctions pour l'accès mémoire distant, l'envoi (passage) et la réception de messages et des fonctions pour la synchronisation (comme les barrières de synchronisation). Les plus connues parmi ces bibliothèques sont :

- The OXFORD BSP Library [Mil93] ;
- The Green BSP Library [GLRT95] ;
- The OXFORD BSP toolset [HMS+98] ;
- The Paderborn University BSP-Library (PUB) [BJVOR03] ;
- The Paderborn University BSP-WEB-Library (PUB-WEB) [PW013].

La bibliothèque MPI (*Message Passing Interface*) est souvent utilisée pour la réalisation individuelle (par le programmeur) des opérations normalement offertes par les biblio-

thèques BSP [Bis04]. Il existe aussi plusieurs implémentations de MPI (comme OpenMPI, MPICH, etc.)

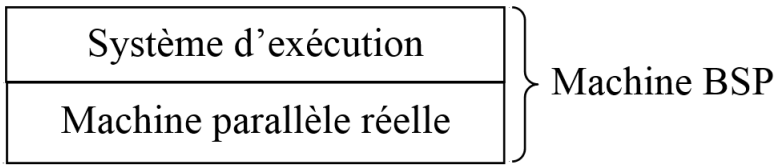


FIGURE 2.2 – Implémentation BSP

L'apparition du modèle BSP a provoqué deux axes de recherche qui sont en pleine effervescence jusqu'à ce jour [BSP13a, Deh06, PW013] : Le développement d'algorithmes BSP et l'amélioration du modèle. En effet, différents raffinements du modèle BSP, et plusieurs variantes sont apparus [McC95b, Kec09, DFRC93].

2.3 Quelques raffinements du modèle BSP

2.3.1 Modèle BSP développé de McColl

Le raffinement de McColl [McC95b] est une spécialisation opéré à deux niveaux :

Développement au niveau de la fréquence des synchronisations : Tandis que dans le modèle original une tentative de synchronisation est effectuée chaque L unité de temps, dans le modèle développé de McColl, il n'y a qu'un seul appel à la synchronisation par super-étape. Cet appel a lieu à la fin de la super-étape. Il est initialisé par le premier processeur qui termine son travail (calcul et communication) inhérent à la super-étape. Ainsi, dans ce raffinement, le paramètre L reflète la durée de la synchronisation.

Développement de la sémantique des opérations de communication : Il part du fait que le modèle BSP original n'interdit pas que les informations échangées dans une super-étape soient visibles et ainsi exploitables par les processeurs récepteurs avant la fin de la super-étape. Ceci permet un chevauchement entre les opérations de calcul et les opérations de communication, et peut grandement compliquer l'analyse et la prédiction des coûts d'exécution des algorithmes. McColl simplifie la situation en *interdisant explicitement* une telle tolérance. *Les lectures distantes n'auront d'effet (ne seront visibles) qu'à la fin de la super-étape.*

Le raffinement de McColl permet d'optimiser les performances des algorithmes BSP à plusieurs niveaux :

- *Le paquetage de messages* : toutes les informations devant aller d'un processeur à un autre dans une super-étape peuvent être rassemblées dans le même paquet (message) à la fin de la super-étape. De ce fait, le cout d'initialisation de la communication entre une paire de processeurs est grandement diminué. Cette initialisation sera effectuée une seule fois pour chaque paire de processeurs, au lieu d'une fois pour chacun des messages qu'ils se communiquent ;
- *Le réordonnancement des envois* : le retardement des opérations de communication permet de mieux apprécier la quantité d'informations qui arrive ou part d'un processeur pendant la phase de communication. Ainsi, il permet de mettre en place un schéma de communication adéquat, qui minimise les congestions au niveau des processeurs et qui permet une meilleure distribution des données.

Le retardement des opérations de communication à la fin d'une super-étape (à la fin des calculs) sous la forme d'une communication globale, fait du modèle BSP un *modèle de programmation parallèle* [Kec09]. Un programme BSP consiste ainsi en une suite de super-étapes, chacune constituée de trois phases successives (figure 2.3) : la phase de calcul, la phase de communication globale et la phase de synchronisation ([McC94, McC95a]). Cette augmentation lui a apporté plusieurs avantages, notamment sur l'efficacité des algorithmes. Il permet par exemple à un concepteur d'algorithmes BSP de :

- avoir une idée claire et unique sur la *forme future* de l'implantation réelle de son algorithme ;
- avoir un modèle de coût unique et correcte. C'est le modèle de coût appelé dans la littérature *modèle de coût standard* [HMS97, Ski02]. Dans le modèle de coût standard, le coût d'exécution d'une super-étape est prédit par l'expression $w + g \times h + L$, très compatible avec la forme successive des phases d'une super-étape.

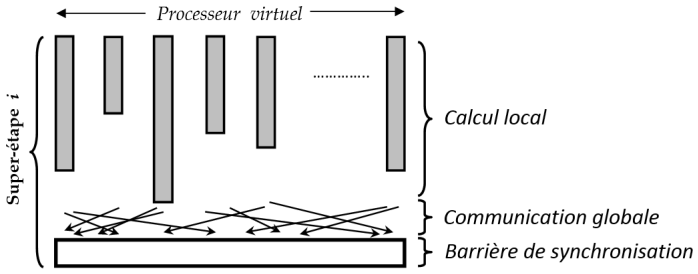


FIGURE 2.3 – Organisation d'une super-étape selon le modèle de programmation BSP

Remarque 1 *Dans la suite de ce document, le modèle BSP considéré est celui développé par McColl et la variante du modèle de coût utilisée est le modèle de coût standard.*

2.3.2 Autres modèles BSP développé

Malgré tous les outils de prédiction de performances que propose le modèle BSP, le modèle de coût standard surestime le coût réel d'exécution de certains algorithmes, et le sous-estime pour d'autres algorithmes.

Modèle BSP développé de Miller

D'après Miller [Mil94], l'imprécision des performances observée provient des coûts affectés aux opérations de chacune des super-étapes. Précisément, le paramètre g dont les valeurs sont mesurées dans les conditions où le paquetage est exploité au maximum, ne permet pas d'obtenir des estimations exactes du coût de la communication pour les super-étapes à petite h-relation, là où le paquetage n'arrive pas à rendre l'effet du start-up [2] négligeable.
Ainsi, pour produire des prédictions plus exactes du coût de communication dans une super-étape, la valeur utilisée pour l'estimation du coût d'envoi d'un mot doit résulter d'une mesure (benchmark) faite dans le même contexte que celui dans lequel s'exécute la super-étape. Le contexte étant défini dans ce cas par le triplet *(machine parallèle, système d'exécution BSP, taille de la h-relation)*.

Pour ce faire, Miller [Mil94] ajoute un nouveau paramètre au modèle de coût BSP et propose une formule pour calculer les valeurs du paramètre g. Ainsi, pour deux messages M_1 et M_2 de tailles différentes, $g(M_2) < g(M_1)$ si $M_1 < M_2$.

Modèle BSP développé de Kechid-Myoupo

D'après Kechid et Myoupo [KM05], utiliser des valeurs de g calculées à partir de benchmarks faits dans le même contexte (même machine parallèle, même système d'exécution BSP et même taille de la h-relation) mais avec des schémas de communication différents de ceux des super-étapes à prédire donnera sûrement de faux résultats. Ainsi, le sens du terme contexte mérite d'être une fois de plus incrémenté pour comprendre en plus de la machine parallèle, de l'implémentation BSP et de la taille de la h-relation : le schéma de communication.

Ce raffinement n'agit pas sur l'expression du modèle de coût. Il intervient plutôt au niveau des techniques de benchmark des paramètres L et g dans les machines réelles et leur utilisation au niveau de la prédiction. Dans ce raffinement, le schéma de communication est pris en compte seulement pour les super-étapes à petites h-relation, car c'est dans ceux-là que le schéma de communication influe sur le coût d'envoi d'un mot.

2. Le start-up est le coût supplémentaire dû au temps système pour l'initialisation d'une communication.

2.4 Modèle BSP simplifié : le modèle CGM

Beaucoup d'applications expriment un besoin de haute-performance qui s'intensifie d'une année à l'autre, mais les systèmes parallèles à grain fin apparus les premiers et traditionnellement utilisés dans le traitement de ces applications, n'arrivent plus à y aller de pair [Ola08]. L'une des raisons est que le matériel parallèle ces dernières années migrent des systèmes à grain fin vers des systèmes multi-processeurs gros-grain [Ola08]. Comme nous l'avons dit à la section 2.2, ces architectures convergent vers des machines parallèles à gros-grain constitués d'un ensemble d'ordinateurs complets reliés par un réseau de communication : ce sont les *« Clusters de stations de travail »*. En novembre 2008, 82% des 500 super-calculateurs les plus rapides du monde sont des Clusters [MSDS10]. Le problème qui se pose est que la majorité du logiciel parallèle traditionnel, conçu généralement à grain-fin et pour des machines à mémoire partagée, n'est pas adapté aux machines multi-processeurs gros-grain. C'est ce qu'on appelle dans la littérature du calcul parallèle *l'écart (gap) entre le logiciel et le matériel* [Son01] ou aussi *la barrière du logiciel* (parallel software barrier) [McC95a]. Dans l'éditorial consacré au calcul scalable (pratique et expérience), du journal scientifique international du calcul parallèle et distribué, Song [Son01] conclut que : *« Dans la prochaine décennie, le défi des chercheurs en conception d'algorithmes parallèles est de réduire cet écart »*.

Des expérimentations montrent que les modèles de calcul parallèle à gros-grain permettent de produire des algorithmes qui sont bien adaptés aux clusters [CDBL08, DFC+02, Deh99]. C'est dans ce contexte que Dehné et al [DFRC93] ont proposés le modèle CGM (*Coarse-Grained Multicomputer*) qui est une version complètement gros-grain du modèle BSP [Deh99, Deh06]. Ainsi, ils contribuent à la réduction de l'écart logiciel-matériel dans le calcul parallèle.

2.4.1 Structure d'un algorithme CGM

Le modèle CGM (*Coarse-Grained Multicomputer*) est une version simplifiée du modèle BSP. Il s'affranchit des paramètres L, g, h et w ainsi que de l'étape de synchronisation du modèle BSP. Ce modèle n'utilise que deux paramètres :

- p : qui est le nombre de processeurs utilisés ; et
- n : qui est le nombre de données en entrée du problème.

Dans ce modèle, p doit être nettement inférieur à n ($p \ll n$). Ce modèle représente beaucoup mieux les architectures existantes composées de plusieurs milliers de processeurs qui peuvent traiter des millions, voir des milliards de données. Sa particularité par rapport

au modèle BSP peut s'exprimer à deux niveaux : au niveau de la granularité de calcul et au niveau de la structure des super-étapes.

Au niveau de la granularité de calcul, ce modèle donne explicitement le nombre de données par processeur. En effet, pour une taille de problème égale à n, chaque processeur est supposé posséder une mémoire locale de taille en $O\left(\frac{n}{p}\right)$. Il est supposé que le rapport $\frac{n}{p}$ est très supérieur à 1, ce qui permet de qualifier ce modèle de gros grain. Ceci est vrai en pratique car la plupart des applications parallèles actuelles manipulent des tailles de données relativement très grandes (de l'ordre de millions de données en entrée) par rapport aux nombres de processeurs (de l'ordre de centaines) dans les machines parallèles actuelles [Deh06]. La plupart du temps, il est supposé que $\frac{n}{p} \geq p$, car ainsi, chaque processeur peut stocker des informations sur les autres processeurs.

Au niveau de la structure des super-étapes, le modèle CGM peut être vu comme une version BSP exploitant explicitement les avantages du modèle BSP développé de MacColl [McC95b] (section 2.3.1). En effet, Les algorithmes écrits dans ce modèle sont composés d'une succession de deux phases : une phase où chaque processeur effectue un calcul sur ses données locales (ronde de calculs locaux) et une phase où les processeurs échangent des données afin de les redistribuer (ronde de communication globale). Pendant cette dernière phase, chaque processeur peut envoyer $O\left(\frac{n}{p}\right)$ données et en recevoir $O\left(\frac{n}{p}\right)$. Ainsi, en plus de la grosse-granularité de calcul hérité du modèle BSP [3], le modèle CGM assure explicitement la grosse-granularité des communications. De ce fait, dans une ronde de communication la h-relation est assez grande et l'effet du start-up est réduit au minimum.

Un algorithme CGM est donc une alternation de rondes de calcul et de communication. Chaque couple *(ronde de calcul, ronde de communication)* d'un algorithme CGM correspond à une super-étape BSP dont le coût de communication est $g \times \frac{n}{p}$. Un algorithme CGM de s rondes de communication est un algorithme BSP dont le coût de communication est $s \times g \times \frac{n}{p}$ [CDBL08].

2.4.2 Prédiction de coût d'algorithmes CGM

La phase de communication correspond à une communication globale entre les processeurs et n'importe quel réseau d'interconnexion peut être utilisé. Ainsi, la synchronisation

3. Avec la minimisation de la fréquence des synchronisations pour obtenir des super-étapes dans lesquelles un grand nombre de calculs sont effectués.

dans le modèle CGM n'est plus considérée à part, elle est plutôt prise en compte dans les rondes de communication. Si T_L est le temps nécessaire pour effectuer les calculs locaux, et T_C le temps pour les communications, la complexité d'un algorithme sera défini comme étant $T_L + T_C$.

Il est aussi possible de caractériser le temps d'exécution par les deux paramètres T_L et N_C (N_C étant le nombre de rondes de communications) sans prendre en compte le temps nécessaire pour les communications [4]. Faire ce choix consiste à considérer que le nombre de rondes de communications est un nombre important qui exprime véritablement un facteur d'efficacité. Le nombre de rondes de communications est caché dans la représentation de la complexité par la notation O.

La simplification du modèle de coût est l'un des avantages principaux du modèle CGM. Ce modèle de coût permet de bonnes prédictions des performances des algorithmes. En pratique, les algorithmes CGM ont des performances très compatibles avec celles prévues à leur conception.

2.4.3 Critères de conception d'algorithmes CGM

Une bonne conception d'algorithmes CGM doit être axée principalement sur les critères suivants :

- *Minimiser la taille de calcul par processeur* : lors des calculs locaux, ceci est réalisé idéalement par l'utilisation sur chaque processeur, des algorithmes optimaux adaptés aux fragments du problème traité ;
- *Minimiser le nombre de ronde de communication* : le modèle CGM cherche à obtenir un nombre de ronde de communications qui soit le plus petit possible. Typiquement ce nombre doit être en $O(\log p)$, et l'idéale est d'obtenir un nombre constant de rondes de communications. Mais en aucun cas ce nombre ne doit dépendre de la taille du problème à résoudre, puisque dans ce cas, lorsque la taille de l'entrée augmente, le nombre de communication augmente aussi et ceci est très souvent à l'origine de mauvaises performances ;
- *Minimiser le nombre de messages dans chaque ronde de communication* : pour diminuer d'avantage l'influence du start-up sur le coût des communications, l'accroissement de la taille du problème doit engendrer l'accroissement de la taille des messages et non le nombre de messages. Un bon schéma de communication permet de diminuer d'avantage les communications.

4. Car le coût de chacune des rondes de communication est le même (majoré par $O\left(g \times \frac{n}{p}\right)$).

Corollaire 1 *Les algorithmes CGM dont l'accroissement de la taille de données n'engendre ni l'accroissement du nombre de rondes de communication, ni celui du nombre de messages par ronde, sont extensibles*[5] *et efficace*[6] *sur les clusters [CDBL08, Deh06]. Cette propriété est l'un des avantages fondamentaux de la conception d'algorithmes parallèles via le modèle CGM.*

Les algorithmes proposés dans cette thèse sont conçus suivant le modèle CGM. Leurs coûts sont également exprimés à travers le modèle de coût CGM (c'est-à-dire en fonction du temps de calcul par processeur et du nombre de rondes de communication). En raison des correspondances que nous avons faites sur les modèles BSP et CGM, nous utilisons dans cette thèse l'expression « Algorithme BSP/CGM » au lieu de « Algorithme CGM ».

2.4.4 Motivation du choix du modèle CGM pour cette thèse

L'aspect peu contraignant, réaliste et portable du modèle CGM permet d'implanter des algorithmes et de les expérimenter sur des grappes de machines divers et variées. En effet, le modèle CGM permet de s'affranchir des paramètres présents dans BSP et LogP, d'obtenir des solutions facilement implantables et de concevoir des algorithmes qui ne sont pas dépendant d'une architecture spécifique.

Il permet aussi de réutiliser des algorithmes optimaux lors des calculs locaux. Ce qui permet d'abaisser le coût du développement qui n'est pas négligeable lors d'implantations parallèles. Si la machine ne contient que des monoprocesseurs, les algorithmes optimaux pourront être utilisés.

C'est donc un modèle *« moderne »* adapté aux différentes plates-formes existantes. Loin d'être un modèle purement théorique comme le modèle PRAM, il permet de concilier théorie et pratique.

2.5 Synthèse

Dans ce chapitre, nous avons présentés les spécificités d'un certain nombre de modèles de calcul parallèle. Nous avons commencés par le modèle BSP qui a été proposé dans le but de concevoir des algorithmes aussi efficaces que portables. Nous avons vu qu'il permet de décrire la majorité des machines parallèles tout en restant réaliste. Il permet aussi d'effectuer de bonnes prédictions des performances des algorithmes à l'aide du modèle de

5. Ne perdent pas leur performance avec l'augmentation de la taille du problème.
6. C'est-à-dire qu'ils ont des performances compatibles avec celles prédites à leur conception.

coût standard.
Le raffinement de McColl [McC95b] permet de fusionner la phase de synchronisation avec la phase de communication globale effectuée une seule fois à la fin de chaque super-étape. Ainsi, les communications sont mieux gérées et le coût d'implantation des algorithmes est plus faible. Dans le but d'améliorer la prédiction des performances des algorithmes BSP, les raffinements de Miller [Mil94] et de Kechid-Myoupo [KM05] proposent d'ajouter deux éléments au contexte d'exécution des algorithmes BSP : la taille de la h-relation et le schéma de communication.

Comme le modèle CGM, en plus des avantages hérités du modèle BSP, permet de réduire l'écart entre le logiciel et le matériel, il est le choix idoine pour la conception des algorithmes parallèles permettant de résoudre les problèmes de la classe abordée dans cette thèse. Cette classe de problèmes est présentée dans le chapitre 3.

Chapitre 3

Classe de problèmes étudiée

Sommaire

3.1 Introduction

Dans sa forme la plus générale, un problème d'optimisation combinatoire (on dit aussi d'optimisation discrète) consiste à trouver dans un ensemble discret, un parmi les *meilleurs* sous-ensembles (ou solutions) réalisables. La notion de *meilleure solution* étant définie par une fonction objectif (un ou plusieurs critères). La plupart des problèmes d'optimisation doivent être posés de manière dynamique (avec un historique), et non à un moment donné de manière isolée. En effet, d'une part, les conséquences de chaque décision s'échelonnent sur une période relativement longue et l'évaluation des effets immédiats ne doit pas constituer le seul critère d'appréciation. D'autre part, l'éventail des possibilités dépend de l'évolution passée, en particulier les contraintes introduites par les décisions précédentes.

Il existe plusieurs méthodes permettant de résoudre les problèmes d'optimisation, parmi lesquelles, le DPR (*Diviser Pour Régner*), les algorithmes aveugles [1], l'utilisation

1. Les algorithmes aveugles prennent les décisions à une étape donnée de leur déroulement sans tenir compte des décisions prises au cours des étapes précédentes.

des fonctions heuristiques[2], la programmation dynamique, etc. Les algorithmes aveugles qui prennent la meilleure décision locale à chaque pas, sans tenir compte des décisions précédentes, posent très souvent le *« problème du maximum local »*[3] et ne conduisent pas toujours à un maximum global. Pour résoudre ce problème, l'on utilise souvent les fonctions heuristiques, mais ceux-ci ne minimisent pas le nombre de calculs nécessaires au traitement à effectuer. D'autres méthodes plus récentes existent et garantissent que les calculs qui se répètent avec certaines méthodes ne soient effectués qu'une et une seule fois. Parmi ces méthodes, la programmation dynamique est l'une des plus utilisée.

Dans ce chapitre, nous nous intéressons à la résolution d'une classe de problèmes d'optimisation, en utilisant la technique de programmation dynamique. Après avoir présenté le principe de résolution d'un problème par la programmation dynamique, nous présenterons les différents problèmes d'optimisation abordés et quelques solutions (algorithmes) séquentielles permettant de les résoudre. Nous proposerons par la suite un moyen d'améliorer les performances de l'une des solutions (sans toutefois changer sa complexité temporelle).

3.2 Programmation dynamique

La programmation dynamique (DP pour *« Dynamic Programming »*) peut être définie comme une technique de recherche opérationnelle employée pour résoudre une large variété de problèmes d'optimisation discrets [Gen96, SL06]. C'est une méthode d'optimisation procédant par énumération implicite des solutions. Bien que déjà pratiquée auparavant, elle est élevée au rang de méthode générale de résolution avec les travaux de Bellman, qui formalise l'approche et la baptise. Depuis, elle n'a cessé d'évoluer et une multitude de travaux de recherches s'intéressent à l'analyser en tant que méthode de résolution et à l'utiliser dans la résolution des problèmes d'optimisation [Gen96, SL06]. En effet, beaucoup de problèmes dans des domaines divers (recherche opérationnelle, apprentissage, bio-informatique, commande de systèmes, linguistique, théorie des jeux, processus de Markov, etc.) ont pu trouver une solution élégante suivant la technique de programmation dynamique. On peut citer par exemple : la clôture transitive d'une relation, le plus court chemin dans un graphe, le calcul de la distance de deux chaînes et plus généralement les problèmes d'alignement de séquences, l'analyse syntaxique de phrases, les arbres binaires optimaux, la multiplication d'une chaine de matrices, la triangulation d'un

2. Fonctions qui permettent de déterminer la meilleure solution parmi plusieurs, elles peuvent tenir compte des décisions prises dans les étapes précédentes.

3. Le fait de prendre la meilleure décision à chaque pas ne garantit pas que la solution finale sera optimale.

polygone, etc.

3.2.1 Conditions d'utilisation

La programmation dynamique s'applique aux problèmes vérifiant les deux propriétés suivantes :

- *La propriété de l'optimalité* : dans une séquence optimale (de décisions ou de choix), chaque sous-séquence doit aussi être optimale, c'est-à-dire que quelque soit l'état initial d'un problème, les décisions prises dans les états suivants doivent être optimales et conformes aux décisions prises dans les états précédents ;
- *La propriété de chevauchement de sous-problèmes* : dans de tels problèmes, les sous-problèmes ne sont pas disjoints. Ils ont des sous-problèmes en commun [LRSC01]. Cette caractéristique apparaît principalement dans les problèmes d'optimisation dans lesquelles un même sous-problème peut être résolu plusieurs fois dans le processus récursif de résolution.

Des problèmes combinatoires peuvent avoir la propriété de l'optimalité, mais peuvent employer trop de mémoire ou de temps pour être efficaces. Le problème du voyageur de commerce en est un exemple [4]. Puisque la programmation dynamique est une technique d'optimisation permettant le traitement efficace de fonctions récursive en triant les résultats partiels, elle peut être efficace uniquement lorsqu'il n'y a pas trop de résultats partiels à calculer. Par exemple, il y a $n!$ permutations d'un jeu de n éléments, l'on ne peut donc pas employer la programmation dynamique pour stocker la meilleure solution pour chaque sous-permutation. De même, il y a 2^n sous-ensembles d'un ensemble de n éléments, et de ce fait, l'on ne peut employer la programmation dynamique pour stocker la meilleure solution pour chacun d'eux.

Cependant, il y a seulement $n(n-1)/2$ sous-chaînes d'une chaîne de longueur n donnée. Chacune décrite par un point de départ et d'une fin, il est donc possible, et même opportun, d'employer la programmation dynamique pour des problèmes sur les chaînes. De même, il y a seulement $n(n-1)/2$ sous-arbres possibles d'un arbre binaire de recherche de n nœuds, chacun décrit par une racine et des feuilles, ainsi, l'on peut employer la programmation dynamique pour l'optimisation d'arbres binaires de recherche.

Plus généralement, *la programmation dynamique est plus efficace sur des objets que l'on peut ordonner linéairement et qui ne peuvent pas être réordonnés* : des caractères dans

4. Le problème du voyageur de commerce peut être énoncé comme suit : étant donné un ensemble de villes séparées par des distances données, trouver le plus court chemin qui relie toutes les villes, en passant une et tune seule fois dans chaque ville.

une chaîne, des matrices dans une chaîne, des points sur les contours d'un polygone, le parcours de gauche à droite dans un arbre de recherche, etc.

3.2.2 Principe et méthode de résolution

La stratégie utilisée en programmation dynamique se résume en trois points essentiels [KK93] :

1. Formuler la solution comme une relation de récurrence ou un algorithme récursif ;
2. Montrer que le nombre des différents cas de figure possible à chaque étape est borné de façon polynomiale ;
3. Spécifier un ordre d'évaluation pour les résultats intermédiaires de sorte à avoir ce dont on a besoin pour continuer à chaque moment.

L'idée de la programmation dynamique est de subdiviser le problème à résoudre en sous-problèmes et organiser leur résolution de sorte que chacun d'eux soit évalué une seule fois. Précisément, on cherche une solution optimale par rapport à un certain coût (ou fonction objectif), et donc, c'est non seulement le meilleur coût qu'on cherche, mais aussi une solution qui corresponde à ce coût. Ceci peut se faire de deux manières différentes : de manière descendante [5] (*top-down dynamic programming*), ou bien de manière ascendante (*bottom-up dynamic programming*).

Méthode descendante :

Dans la méthode descendante, dite aussi *méthode de mémorisation* ou *méthode récursive*, l'arbre de récursivité est traité (de haut en bas) suivant un parcours en profondeur postfixe. Pour éviter la redondance de calcul, l'algorithme est paramétré de telle sorte qu'il se souvient des solutions des sous-problèmes déjà résolus (rencontrés dans le parcours).

Méthode ascendante :

Dans la méthode ascendante, on se base sur le choix du bon ordre de résolution des sous-problèmes. Cet ordre doit garantir que chaque sous-problème soit traité une seule fois, et qu'il ne soit traité qu'après le traitement de tous les sous-problèmes desquels il dépend. Pour ce faire, on subdivise le problème en étapes, chaque étape est composée de plusieurs sous-problèmes, et a une stratégie de résolution associée et un certain nombre d'états associés. La solution de chacun des sous-problèmes d'une étape donnée ne dépend que de celles des sous-problèmes appartenants aux étapes précédentes. Les sous-problèmes sont résolus de manière ascendante, de la première à la dernière étape.

5. Comme pour la technique DPR (*Diviser Pour Régner*).

Les dépendances entre les sous-problèmes peuvent se représenter sous la forme d'un *graphe multi-niveaux, orienté et acyclique*[6] ou *multi levels DAG* (multi levels *Direct Acyclic Graph*). Il est structuré comme suit :

- Un niveau (une étape) est un ensemble de nœuds ;
- Chaque nœud représente un ensemble de sous-problèmes[7]. Le nœud qui ne dispose pas d'arcs sortants représente le problème original à résoudre, et les nœuds qui n'ont pas d'arc entrant correspondent aux éléments initiaux du problème ;
- S'il existe un arc allant d'un nœud N_1 vers un nœud N_2, alors, le calcul des solutions optimales des sous-problèmes correspondants au nœud N_2 dépend de ceux des sous-problèmes correspondant au nœud N_1 ;
- Deux nœuds qui ne sont pas reliés par un arc sont dits nœuds indépendants.

Une étape est un ensemble de sous-problèmes mutuellement indépendants. Un sous-problème appartient à l'étape i si dans le DAG-multi-niveaux sus présenté, le nœud qui le représente n'a pas d'arc allant vers un nœud de l'étape j, $(j \leq i)$ et contient au moins un arc allant vers un nœud de l'étape $i + 1$.

La résolution d'un problème de programmation dynamique se réduit au calcul des valeurs des nœuds du DAG-multi-niveaux décrit ci-dessus. Ce graphe est appelé dans la littérature *« graphe des tâches »* ou *« graphe de la programmation dynamique »*[Bra94, GGKK03, Kec09].

Un algorithme qui résoud les sous-problèmes correspondants à ce DAG-multi-niveaux, niveau par niveau, suivant une approche ascendante, allant du niveau le plus bas vers le niveau le plus élevé, est appelé *« Algorithme de programmation dynamique »*. Pour chacun des sous-problèmes qu'il traite, cet algorithme sauvegarde :

1. la valeur de sa solution optimale dans une table dite *« Table de programmation dynamique »* ;
2. les informions nécessaires à la construction de cette solution dans une table dite *« Table de traces »*.

A partir du même ensemble de sous-problèmes, on peut souvent construire différents DAG-multi-niveaux et ainsi avoir différents algorithmes de programmation dynamique. Selon l'intensité des dépendances entre les nœuds du même niveau, deux types de DAG-multi-niveaux se distinguent :

- les *DAG-multi-niveaux à niveau-indépendant* si tous les nœuds de chaque niveau sont indépendants les uns des autres et

6. C'est un graphe qui ne contient pas de cycle.
7. Dans le cas le plus simple, cet ensemble est un singleton.

– les *DAG-multi-niveaux à niveau-dépendant* dans le cas contraire.

Les algorithmes de programmation dynamique inhérents aux différents DAG-multi-niveaux d'un problème ont, en général, la même performance. Cependant, si l'objectif est de paralléliser, l'idéal est d'utiliser un DAG-multi-niveaux à niveau-indépendant.

Terminologie 1 *Nous appelons problème de programmation dynamique tout problème dont la solution peut s'obtenir à l'aide d'un algorithme de programmation dynamique.*

3.2.3 Formulation de solution

La solution d'un problème de programmation dynamique est souvent représentée par une équation récursive appelée dans la littérature : *équation fonctionnelle, équation d'optimisation* ou encore *fonction de coût* [GGKK03]. Dans cette équation, le membre de gauche est une quantité inconnue (la solution optimale du sous-problème à résoudre), et le membre de droite est une expression de minimum (où maximum) d'un ensemble d'éléments, écrit chacun en fonction de solutions de sous-problèmes appartenant aux niveaux précédents. La nature de la fonction de composition dépend du problème en question. L'équation 3.1 est l'équation fonctionnelle du problème de la recherche du plus court chemin entre une paire de nœuds dans un graphe acyclique. Ce problème est présenté dans la sous-section 3.2.4.

Note 1 *Si la solution optimale d'un problème est déterminée par composition de solutions optimales de sous-problèmes et sélection du minimum (ou du maximum), on dit que la formulation de cette solution est une formulation de programmation dynamique.*

3.2.4 Illustration : le problème du plus court chemin

Pour illustrer les notions abordés dans les sous-sections précédentes, considérons le problème consistant à trouver le plus court chemin entre une paire de nœuds dans un graphe acyclique.

Soit un graphe contenant n nœuds numérotés par $0, 1, \ldots, n-1$, un arc reliant un nœud i à un nœud j a un coût donnée, et un arc relie le nœud i au nœud j si et seulement si $i < j$. Le nœud 0 est le nœud de départ et le nœud $n-1$ est le nœud de destination.

L'approche directe permettant de résoudre ce problème consiste à considérer tous les chemins possibles, les évaluer et les comparer. Cela conduit à une solution ayant une

complexité exponentielle [8]. Ce qui n'est pas envisageable si le graphe a un grand nombre de nœuds et que les nœuds sont fortement connectés.

Ce problème est un problème typique de programmation dynamique. En effet, désignons par $c(i, j)$ le coût d'un arc allant d'un nœud i à un nœud j et notons $(s \rightarrow d)$ le plus court chemin entre deux nœuds s et d.

On peut montrer facilement (par l'absurde) que si le chemin $(0 \rightarrow (n-1))$ passe par un nœud k, alors la partie de ce chemin entre 0 et k est le chemin optimal entre 0 et k (c'est-à-dire $(0 \rightarrow k)$). De même, la partie résiduelle de ce chemin allant de k à $(n-1)$ est aussi le chemin optimal entre k et $n-1$ (c'est-à-dire $(k \rightarrow (n-1))$. On peut donc dire que ce problème respecte le principe d'optimalité.

Soit $f(x)$ le coût du chemin le plus court allant du nœud 0 à un nœud x. La formulation récursive du problème du plus court chemin définit $f(x)$ en fonction de $f(k)$, $0 \leq k \leq x$ comme suit :

$$f(x) = \begin{cases} 0 & \text{, si } x = 0 \\ \min_{0 \leq k < x} \{f(k) + c(k, x)\} & \text{, si } 1 \leq x \leq n-1 \end{cases} \tag{3.1}$$

Cette équation découle du fait que le plus court chemin allant du nœud 0 à un nœud x quelconque est constitué du plus court chemin allant de 0 à un nœud k directement connecté à x, *et de l'arc* (k, x).

Résoudre le problème du plus court chemin entre les nœuds 0 et $n-1$ reviens à calculer la valeur de $f(n-1)$. De cette équation, on observe clairement que le problème du plus court chemin vérifie aussi la propriété de *chevauchement des sous-problèmes*.

Le DAG schématisant la relation de dépendance entre les sous-problèmes (les sous-chemins) est donné dans la figure 3.1. Plusieurs organisations de ce DAG en DAG-multi-niveaux sont possibles, certains de type niveau-dépendant (figure 3.2), et au moins un de type niveau-indépendant (figure 3.3).

3.3 Classe de problèmes abordée

Nous nous intéressons dans cette thèse à la résolution d'une classe de problèmes de programmation dynamique largement utilisée. Il s'agit des problèmes dont l'équation fonctionnelle a la forme de l'équation 3.2.

8. Le nombre de chemins possible pouvant être très élevé.

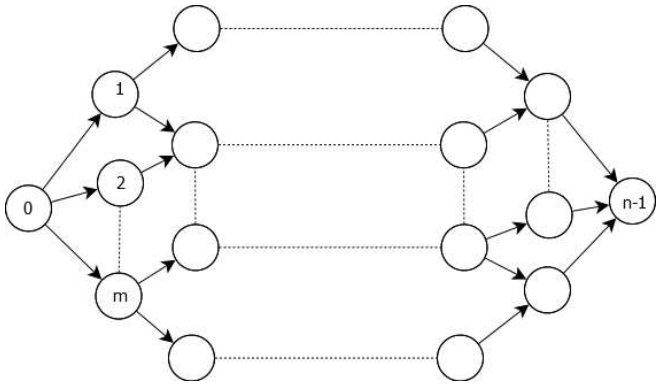


FIGURE 3.1 – DAG représentant les dépendances entre les différents sous-problèmes pour le problème du plus court chemin.

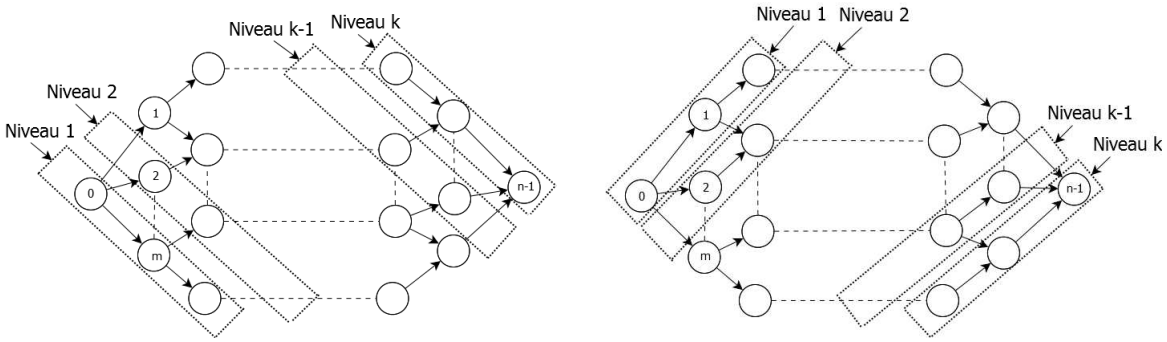


FIGURE 3.2 – DAG-multi-niveaux à niveau-dépendant pour le problème du plus court chemin

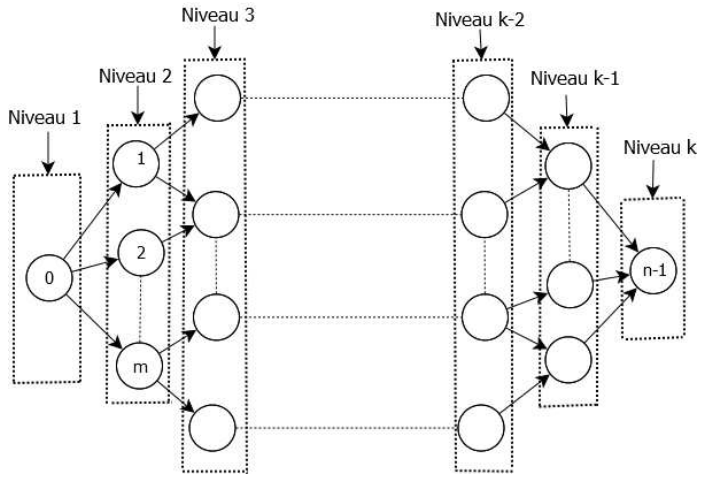


FIGURE 3.3 – DAG-multi-niveaux à niveau-indépendant pour le problème du plus court chemin

$$Cost(i,j) = \begin{cases} Init(i) & \text{, si } i = j \\ Opt_{i \leq k < j}\{Cost(i,k) + Cost(k+1,j) + U(i,k,j)\} & \text{, si } 1 \leq i < j \leq n \end{cases} \tag{3.2}$$

Dans cette équation, *Opt* représente la fonction de l'optimum qui dépend du problème à résoudre. C'est la fonction *min* quand il s'agit de minimiser le coût, et la fonction *max* quand il s'agit de maximiser. n est la taille du problème. Les valeurs $U(i,k,j)$ et $Init(i)$ sont connues d'avance ou peuvent être calculées facilement.

Dans l'équation de récurrence 3.2, $Cost(i,j)$ correspond à la valeur de la solution optimale du sous-problème (i,j). Il est obtenu par l'addition des valeurs des solutions optimales des deux sous-problèmes (i,m) et $(m+1,j)$ où $i \leq m < j$. Le coût de la fusion (union) de ces deux sous-problèmes est donné par la fonction de l'union $U(i,m,j)$. Parmi les $(j-i)$ décompositions possibles du (sous-) problème (i,j) en deux sous-problèmes, le couple de sous-problèmes $((i,m),(m+1,j))$ est celui qui correspond à la décomposition qui donne la solution optimale.

Beaucoup de problèmes appartiennent à la classe sus-présentée[9]. Ceux qui retiennent particulièrement notre attention sont : le problème de parenthésage minimale (MPP pour *« Minimum Parenthesization Problem »*), le problème de triangulation de polygone convexe (TCP pour *« Triangulation of Convexe Polygon »*) et le problème de construction d'arbre binaire de recherche optimale (OBST pour *« Optimal Binay Search Tree »*).

La différence principale entre ces problèmes réside dans la définition de la fonction U dont le rôle est de fournir, pour chaque sous-problème (i,j), les différentes valeurs $U(i,k,j)$ correspondant aux *coûts de l'union* des solutions optimales des couples de sous-problèmes $((i,k),(k+1,j))$ pour $k = i,\ldots,j-1$. La définition de cette fonction dépend de la sémantique du problème à résoudre.
Nous présentons les différents problèmes sus-cités dans les sous-sections suivantes.

3.3.1 Problème de parenthésage minimale (MPP)

Le paranthésage minimale d'une chaîne de n symboles consiste à y insérer des parenthèses (ouvrantes et fermantes) afin de définir le meilleur ordre de traitement séquentiel (suivant un objectif visé), via une loi de composition interne sur cet ensemble de symboles. Dans la littérature, on appelle processus de parenthésage tout processus permettant de définir un tel ordre, même si ce processus n'est pas basé sur les parenthèses (par exemple via

9. Par exemple : le problème de l'analyseur syntaxique CYK et l'algorithme de Nussinov sur l'ARN [Dur98].

un arbre). Le nombre de parenthésages possibles pour une chaîne de taille n est $\Omega\left(\frac{4^n}{n^{3/2}}\right)$ [Knu71, Kec09]. Ce nombre est exponentiel en n.

Le problème de parenthésage optimal d'une chaîne de symboles, souvent désigné par l'acronyme MPP (*« Minimum Parenthesization Problem »*), consiste à trouver pour une chaîne de symboles donnée (caractères, nombres, matrices, objets, etc.), le parenthésage qui correspond au coût optimal des traitements qu'elle va subir. Selon le type d'entités de la chaîne à parenthéser et du traitement à effectuer (c'est-à-dire, la sémantique de la loi de composition interne utilisée), ce problème apparaît dans la littérature sous plusieurs variantes. La plus connue est le problème d'Ordonnancement de Produit de Chaîne de Matrices (OPCM) souvent désigné par l'acronyme MCOP (*« Matrix Chain Ordering Problem »*) [LRSC01]. L'objectif dans ce problème est de trouver le parenthésage minimisant le coût du produit d'une chaîne de matrices. Une autre variante bien connue du MPP est le problème de parenthésage de l'ordre lexical des blocs de calcul pour les programmes incorporés dans les microprocesseurs de calcul du type DSP [10] (*Digital Signal Processor*). L'objectif est de trouver le parenthésage minimisant le besoin en mémoire sur les DSP [BML95].

3.3.2 Problème d'Ordonnancement de Produit de Chaîne de Matrices (MCOP)

À cause de ses diverses applications [LKHL03], ce problème est l'un des problèmes d'optimisation les plus étudiés. Il consiste à déterminer le bon ordre de multiplication dans les produits de chaînes de matrices.

Formalisation du problème :

Soit P le produit d'une chaîne de n matrices $M_1, M_2, \ldots, M_i, \ldots, M_n$ de dimension respective $(d_0, d_1), (d_1, d_2), \ldots, (d_{i-1}, d_i), \ldots, (d_{n-1}, d_n)$, une matrice M_i étant de dimension (d_{i-1}, d_i).

$$P = M_1 \times M_2 \times \ldots \times M_i \times \ldots \times M_n \tag{3.3}$$

La multiplication des matrices étant une opération associative, l'on obtient le même résultat en modifiant l'ordre de multiplication (le parenthésage) des matrices. Toutefois,

10. Un DSP (de l'anglais *« Digital Signal Processor »*, qu'on pourrait traduire par *« processeur de signal numérique »* ou *« traitement numérique de signal »*) est un microprocesseur optimisé pour exécuter des applications de traitement numérique du signal (filtrage, extraction de signaux, etc.) le plus rapidement possible.

la multiplication de la même chaîne suivant deux parenthésages différents n'a pas le même coût. Le coût de la multiplication dépend du nombre d'opérations élémentaires qu'il nécessite. Le coût de la multiplication de deux matrices de dimensions (h, m) et (m, n) est supposé égal à $h \times m \times n$.

Illustration :

Considérons quatre matrices M_1, M_2, M_3 et M_4 de dimensions respectives $(100, 1)$; $(1, 50)$; $(50, 20)$ et $(20, 10)$. Pour calculer le produit de la chaîne $M_1 \times M_2 \times M_3 \times M_4$ on peut procéder de plusieurs façons (parenthésages) différentes, parmi lesquelles :

1. Possibilité 1 : $((M_1 \times M_2) \times M_3) \times M_4$, dont le coût de calcul est $(100 \times 1 \times 50) + (100 \times 50 \times 20) + (100 \times 20 \times 10) = 125.000$;
2. Possibilité 2 : $M_1 \times ((M_2 \times M_3) \times M_4)$, dont le coût de calcul est $(1 \times 50 \times 20) + (1 \times 20 \times 10) + (100 \times 1 \times 10) = 2.200$.

On constate que le coût des calculs est de 125.000 opérations pour le premier parenthésage alors qu'il n'est que de 2.200 opérations pour le second. Ainsi, l'ordre d'évaluation des éléments pour le produit d'une chaîne de matrices ne change pas le résultat final mais influence énormément le nombre d'opérations nécessaires pour ce calcul.

Formulation de solution :

Désignons par $P_{i,j}$ le sous-produit $M_i \times M_{i+1} \times \ldots \times M_j$ et par $Cost(i, j)$ son coût optimal [11]. Le produit $P_{i,i}$ désigne alors la matrice M_i et son coût optimal $Cost(i, i)$ est nul. La dimension de la matrice résultat d'un produit $P_{i,j}$ étant (d_{i-1}, d_j), son coût optimal $Cost(i, j)$ se définit récursivement comme suit :

$$Cost(i,j) = \begin{cases} 0 & \text{, si } 1 \leq i = j \leq n \\ \min_{i \leq k < j} \{Cost(i,k) + Cost(k+1,j) + d_{i-1} \times d_k \times d_j\} & \text{, si } 1 \leq i < j \leq n \end{cases} \tag{3.4}$$

calculer efficacement P (avec $P = M_1 \times M_2 \times \ldots \times M_i \times \ldots \times M_n$) reviens à calculer le coût $Cost(1, n)$. La récurrence 3.4 est équivalente à la récurrence 3.2 avec $Opt = \min$, $U(i, k, j) = d_{i-1} \times d_k \times d_j$ et $Init(i) = 0$, $i = 1, ..., n$.

Propriétés de programmation dynamique :

On remarque clairement, à partir de l'équation 3.4, que le problème MCOP vérifie les propriétés de la programmation dynamique vues à la section 3.2.1 :

11. C'est-à-dire le nombre minimum d'opérations arithmétiques scalaires nécessaires à son calcul.

- *La propriété de l'optimalité de la sous-structure* : On observe que la recherche du coût optimal d'un (sous-) produit $P_{i,j}$ englobe en elle la recherche des coûts optimaux de tous les sous-produits $P_{l,f}$, $(i \leq l < f \leq j)$;
- *La propriété de chevauchement de sous-problèmes* : Ceci s'observe en considérant deux (sous-) problèmes $P_{i,j}$ et $P_{i+1,j}$ quelconque dont les coûts de résolution sont respectivement $Cost(i,j)$ et $Cost(i+1,j)$. D'après la récurrence 3.4, l'évaluation de ces deux coûts nécessite en commun les valeurs $Cost(k,j)/i+1 < k \leq j$ des coûts d'évaluation des sous-problèmes $P_{k,j}$, $i+1 < k \leq j$. Il en est de même pour les (sous-) problèmes $P_{i,j}$ et $P_{i,j+1}$ dont l'évaluation des coûts $Cost(i,j)$ et $Cost(i,j+1)$ nécessite en commun les valeurs $Cost(i,m), i \leq m < j$ des coûts d'évaluation des sous-problèmes $P_{i,m}$, $i \leq m < j$.

Note 2 *De la même façon, on peut retrouver les critères d'utilisation de la technique de programmation dynamique pour tous les problèmes de la classe étudiée.*

3.3.3 Problème de la recherche de l'Arbre Binaire de Recherche Optimal (OBST)

Une des méthodes les plus populaires pour faciliter la recherche d'informations à partir d'une clé (tel qu'un nom), consiste à stocker ces informations dans un arbre binaire de recherche. Le *problème de recherche de l'arbre binaire de recherche optimal* (OBST pour *« Optimal Binary Search Tree »*) est un problème de recherche opérationnelle très étudié.

Un arbre binaire de recherche est un arbre binaire qui permet d'effectuer une recherche efficace sur l'ensemble ordonné des éléments qui le constituent. Un tel arbre ordonne complètement les éléments qui le composent de la manière suivante :

- Toutes les valeurs inférieures à celle d'un nœud sont disposées dans son sous arbre gauche ;
- Toutes les valeurs supérieures à celle d'un nœud sont disposées dans son sous arbre droit.

Le coût de la recherche d'un élément donné dans un arbre binaire de recherche dépend de sa profondeur dans l'arbre. Par exemple sur la figure 3.4, le coût de la recherche de l'élément *rho* est 1 tandis que celui de l'élément *kiss* est 5. Il est facile de constater que :

- Pour trier un ensemble de n éléments, il est possible de construire $\Omega\left(\frac{4^n}{n^{3/2}}\right)$ arbres binaire différents [Knu71]. La figure 3.5 présente les différents arbres possibles contenant les mots {do, re, mi} ;
- Le coût de la recherche d'un élément k dépend de l'arbre utilisé pour la recherche.

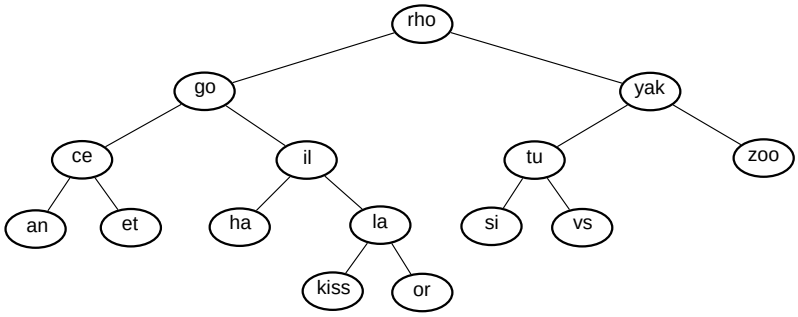


FIGURE 3.4 – Arbre binaire de recherche correspondant à l'ensemble de mots {an, ce, et, go, ha, il, kiss, la, or, rho, si, tu, vs, yak, zoo} trié par ordre alphabétique.

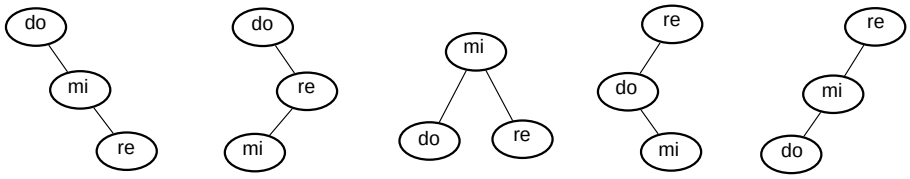


FIGURE 3.5 – Pour la séquence {do, re, mi}, il existe 5 arbres binaires possible et le coût de la recherche de l'élément "do" est 1 pour le premier arbre, et 2 pour le quatrième.

Formalisation du problème :

Soit donné un ensemble de n éléments $k_1 < k_2 < \ldots < k_n$ à disposer sur un arbre binaire de recherche, et $2n+1$ probabilités $p_1, p_2, \ldots, p_n, q_0, q_1, q_2, \ldots, q_n$ avec $\sum p_i + \sum q_i = 1$ où :

- $p_i/(1 \leq i \leq n)$ est la probabilité que la valeur recherchée durant le processus de recherche soit égale à k_i ;
- q_i est la probabilité que la valeur recherchée durant le processus de recherche soit :
  - dans l'intervalle $]k_i, k_{i+1}[$ si $0 < i < n$;
  - inférieure à k_1 si $i = 0$;
  - supérieure à k_n si $i = n$.

Soit T l'arbre binaire de recherche constitué de cet ensemble d'éléments. Soit $\{f_0, f_1, f_2, \ldots, f_n\}$ un ensemble de feuilles fictives rajoutées dans l'arbre T, où f_j,$(0 < j < n)$ représente les valeurs de l'intervalle $]k_i, k_{i+1}[$, f_0 et f_n représentent respectivement les valeurs inférieures à k_1, et les valeurs supérieures à k_n. Soit b_i le nombre d'arcs constituant le chemin allant de la racine à un nœud interne k_i et a_i le nombre d'arcs constituant le chemin allant de la racine à la feuille f_i. Le coût moyen de recherche d'une valeur dans

l'arbre T peut être donné par l'expression :

$$Cost(T) = \sum_{0<i\leq n} p_i \times (b_i + 1) + \sum_{0\leq j\leq n} q_j \times a_j$$

b_i+1 est le coût nécessaire pour accéder à l'élément k_i tandis que celui de f_j est a_j. L'arbre binaire de recherche optimale pour cet ensemble d'éléments est un arbre binaire dans lequel la recherche se fait avec *un coût de recherche moyen minimal*. Ainsi, le problème de recherche de d'arbre binaire de recherche optimale consiste à construire un arbre binaire de recherche optimale à partir d'un ensemble d'éléments et leurs probabilités d'accès.

Formulation de solution :

Désignons par $obst(i, j)$ l'arbre binaire optimal dont les éléments correspondent au sous-ensemble $\{k_{i+1}, \ldots, k_j\}$ et désignons par $Tree(i, j)$ le coût de construction de cet arbre. Soient $w(i, j) = p_{i+1} + \ldots + p_j + q_i + q_{i+1} + \ldots + q_j$ et $w(i, i) = q_i$. Il n'est pas difficile d'observer que $w(0, n) = 1$ et que le coût $Tree(i, j)$ obéit à la relation de récurrence suivante :

$$Tree(i,j) = \begin{cases} q_i & \text{, si } 0 \leq i = j \leq n \\ \min_{i<k\leq j} \{Tree(i, k-1) + Tree(k, j) + w(i, j)\} & \text{, si } 0 \leq i < j \leq n \end{cases} \tag{3.5}$$

Le coût de la recherche de l'arbre binaire de recherche optimal est $Tree(0, n)$. La récurrence 3.5 est équivalente à la récurrence 3.2 avec $Opt = \min$, $U(i, k, j) = w(i, j)$ et $Init(i) = q_i$, $i = 0, ..., n$.

3.3.4 Problème de Triangulation d'un Polygone Convexe (TCP)

La triangulation de polygone convexe est un problème très étudié dans l'algorithmique géométrique. La recherche de la triangulation la moins coûteuse constitue l'un des aspects les plus important de ce problème.

Formalisation du problème :

Considérons un polygone convexe[12] P du plan, constitué de $(n + 1)$ sommets pondérés $\langle v_0, v_1, \ldots, v_n\rangle$. Chaque sommet v_i a un poids $w(v_i)$. Une triangulation de P est un ensemble de $(n - 3)$ diagonales (dites aussi cordes internes) de P qui ne se croisent pas. Une triangulation sépare donc le polygone P en $(n - 2)$ triangles disjoints.

Le coût d'un triangle de sommets v_i, v_j et v_k est défini par le produit des poids de

12. Un polygone est dit convexe si tous ses angles sont strictement inférieurs à 180°.

ses sommets ($w(v_i) \times w(v_j) \times w(v_k)$), et le coût d'une triangulation d'un polygone est la somme des coûts de ses triangles. Le problème est de trouver une triangulation optimale (c'est-à-dire une triangulation de coût minimum).

Formulation de solution :

Désignons par $T(i,j)$, $(1 \leq i < j \leq n)$ le coût d'une triangulation optimale du polygone $\langle v_{i-1}, v_i, \ldots, v_j \rangle$ avec $T(i,i) = 0$, $(1 \leq i \leq n)$. Le coût $T(i,j)$ se défini récursivement comme suit :

$$T(i,j) = \begin{cases} 0 & \text{, si } 1 \leq i = j \leq n \\ \min_{i \leq k < j} \{T(i,k) + T(k+1,j) + w(v_{i-1}) \times w(v_k) \times w(v_j)\} & \text{, si } 1 \leq i < j \leq n \end{cases} \tag{3.6}$$

Le coût de la triangulation optimale du polygone $P = \langle v_0, v_1, \ldots, v_n \rangle$ est $T(1,n)$.

La récurrence 3.6 est équivalente à la récurrence 3.2 avec $Opt = \min$, $U(i,k,j) = w(v_{i-1}) \times w(v_k) \times w(v_j)$ et $Init(i) = 0$, $i = 1, \ldots, n$.

Tous les problèmes sus-présentés sont des problèmes de programmation dynamique, et il existe des algorithmes de programmation dynamique permettant de les résoudre.

3.4 Algorithmes séquentiels

Nous avons vu avec l'exemple de l'OBST que le nombre de solutions possibles pour un problème de taille n, appartenant à la classe de problèmes sus-présenté, est exponentiel en n : $\Omega\left(\frac{4^n}{n^{3/2}}\right)$. Ainsi, une solution basée sur l'énumération exhaustive directe est très médiocre. Au début des années 1970, des chercheurs ont remarqué la présence des caractéristiques de l'optimalité et du chevauchement des sous-problèmes pour le problème MCOP. En 1973, en appliquant la technique de programmation dynamique, Godbole [God73] proposa (pour le problème MCOP) la première solution polynômiale. Elle nécessite $O(n^3)$ temps d'exécution et $O(n^2)$ espace mémoire. La structure et la complexité temporelle et spatiale de cette solution étant indépendante de la fonction de l'union U du problème MCOP, elle est devenue l'algorithme standard de résolution de tous les problèmes appartenant à la classe considérée. Dans la littérature cet algorithme est souvent appelé *algorithme générique séquentiel*.

Pour certains des problèmes de la classe abordée (comme MCOP, OBST et TCP), des solutions plus efficaces que l'algorithme séquentiel générique existent. Elles exploitent certaines propriétés de la fonction de l'union U, pour ces problèmes, pour améliorer les performances. Ces améliorations sont désignées dans la littérature par : *les accélérations*

en programmation dynamique (speed-up dynamic programming).

3.4.1 Algorithme générique séquentiel en $O(n^3)$ pour les problèmes de la classe considérée

Pour bien illustrer les propriétés des problèmes de la classe étudiée, on présente souvent dans la littérature les différentes étapes de la solution de Godbole à travers la résolution du problème d'Ordonnancement du Produit de Chaîne de Matrices. Nous aussi nous ferons de même. Comme nous l'avons dit plus haut, cette solution est générique. Elle n'est pas dédiée pour un seul problème de cette classe. Pour obtenir la version Godbole pour les autres problèmes de la même classe (section 3.3), il suffit de modifier les instructions implémentant la fonction de l'union U dans l'algorithme 1.

D'après la formulation de programmation dynamique du problème MCOP présenté à la section 3.3.2, la résolution d'un sous-problème $P_{i,j}$ représentant le sous-produit $M_i \times M_{i+1} \times \ldots \times M_j$ se fait suivant la récurrence 3.4 que nous reproduisons ci-dessous.

$$Cost(i,j) = \begin{cases} 0 & \text{, si } 1 \leq i = j \leq n \\ \min_{i \leq k < j} \{Cost(i,k) + Cost(k+1,j) + d_{i-1} \times d_k \times d_j\} & \text{, si } 1 \leq i < j \leq n \end{cases}$$

Dans cette équation, le coût optimal de traitement $Cost(i,j)$ est obtenu par addition des coûts optimaux de deux sous-produits $P_{i,k} = M_i \times M_{i+1} \times \ldots \times M_k$ et $P_{k+1,j} = M_{k+1} \times \ldots \times M_j$ et la valeur du coût de leur multiplication $U(i,k,j) = d_{i-1} \times d_k \times d_j$. Le but est de trouver parmi les $(j-i)$ décompositions possibles du produit $P_{i,j}$ en deux sous-produits, le couple de sous-produits $(P_{i,k}, P_{k+1,j})$ qui minimise $Cost(i,j)$, sachant que la résolution d'un problème d'ordre n (c'est-à-dire $P_{1,n}$) se réduit au calcul de $Cost(1,n)$.

Présentation de l'algorithme :

L'objectif de cet algorithme est de déterminer le bon ordre d'évaluation des sous-problèmes qui évite la redondance due au chevauchement. Cet ordre est régi par la relation de dépendance entre les différents sous-problèmes. En effet, comme expliqué dans la section 3.2, l'idée est de définir un DAG-multi-niveaux de sous-produits (sous-problèmes) et de les résoudre suivant une approche ascendante.

La récurrence 3.4 montre que le calcul du coût optimal d'un produit $P_{i,j}$ ne dépend jamais d'un produit de longueur supérieur à $(j-i)$. Les sous-produits peuvent donc être organisés selon leur longueur en n niveaux, le niveau N_d correspondant aux produits de longueur d tel que $N_d = P_{i,i+d-1}/1 \leq i \leq n-d+1$. Les produits de longueur d ne sont

traités qu'après le traitement de tous les produits de longueur m inférieures à d. Ceci donne lieu à un DAG-multi-niveaux à n niveaux. Dans ce DAG, le calcul du coût du sous-produit $P_{i,j}$ appartenant au niveau $(j - i + 1)$, dépend des coûts de calcul de deux sous-produits de chacun des niveaux précédents. La figure 3.6 donne la forme de ce DAG-multi-niveaux pour $n = 4$. Elle montre par exemple que le calcul de $Cost(1, 4)$ dépend de : $Cost(1, 3)$, $Cost(1, 2)$, $Cost(1, 1)$, $Cost(2, 4)$, $Cost(3, 4)$ et $Cost(4, 4)$.

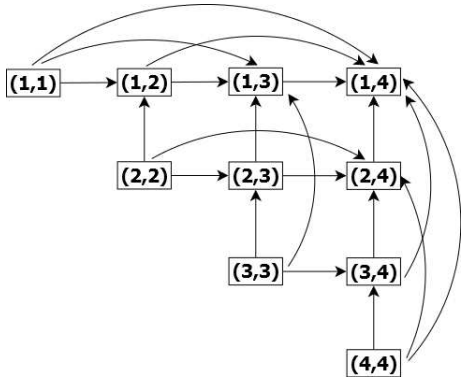


FIGURE 3.6 – La forme du DAG-multi-niveaux schématisant les dépendances entre les différents sous-problèmes, pour un problème MCOP d'ordre $n = 4$.

L'algorithme de Godbole résout les sous-problèmes du DAG-multi-niveaux, niveau par niveau, à partir du niveau 1 jusqu'au niveau n. Sa structure générale est donnée par l'algorithme 1. Dans cet algorithme, la diagonale N_d correspond au niveau N_d du DAG.

Algorithme 1 : Structure générale de l'algorithme de Godbole.

Entrées : Nombre de matrices et leurs dimensions
Sorties : Le coût du meilleur paranthésage

```
1 pour  i = 1 à n faire
2 |   Résoudre les sous-produits de la diagonale N_d ;
3 fin
```

L'évaluation des sous-problèmes d'un niveau N_d est effectuée via l'algorithme 2 (la procédure $EvaluerNiveau(N_d)$). Elle consiste à calculer, suivant la récurrence 3.4, le coût optimal de chaque sous-produit $P_{i,j}$ du niveau d, et le sauvegarder dans une table qui contient autant de cases que de sous-problèmes. C'est la table de programmation dynamique. Désignons cette table TP et désignons la table de trace (vu à la section 3.2.2) correspondante TT. Ainsi, $TP(i, j)$ contiendra (à la fin de l'algorithme) le coût optimal du parenthésage du produit $P_{i,j}$ et $TT(i, j)$ contiendra la valeur de k (position des parenthèses) correspondant au couple de sous-produits $(P_{i,k}, P_{k+1,j})$ qui minimise $Cost(i, j)$.

Les deux tables sus présentées ont la même forme que le DAG-multi-niveaux du problème à résoudre. Ce sont des matrices triangulaires supérieures. La figure 3.7 représente la table de programmation dynamique correspondante à $P_{1,4}$ (elle a la même forme que le DAG-multi-niveaux de la figure 3.6). Chaque diagonale (à partir de la diagonale centrale) sauvegarde les coûts d'un niveau de sous-produits. Autrement dit, la tâche de l'algorithme de Godbole consiste à remplir les cases de la table de programmation dynamique TP, diagonale par diagonale, à partir de la diagonale centrale et en allant vers la droite. A chaque fois qu'un coût est renseigné dans TP, la valeur de k correspondant est renseignée dans TT (lignes 6 et 7 de l'algorithme 2).

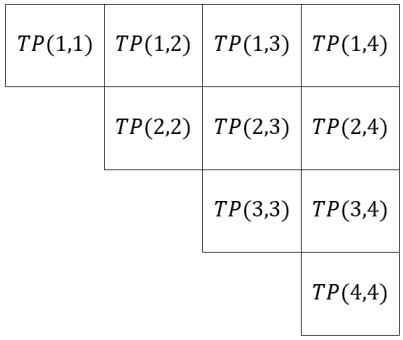


FIGURE 3.7 – Table de programmation dynamique pour un problème MCOP d'ordre $n = 4$.

Algorithme 2 : Procédure *EvaluerNiveau* (N_d) pour le problème MCOP

Entrées : Le niveau d à évaluer
Sorties : Les valeurs du coût optimal correspondantes aux nœuds du niveau d

```
1  pour  i = 1 à n − d + 1 faire
2  |  TP(i, i + d − 1) ← ∞ ;
3  |  pour  k = i à i + d − 2 faire
4  |  |  Temp ← TP(i, k) + TP(k + 1, i + d − 1) + d_{i−1} × d_k × d_{i+d−1} ;
5  |  |  si  Temp < TP(i, i + d − 1) alors
6  |  |  |  TP(i, i + d − 1) ← Temp ;
7  |  |  |  TT(i, i + d − 1) ← k ;
8  |  |  fin
9  |  fin
10 fin
```

Complexité de l'algorithme :

Considérons un problème MCOP d'ordre n.

- *Complexité en espace* : Comme la table de programmation dynamique TP est une matrice triangulaire supérieure d'ordre n, on peut dire, sachant qu'elle est similaire à la table de trace TT, que l'algorithme séquentiel de Godbole utilise $O(n^2)$ espace mémoire ;
- *Complexité en temps* : Comme il y a au plus n entrées dans une diagonale de la table TP (c'est-à-dire au plus n sous-problèmes dans une étape de l'algorithme) et que l'évaluation d'un sous-problème se fait dans le pire des cas en $O(n)$ opérations, l'algorithme 2 évalue une diagonale de TP en $O(n^2)$ temps. Puisqu'il y a n diagonales dans la table TP, l'algorithme séquentiel de Godbole (algorithme 1) résoud un problème MCOP d'ordre n en $O(n^3)$ temps d'exécution.

Quelques contraintes de parallélisation :

- Comme les sous-problèmes d'une même étape (diagonale de la table TP) sont tous indépendants, on peut les distribuer sur des processeurs et les évaluer en même temps ;
- La charge de calcul pour l'évaluation des sous-problèmes d'une étape i est plus petite que celle des sous-problèmes de l'étape $i+1$. En effet, l'évaluation d'un sous-problème d'une étape i dépend des couples de sous-problèmes de chacune des étapes précédentes. Ainsi, dans le processus de parallélisation, il faut tenir compte de ce déséquilibre de charge de calculs entre les étapes de l'algorithme séquentiel de Godbole (c'est-à-dire entre les diagonales de la table TP) qui peut causer un grand déséquilibre de charge entre les processeurs et ternir l'efficacité de l'algorithme parallèle.

Il existe des algorithmes séquentiels permettants de résoudre plus rapidement trois des problèmes de la classe abordée. Ils nécessitent $O(n^2)$ temps d'exécution et $O(n^2)$ espace mémoire pour un problème d'ordre n. Il s'agit de l'algorithme séquentiel de Knuth [Knu71] pour l'OBST et l'algorithme séquentiel de Yao [Yao82] pour les problèmes MCOP et TCP.

3.4.2 Algorithme séquentiel de Knuth

Dans cette section, nous présentons d'abord la version de l'algorithme séquentiel de Godbole pour l'OBST. Ensuite, nous montrons comment Knuth part de là pour dériver un algorithme en $O(n^2)$ temps d'exécution.

Solution générique séquentielle pour l'OBST (algorithme de Godbole)

La résolution du problème OBST se distingue de celle du problème MCOP, via l'algo-

rithme générique de Godbole, au niveau de la résolution des sous-problèmes d'une même étape. En effet, dans la variante destinée à l'OBST, la fonction de l'union est la fonction $w(i,j)$ (voir section 3.3.3). L'algorithme 3 donne la structure générale de l'algorithme de Godbole pour le problème OBST.

Algorithme 3 : Structure générale de l'algorithme séquentiel de Godbole pour un problème OBST d'ordre n.

Entrées : les $2n+1$ probabilités
Sorties : le coût du meilleur arbre binaire de recherche

```
1 Initialiser la table de programmation dynamique : q_i pour chaque élément TP(i,i)
  de la première diagonale et 0 pour le reste des cases ;
2 pour chaque diagonale d de 1 à n faire
3 |   pour chaque entrée TP(i,j) de la diagonale d faire
4 |   |   TP(i,j) ← min_{i<k≤j} {TP(i,k-1) + TP(k,j) + w(i,j)} ;
5 |   |   Cut(i,j) ← la valeur de k qui minimise TP(i,j) ;
6 |   fin
7 fin
```

La table $Cut(n,n)$, construite en même temps, et ayant la même structure que la table TP (ligne 5), est la table de trace. $Cut(i,j)$ donne la valeur de k qui minimise $TP(i,j)$. C'est le premier point donnant une décomposition optimale de l'arbre $obst(i,j)$ en deux sous-arbres.

La table $Cut(n,n)$ est très utile dans la solution séquentielle de Knuth. En effet, en observant la propriété, dite *propriété de monotonie*, que vérifie les éléments de la table $Cut(n,n)$, D. Knuth [Knu71] a réussi à déduire un algorithme dont la complexité temporelle et spatiale est en $O(n^2)$. Cet algorithme a le même principe que celui de Godbole.

Source de l'accélération de Knuth :

La propriété de monotonie que vérifie les éléments de la table $Cut(n,n)$ est définie comme suit :

$$i \leq i' \leq j \leq j' \Rightarrow Cut(i,j) \leq Cut(i',j') \tag{3.7}$$

Algorithme séquentiel :

Cette propriété transforme l'algorithme 3 comme ceci (algorithme 4) :

Il est facile de constater que l'accélération de Knuth provient de la boucle **Pour** interne de cet algorithme (ligne 4). Dans cette boucle, l'évaluation de tout une diagonale (la diagonale d) ne requiert que $O(n)$ opérations de comparaison au lieu de $O(n^2)$ comme

Algorithme 4 : Structure générale de l'algorithme de Knuth pour un problème OBST d'ordre n.

Entrées : les $2n+1$ probabilités
Sorties : le coût du meilleur arbre binaire de recherche

```
1 Initialiser la table de programmation dynamique : q_i pour chaque élément TP(i,i)
  de la première diagonale et 0 pour le reste des cases ;
2 pour chaque diagonale d de 1 à n faire
3 |   pour chaque entrée TP(i,j) de la diagonale d faire
4 |   |   TP(i,j) ← min_{Cut(i,j-1)≤k≤Cut(i+1,j)} {TP(i,k-1) + TP(k,j) + w(i,j)} ;
5 |   |   Cut(i,j) ← la valeur de k qui minimise TP(i,j) ;
6 |   fin
7 fin
```

pour la solution générique. En effet, lors du calcul d'un élément de la diagonale $(j-i+1)$, le temps de calcul est réduit de $(j-i)$ à $(Cut(i+1,j)-Cut(i,j-1))$. Cette accélération n'est pas régulière, car sur une même diagonale, le gain obtenu en nombre d'opérations de comparaison est différent d'un élément à l'autre. $((Cut(i+1,j)-Cut(i,j-1))$ ne dépend pas de n et peut être différent pour tous les couples (i,j)).

Contrainte de parallélisation :

Contrairement à la version de Godbole où la charge de calcul est la même pour chacun des éléments d'une même diagonale d (c'est-à-dire $O(d-1)$), dans cet algorithme, si les éléments d'une même diagonale sont distribués équitablement entre différents processeurs, rien ne garanti que ces processeurs auront la même charge de calcul. Ceci provoque un déséquilibre de charge ingérable entre les processeurs capable de rendre inutilisable l'accélération de Knuth. En conséquence, une nouvelle contrainte doit être prise en compte dans la conception d'algorithmes parallèles basés sur cette approche : *l'inégalité du nombre d'opérations nécessaire au calcul des valeurs des entrées d'une même diagonale dans la table de programmation dynamique.*

3.4.3 Algorithme séquentiel de Yao

Pour le problème MCOP (*Matrix Chain Ordering Problem*), Yao [Yao82] a proposé un algorithme qui fait aussi passer le temps d'exécution de $O(n^3)$ à $O(n^2)$. Au lieu de résoudre plus rapidement chaque sous-problème comme dans l'algorithme de Knuth, Yao se penche plutôt sur la diminution du nombre des sous-problèmes. Cette approche est basée sur l'*inégalité du quadrangle* observé sur un problème équivalent : le problème de

triangulation optimale d'un polygone convexe.

Il peut être démontré (voir [HS84]) qu'il existe une bijection entre les différents parenthésages possibles d'un produit de n matrices $M = M_1 \times \ldots \times M_i \times \ldots \times M_n$ (M_i étant de dimensions (d_{i-1}, d_i)) et les différentes triangulations possibles du polygone $P = \langle v_0, v_1, \ldots, v_n \rangle$ avec $w(v_i) = d_i$.

Pour ce faire, on considère un polygone convexe P de $n+1$ sommets $\langle v_0, v_1, \ldots, v_n \rangle$. Un segment $v_i v_{i+1}$,$(0 \leq i < n)$ correspond à la matrice M_{i+1}. Le segment $v_i v_j$,$(i < j-1)$ correspond au produit $M_i \times M_{i+1} \times \ldots \times M_j$ (voir figure 3.8). Les poids $\langle d_0, d_1, \ldots, d_n \rangle$ des $n+1$ sommets du polygone représentent les dimensions de n matrices $M_1 \times \ldots \times M_i \times \ldots \times M_n$ tel que la matrice M_i soit de dimensions (d_{i-1}, d_i). La figure 3.9 illustre cette correspondance pour un produit de quatre matrices $M_1 \times M_2 \times M_3 \times M_4$ de dimensions respectives (10,20), (20,30), (30,5) et (5,2), avec la triangulation du polygone $P = \langle v_0, v_1, v_2, v_3, v_4 \rangle$ tel que $w(v_0) = 10$, $w(v_1) = 20$, $w(v_2) = 30$, $w(v_3) = 5$ et $w(v_4) = 2$.

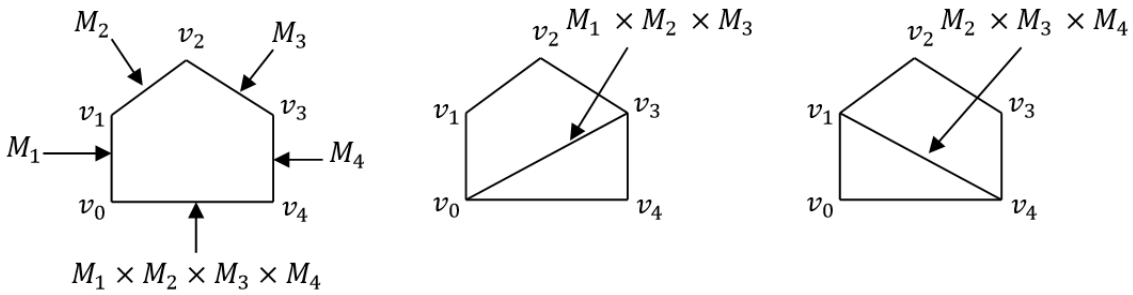


FIGURE 3.8 – Correspondances entre les arcs du polygone $P = \langle v_0, v_1, v_2, v_3, v_4 \rangle$ et les sous-chaînes du produit $M_1 \times M_2 \times M_3 \times M_4$.

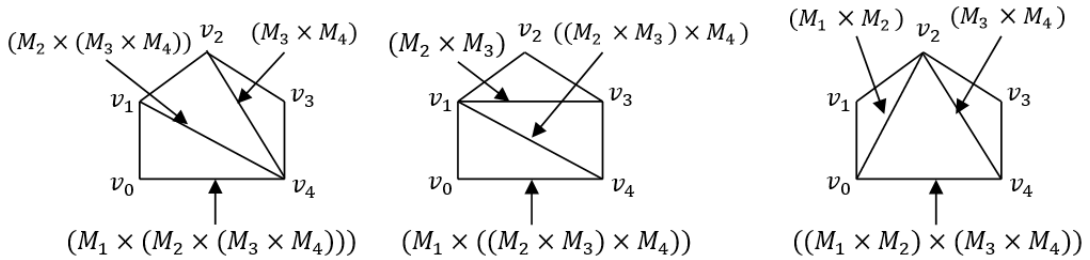


FIGURE 3.9 – Différentes façons de trianguler le polygone $P = \langle v_0, v_1, v_2, v_3, v_4 \rangle$ et leur correspondance pour le parenthésage de l'expression $M_1 \times M_2 \times M_3 \times M_4$.

En considérant que $w(v_i) = d_i$ comme indiqué ci-dessus, les récurrences 3.4 et 3.6 deviennent identiques. Ainsi, les problèmes TCP et MCOP ont une même formulation de programmation dynamique. Et donc, la résolution d'une instance du problème MCOP

peut se faire à partir de la résolution de l'instance équivalente du problème TCP et inversement.

Désormais nous allons considérer que chaque sommet v_i du polygone P a un poids d_i,$(0 \leq i \leq n)$, et que ces poids sont totalement ordonnés. Si plusieurs sommets ont le même poids, alors un ordre leur est imposé au début des traitements, et cet ordre est maintenu durant tout le processus de triangulation (c'est-à-dire dans tous les sous-polygones où ces nœuds apparaissent simultanément).
Dans la suite de cette section, nous désignerons selon le besoin, les nœuds d'un polygone soit par leur rang suivant cet ordre, c'est-à-dire w_j pour le sommet de rang j, soit par leur notation originale v_i (leur position dans le polygone, dans l'ordre des aiguilles d'une montre). Ainsi, le polygone $P = \langle v_0, v_1, v_2, v_3, v_4\rangle$ de la figure 3.9 dont les poids des sommets sont : $w(v_0) = 10$, $w(v_1) = 20$, $w(v_2) = 30$, $w(v_3) = 5$ et $w(v_4) = 2$ peut être représenté par $\langle w_3, w_4, w_5, w_2, w_1\rangle$.

Dans le processus de triangulation, nous utilisons indifféremment les mots *sous-polygone* et *partition*. La partition $P[v_i, v_j]$ du polygone P correspond au sous-polygone constitué des nœuds entre v_i et v_j de P, suivant le sens des aiguilles d'une montre. Par exemple, dans le polygone de la figure 3.9, $P[v_0, v_2] = \langle v_0, v_1, v_2\rangle$, $P[v_2, v_0] = \langle v_2, v_3, v_4, v_0\rangle$ et $P[v_3, v_1] = \langle v_3, v_4, v_0, v_1\rangle$. Nous utiliserons les termes *segment coté* et *segment diagonal* pour désigner respectivement les segments originaux du polygone (qui existaient avant son partitionnement) et les segments rajoutés durant le processus de partitionnement.

Lemme 1 *([Yao82]) : Étant donnée un polygone P de n sommets, il existe une solution optimale au problème TCP ayant les propriétés suivantes :*

1. *w_1 et w_2 sont adjacents, c'est-à-dire reliés soit par un des segments coté, soit par un des segments diagonal ;*
2. *w_1 et w_3 sont adjacents ;*
3. *Si w_1w_2 et w_1w_3 sont des segments voisins dans P alors, soit le segment w_2w_3, soit le segment w_1w_4 est dans la solution optimale comme segment diagonal.*

Algorithme récursif :

Pour un problème MCOP/TCP d'ordre n, l'algorithme récursif de Yao [Yao82] utilise le lemme 1 pour rechercher la triangulation optimale d'un polygone. Cette triangulation est donnée par l'algorithme 5. $Partition[P]$ désigne le tableau contenant la solution optimale pour un sous-polygone P.

Algorithme 5 : Algorithme récursif de Yao pour un problème MCOP d'ordre n (PROCEDURE $Partition[P]$).

Entrées : Nombre de matrices et leurs dimensions
Sorties : Le coût du meilleur paranthésage

```
1  début
2  |  si |P| ≤ 2 alors
3  |  |  retourner Ø ;
4  |  sinon si P est un triangle alors
5  |  |  retourner P ;
6  |  sinon si w1 et w2 ne sont pas adjacents alors
7  |  |  retourner Partition[1,2] ∪ Partition[2,1] ;
8  |  sinon si w1 et w3 ne sont pas adjacents alors
9  |  |  retourner Partition[1,3] ∪ Partition[3,1] ;
10 |  sinon
11 |  |  retourner le meilleur entre {Partition[2,3] ∪ Partition[3,2],
   |  |  Partition[1,4] ∪ Partition[4,1]} ;
12 |  fin
13 fin
```

Dans le pire des cas, cet algorithme est exponentiel en n ($2^{n-4} + 2$). Pourtant, on montre que le nombre de sous-problèmes (sous-polygones) distincts pouvant apparaître dans ce processus de triangulation récursif est en $O(n^2)$. En identifiant ces $O(n^2)$ sous-polygones distincts et en étudiant leurs interdépendances, Yao [Yao82] a pu ordonner leur évaluation de sorte qu'au moment de la triangulation d'un (sous-) polygone $P[v_i, v_j]$, les triangulations des sous-polygones dont il dépend sont déjà effectués. Ainsi il a pu dériver, pour le problème MCOP, un algorithme de programmation dynamique qui a une complexité temporelle et spatiale en $O(n^2)$.

Algorithme de programmation dynamique :

L'algorithme séquentiel de Yao est divisé en deux phases : la première consiste à identifier et à organiser de façon ascendante les $O(n^2)$ sous-problèmes en un DAG-multi-niveaux. La seconde phase consiste en la résolution proprement dite du problème. Dans cette seconde phase, les sous-problèmes sont résolus suivant l'ordre fourni dans la phase 1.

Phase 1 :

Le DAG-multi-niveaux produit dans cette phase est une union de deux arbres binaires dont les racines sont désignés par les sous-polygones w_1w_2 et w_2w_1. Les segments diagonaux w_iw_j considérés sont ceux pour lesquels tout sommet w_k de $P[v_i, v_j]$ satisfait

$k \geq \max\{i, j\}$. w_1 et w_2 sont les seules sommets pour lesquelles on a deux segments w_1w_2 et w_2w_1 valides. Ce DAG est structuré comme suit :

- Chacune des n feuilles correspond à un ensemble de triangles $\{(w_i, w_j, w_k)/w_k < \min\{w_i, w_j\}\}$ qui ont en commun l'un des n segments coté du polygone original, le segment w_iw_j. Une feuille est désignée par le segment commun à ses triangles. Nous désignons par (k, l) une feuille dont le segment commun à ses triangles est w_kw_l ;
- Chaque nœud interne correspond à un ensemble de sous-polygones dont l'un, appelé *« polygone de base du nœud»*, est de type $Q = P[w_i, w_j]$ (ou bien $P[w_j, w_i]$), et les autres sont de type (w, Q) (ou (Q, w)), où w est un nœud du polygone original tel que $w < w_i$. Un nœud interne est désigné par le polygone de base lui correspondant. Nous désignons par (i, j) un nœud dont le polygone de base est $P[w_i, w_j]$;
- Les deux fils (respectivement fils gauche et fils droit) d'un nœud interne (i, j) sont les nœuds (i, k) et (k, j) tel que k est le plus petit indexe (en dehors de i et j) dans $P[w_i, w_j]$.

Les nœuds de ce graphe sont identifiés et générés dans un ordre proche de la représentation en ordre postfixe. La figure 3.10 montre un exemple de polygone avec le DAG-multi-niveaux correspondant. Les nœuds du graphe sont générés dans l'ordre suivant : (7,10) ; (10,5) ; (4,7) ; (7,5) ; (4,5) ; (5,2) ; (1,4) ; (4,2) ; (2,9) ; (9,6) ; (6,11) ; (11,8) ; (6,8) ; (8,3) ; (2,6) ; (6,3) ; (2,3) ; (3,1) ; (1,2) ; (2,1).

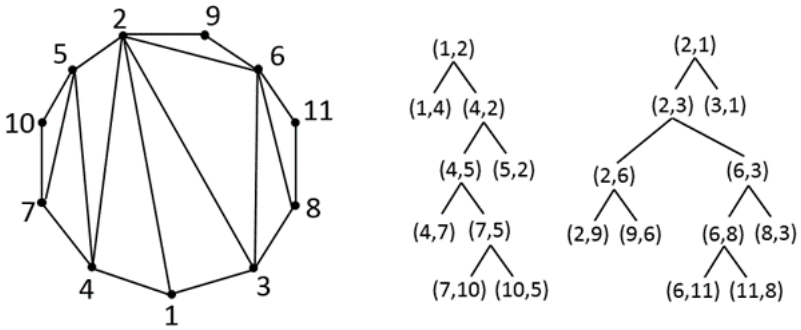


FIGURE 3.10 – Un polygone P de 11 sommets et le DAG-multi-niveaux correspondant. Les sous-polygones correspondants au nœud $(4, 5)$ sont : $P[w_4, w_5]$, qui est le sous-polygone de base du nœud, et $\{(w_3, P[w_4, w_5])$, $(w_2, P[w_4, w_5])$, $(w_1, P[w_4, w_5])\}$. Les triangles correspondants à la feuille $(8, 3)$ sont : $\langle w_2, w_8, w_3\rangle$ et $\langle w_1, w_8, w_3\rangle$.

Un sous-polygone Q de P est appelé *cône* si $Q = P[w_i, w_j] \cup w_iw_jw_k$ où $b = w_iw_j$ est un segment diagonal de P et $k \leq \min\{i, j\}$. On désigne aussi le cône Q par (b, w_k).

Dans le graphe de la figure 3.10, les nœuds sont organisés de telle sorte que :

Lemme 2 *([Yao82]) : la recherche de la triangulation optimale de chaque sous-polygone*

correspondant à un nœud donné de l'arbre se déduit uniquement de la triangulation optimale des sous-polygones correspondant à ses deux fils.

Phase 2 :

La triangulation est effectuée dans cette phase. L'algorithme utilisé parcourt l'arbre binaire obtenu dans la phase 1 des feuilles vers la racine en extrayant les résultats associés à chaque nœud. Le résultat obtenu à la racine de l'arbre (union des deux arbres binaires constituants le graphe) constitue la solution au problème.

Cette triangulation est faite via l'algorithme 6. Dans cet algorithme, $Partition[Q]$ est un tableau contenant la solution optimale pour le cône Q. La boucle **for** externe (ligne 2) parcourt l'ensemble des nœuds du graphe dans l'ordre généré dans la phase 1. Tandis que les boucles **for** internes (lignes 4 et 13) parcourent l'ensemble des nœuds dans l'ordre décroissant [13].

La complexité en temps de l'algorithme séquentiel de Yao est $O(n^2)$, $O(n)$ pour la phase 1 et $O(n^2)$ pour la phase 2. En effet, il y a au plus $2n$ nœuds dans le DAG-multi-niveaux (il y a n nœuds correspondants aux segments coté, et $n-1$ correspondants aux segments diagonaux) et pour chaque nœud, il y a au plus $O(n)$ cônes.

Contrainte de parallélisation :

Contrairement aux DAG-multi-niveaux utilisés par les algorithmes de Godbole et de Knuth, qui ont la même forme quelque soit l'instance du problème à résoudre, le DAG-multi-niveaux utilisé dans l'algorithme de Yao aura des formes différentes pour deux produits de matrices de même longueur, mais avec des matrices de dimensions différentes. Ainsi, dans la parallélisation de l'algorithme séquentiel de Yao, puisqu'on ne connaît pas à l'avance la forme du DAG-multi-niveaux à considérer, le regroupement de sous-problèmes et leur allocation aux processeurs ne peut pas se faire de manière *statique*. Ces deux tâches doivent se réaliser de manière *dynamique* en fonction de la forme du DAG-multi-niveaux correspondant à l'instance du problème à résoudre.

3.4.4 Une implémentation de l'algorithme séquentiel de Yao

Dans la présentation de son algorithme, Yao ne dit pas comment effectuer une tâche qui peut améliorer significativement ses performances. En effet, avec cet algorithme, lors de l'évaluation d'un nœud w_iw_j du DAG utilisé, l'on doit considérer les sommets du

13. Une relation d'ordre $\prec$ est défini sur l'ensemble des segments tel que : $w'_iw'_j \prec w_iw_j$ signifie que $P[v'_i, v'_j] \subseteq P[v_i, v_j]$.

Algorithme 6 : Algorithme de programmation dynamique de Yao pour un problème MCOP d'ordre n (PROCEDURE $DP - Partition[P]$).

Entrées : Nombre de matrices et leurs dimensions
Sorties : Le coût du meilleur paranthésage

```
1  début
2      pour chaque b = w_i w_j ∈ B faire /* B est l'ensemble ordonné des
       nœuds du graphe et on suppose i < j.                              */
3          si b est une feuille alors
4              pour tous les cônes Q = (b, w_k) tel que k ≤ i faire
5                  si w_k = w_i alors
6                      Partition[Q] ← ∅;                    /* Q = w_i w_j */
7                  sinon
8                      Partition[Q] ← Q ;                   /* Q = w_i w_j w_k */
9                  fin
10             fin
11         fin
12         si b n'est pas une feuille alors
13             pour tous les cônes Q = (b, w_k) tel que k ≤ i faire
14                 si w_k = w_i alors
15                     Partition[Q] ← Partition[(filsGauche(b), w_i)]∪
                       Partition[(filsDroit(b), w_i)] ;
16                 sinon
17                     Partition[Q] ← le meilleur entre {Partition[(b, w_i)] ∪ w_i w_j w_k,
                       Partition[(filsGauche(b), w_k)] ∪ Partition[(filsDroit(b), w_k)] ;
18                 fin
19             fin
20         sinon
21         fin
22     fin
23 fin
```

polygone dont les poids sont inférieurs ou égaux à w_i. Cette évaluation est faite à l'aide des poids de sommets, et dans l'ordre décroissant de ces poids. Yao ne dit ni comment identifier efficacement ces sommets, ni comment les obtenir dans l'ordre décroissant. Dans cette section, nous apportons un élément de réponse à ces questions.

Prétraitement additionnel pour l'algorithme séquentiel de Yao :

Lors de l'évaluation du nœud w_iw_j du DAG, les sommets (du polygone) candidats à être utilisés, sont situés entre les sommets w_j et w_i. Parmi eux, certain ont des poids supérieurs à w_i. Ainsi, la façon la plus naturelle de rechercher les sommets à considérer est de parcourir entièrement cet ensemble de sommets et comparer leur poids à w_i. De plus, ces sommets doivent être utilisés dans l'ordre décroissant de leur poids pourtant ils ne sont pas triés.

Pour résoudre ce problème, nous proposons un prétraitement additionnel à l'algorithme séquentiel de Yao : *le tri des poids des sommets du polygone avant l'évaluation des nœuds du DAG.* Ainsi, l'évaluation du nœud w_iw_j du DAG commence avec le sommet w_i qui est le sommet ayant le plus grand poids à considérer, et continue dans l'ordre décroissant dans l'ensemble trié des poids des sommets du polygone jusqu'au sommet de plus faible poids.

Exemple : Dans la figure 3.10, pour évaluer le nœud $(2, 3)$, seuls les sommets 1 et 2 doivent être considérés. Pourtant, comme l'ensemble $\{1, 4, 7, 10, 5, 2\}$ des sommets candidats n'est pas trié à l'avance, il doit être entièrement parcouru pour n'en retenir que deux ($\{1, 2\}$). Ceci induit une perte de temps énorme car pour le cas d'espèce *quatre* des *six* sommets candidats ne sont pas utiles.

De cette façon, lors de l'évaluation d'un nœud w_iw_j du DAG, l'ensemble des sommets dont les poids sont inférieurs ou égaux à w_i sont facilement identifiés.

Résultats expérimentaux :

Nous comparons les temps d'exécution de trois algorithmes utilisés. Les mêmes instances de problèmes sont résolues avec les différents algorithmes. Chaque courbe représente le temps d'exécution en fonction de la taille du problème et cette taille est comprise dans l'ensemble $\{300, 600, \ldots, 3000, 3600, 4200, 5000, \ldots, 16000\}$.

Terminologie 2 *Les courbes correspondent aux algorithmes et sont désignés comme suit :*

- *la courbe désignée « Godbole » montre le temps d'exécution pour l'algorithme générique séquentiel ;*
- *la courbe désignée « Yao » montre le temps d'exécution pour l'algorithme séquentiel de Yao dans lequel notre prétraitement additionnel n'est pas appliqué ;*

– *la courbe désignée « Yao_Sort » montre le temps d'exécution pour l'algorithme séquentiel de Yao avec notre prétraitement additionnel (le tri des poids des sommets du polygone est effectué à l'aide du QuickSort).*

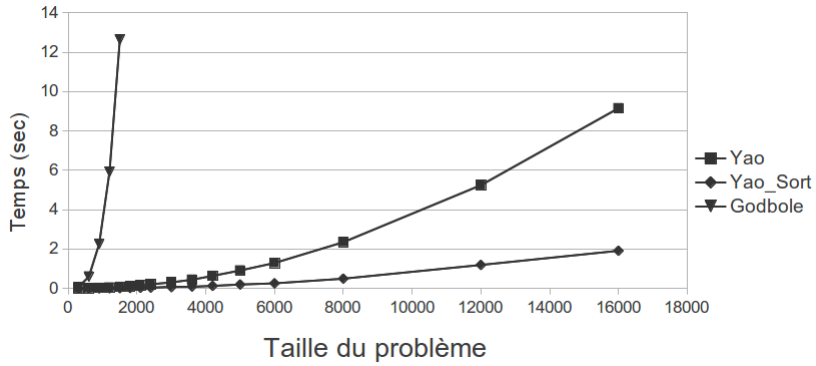


FIGURE 3.11 – Temps d'exécution de quelques algorithmes séquentiels en fonction de la taille du problème.

La figure 3.11 montre que notre proposition a un impact significatif sur le temps d'exécution de l'algorithme. Le tri des poids des sommets du polygone ajoute une procédure à l'algorithme, mais permet de gagner beaucoup de temps au moment de l'évaluation d'un nœud du DAG.

Remarque 2 *Ce prétraitement ne change pas la complexité temporelle de l'algorithme séquentiel de Yao, mais le rend plus efficace. Il ne complexifie pas sa parallélisation.*

3.5 Synthèse

La programmation dynamique permet de résoudre des problèmes respectant deux critères : le critère d'optimalité et le critère de chevauchement de sous-structures. Plusieurs algorithmes de programmation dynamique permettent de résoudre les problèmes de la classe qui nous intéresse, et certains d'entre eux sont appelés *« algorithmes séquentiel accélérés »* en vertu de leur temps d'exécution bien meilleur comparé à celui de *« l'algorithme générique séquentiel »*. En générale, tous ces algorithmes résolvent les problèmes considérés en évaluant de manière ascendante les nœuds (représentant les sous-problèmes) des DAG-multi-niveaux correspondant. Tel est le cas pour les algorithmes que nous avons présentés. Chacun de ces algorithmes séquentiels lève des contraintes de parallélisation provenant de la structure du DAG-multi-niveaux utilisé et des dépendances entre les sous-problèmes à résoudre (dépendance entre les nœuds du DAG). Ces contraintes doivent être

impérativement prises en compte pour avoir une chance d'obtenir de solutions parallèles efficientes.

Nous avons proposé un prétraitement supplémentaire à l'algorithme séquentiel de Yao : le tri des poids des sommets du polygone à trianguler. Ce prétraitement permet d'améliorer ses performances sans changer sa complexité temporelle. Il ne complexifie pas sa parallélisation.

Chapitre 4

Algorithmes CGM génériques pour les problèmes de la classe abordée

Sommaire

4.1 Introduction

Dans ce chapitre, nous présentons deux algorithmes parallèles BSP/CGM génériques, qui résolvent tous les problèmes de programmation dynamique appartenant à la classe abordée (OBST, MPP, TCP, etc.). Lors des calculs locaux, ces algorithmes utilisent une variante de l'algorithme générique séquentiel vu à la section 3.4.1. Il s'agit d'une traduction de cet algorithme générique séquentiel en un algorithme de plus court chemin one-to-all dans un graphe dynamique.

Dans la deuxième partie de ce chapitre, nous présentons quelques spécificités typiques à l'ensemble des problèmes que nous voulons paralléliser. Ensuite, nous montrons comment l'algorithme générique séquentiel peut être traduit en un algorithme de plus court

chemin one-to-all dans un graphe dynamique. Dans la quatrième partie, nous présentons l'algorithme BSP/CGM générique introduit par Kechid-Myoupo [Kec09]. Il permet de résoudre un problème d'ordre n (de la classe abordée) en $O(n^3/p)$ temps de calcul, avec $O(p)$ rondes de communication sur p processeurs. Cet algorithme présente un inconvénient majeur : *le déséquilibre de charge de calcul entre les processeurs utilisés*. Pour pallier à la perte de performance causée par ce déséquilibre de charge, nous proposons un mécanisme de rééquilibrage de charge entre les processeurs. Dans la dernière partie, nous proposons un algorithme BSP/CGM qui, dans les mêmes conditions nécessite $O(n^3/p)$ temps d'exécution, et fait passer le nombre de rondes de communication à $\lceil (2p)^{1/2} \rceil$.

4.2 Programmation dynamique parallèle

Pour répondre aux besoins de performance exprimés par une large variété de problèmes de programmation dynamique, de nombreuses solutions parallèles ont été proposées suivant différents modèles de calcul parallèle. Sans être exhaustif, nous pouvons citer : [ECG93, Gen96, GGKK03, GP91, GP94, IPS88, LT88, Myo91, Myo92, Myo93, Ryt88, TG95]. Cependant, ces algorithmes parallèles sont, soit non portables [1], soit inadaptés aux machines parallèles actuelles [2], soit non extensibles [3]. Le modèle CGM se présente comme un modèle idoine pour palier à cet ensemble d'inconvénients, car il permet comme nous l'avons vu au chapitre 2, de surmonter le compromis Efficacité-Portabilité tout en assurant une extensibilité des algorithmes.

La difficulté de parallélisation des algorithmes de programmation dynamique vient de la nature séquentielle des traitements entre les niveaux du DAG-multi-niveaux utilisé. Plus il y a des dépendances entre les différents niveaux de ce DAG (c'est-à-dire entre les sous-problèmes de niveaux différents), plus il est difficile de concevoir un algorithme parallèle pour le, ou les problèmes correspondant(s) (ceux modélisés par le DAG). En effet, dans ce cas, les sous-problèmes d'un même niveau ont de plus en plus de sous-problèmes en commun. Ce qui augmente les besoins de communication et rend plus complexe l'organisation des calculs. Li et Wah [LW85] ont proposé une classification des problèmes de programmation dynamique, permettant de mettre en évidence les difficultés et contraintes de parallélisation des différents problèmes. Cette classification est basée sur deux critères, exprimant *le type (intensité, diversité) de la dépendance* dans les DAG-multi-niveaux des

1. C'est-à-dire conçues suivant des modèles trop réalistes tels que les modèles systolique et l'hypercube.
2. C'est-à-dire conçu par exemple suivant un modèle trop abstrait comme le modèle PRAM.
3. Subissent une forte dégradation des performances avec l'accroissement de la taille des données en entrée.

problèmes.

La relation de dépendance entre les sous-problèmes :

Suivant l'intensité de la dépendance entre les sous-problèmes des différents niveaux du DAG-multi-niveaux, on peut distinguer deux classes de problèmes de programmation dynamique :

1. **La classe serial** : elle est constituée des problèmes de programmation dynamique pour lesquels la solution d'un sous-problème à un niveau N donné, dépend uniquement des solutions des sous-problèmes du niveau $N-1$ dans le DAG.

   Exemple : L'équation 4.1 décrit le problème PCC du calcul du plus court chemin (all-to-all) dans un graphe orienté pondéré suivant l'algorithme de Floyd. Dans le DAG-multi-niveaux correspondant, les problèmes ayant le même « indice exposant » sont regroupés dans le même niveau. Ainsi, le calcul de $p_{i,j}^{k}$ ne nécessite que les solutions des sous-problèmes $p_{i,j}^{k-1}$, $p_{i,k}^{k-1}$ et $p_{k,j}^{k-1}$ du niveau immédiatement précédent.

$$p_{i,j}^{k} = \begin{cases} C_{i,j} & \text{, si } k = 0 \\ \min\left\{p_{i,j}^{k-1}, p_{i,k}^{k-1} + p_{k,j}^{k-1}\right\} & \text{, si } 1 \leq k \leq n-1 \end{cases} \tag{4.1}$$

2. **La classe non-serial** : elle est constituée des problèmes de programmation dynamique pour lesquels, la solution d'un sous-problème à un niveau N donné, dépend des solutions des sous-problèmes de plusieurs niveaux précédents (inférieurs à N).

   Exemple : Le problème LCS (*Longest Common Subsequence*) de recherche de la plus longue sous-séquence commune de deux chaînes données, appartient à la classe *non-serial*. Dans le DAG-multi-niveaux correspondant, un sous-problème (i, j) appartient au niveau $(i+j-1)$. On observe dans l'équation fonctionnelle 4.2 que le calcul de la solution optimale d'un sous-problème (i, j) dépend de celle du sous-problème $(i-1, j-1)$ appartenant au niveau $(j+i-3)$, et de celles des sous-problèmes $(i, j-1)$ et $(i-1, j)$ appartenant au niveau $(j+i-2)$. Dans cette équation, la chaîne constituée des i premiers éléments d'une chaîne $X = \langle x_1, x_2, \ldots, x_n\rangle$ est désignée par $X_i, (1 \leq i \leq n)$. $L(i, j)$ désigne la longueur de la plus longue sous-séquence commune entre deux sous-chaînes X_i et Y_j.

$$L(i,j) = \begin{cases} 0 & \text{, si } i = 0 \text{ ou } j = 0 \\ L(i-1,j-1)+1 & \text{, si } i,j \geq 1 \text{ et } x_i = y_j \\ \max\{L(i,j-1), L(i-1,j)\} & \text{, si } i,j \geq 1 \text{ et } x_i \neq y_j \end{cases} \quad (4.2)$$

Le nombre de termes récursifs :

Suivant le nombre de termes récursifs (sous-problèmes) dans la (les) fonction(s) de composition de l'équation fonctionnelle décrivant le problème, on distingue deux classes de problèmes de programmation dynamique :

1. **La classe monadique** qui est constituée des problèmes de programmation dynamique pour lesquels le nombre de termes récursifs est égal à un.
   Exemple : Le problème de LCS appartient à la classe monadique. En effet, dans l'équation fonctionnelle 4.2 qui décrit sa solution, la fonction de composition ne contient qu'un seul terme récursif. Dans toutes les alternatives du membre droit de cette équation, l'évaluation du sous-problème L(i,j) ne dépend que d'un seul sous-problème (terme récursif) à la fois : soit $L(i-1,j-1)+1$, soit $L(i,j-1)$, soit $L(i-1,j)$.
2. **La classe polyadique** qui est constituée des problèmes de programmation dynamique pour lesquels le nombre de termes récursifs dans l'équation fonctionnelle est supérieur à 1.
   Exemple : Dans l'équation fonctionnelle 4.1 décrivant le problème PCC, pour calculer $p^k_{i,j}$, pour $(k > 0)$, la fonction de composition utilisée dans la seconde alternative du membre de droite et pour le deuxième élément de l'ensemble, fait intervenir deux termes récursifs : $p^{k-1}_{i,k}$ et $p^{k-1}_{k,j}$. Ce problème appartient donc à la classe polyadique.

Suivant le type de la dépendance entre les sous-problèmes dans le DAG-multi-niveaux, la classification de Li et Wah permet d'avoir une idée générale sur la complexité de parallélisation des problèmes de programmation dynamique : *les algorithmes parallèles pour les problèmes de la classe polyadique non-serial sont plus difficiles à concevoir* [Gen96].

Les problèmes de programmation dynamique dont il existe une solution suivant le modèle BSP/CGM ne sont pas très nombreux. À titre d'exemples, nous pouvons citer les travaux de Alves et al ([ACD+02a, ACD02b, ACDS03]) sur les problèmes d'édition et de similarité de chaînes. En s'inspirant du modèle systolique, Garcia et al [GMS03] ont résolu les problèmes LCS et LIS (*Longest Increasing Subsequence*). Krusche et Tiskin [KT06] ont proposé une solution pour le problème LLCS (*Length of LCS*) qui se base sur la

méthode utilisée dans ([ACD+02a, ACD02b, ACDS03]) avec l'utilisation d'un algorithme bit-parallèle pour les calculs séquentiels. Par ailleurs, dans ([ACS05, ACS06]), les auteurs ont défini et résolu un problème d'édition et de similarité de chaînes plus générique dit « ALCS problem » (*All-substrings Longest Common Subsequence problem*).

Aucun de ces problèmes n'appartient à la classe *polyadique non-serial*. Ces problèmes sont de la classe *monadique à faible dépendance*, et la résolution des sous-problèmes d'un niveau donné, nécessite au plus les solutions des sous-problèmes des deux niveaux immédiatement précédents.
La classe de problèmes étudiée dans cette thèse est une classe de problèmes de programmation dynamique de type *polyadique non-serial* dans lesquels il y a :

- **Une forte dépendance entre les sous-problèmes** : Le calcul des solutions des sous-problèmes d'un niveau donné nécessite celles de tous les niveaux précédents. La figure 4.1 présente le DAG correspondant aux problèmes de cette classe pour $n = 9$. Cette forte dépendance complique la conception de solutions BSP/CGM. Il est plus difficile d'effectuer la division des sous-problèmes et les schémas de communication entre les processeurs qui les hébergent sont plus complexes.
- **Une irrégularité de charge de calcul** : La taille de calcul nécessaire pour l'évaluation d'un sous-problème change d'un niveau à l'autre. Dans certains cas (comme pour la solution séquentielle de Knuth), cette taille change même pour les sous-problèmes d'un même niveau.

Quelques algorithmes parallèles sur le modèle BSP/CGM ont été conçus pour les problèmes de la classe abordée dans cette thèse. Par exemple, pour le problème MCOP, l'algorithme de Kechid et Myoupo [Kec09] basé sur la solution générique séquentielle nécessite $O(p)$ rondes de communication et $O(n^3/p)$ temps de calcul sur p processeurs. Basé sur l'accélération de Yao leur algorithme nécessite $O(p)$ rondes de communication avec $O(n^2/p)$ temps de calcul local. Pour le problème OBST, ils ont proposé une solution basée sur l'accélération de Knuth qui nécessite $O(p)$ rondes de communication avec $O(n^2/p)$ temps de calcul local.
L'inconvénient majeur de ces algorithmes est que, d'une part les charges des processeurs ne sont pas équilibrés, et d'autre part, le nombre de sous-problèmes d'un processeur n'est pas borné.

L'algorithme CGM de Dilson et Marco [HS12], pour le problème MCOP, nécessite $O(1)$ rondes de communication avec $O(n^3/p^3)$ temps de calcul local. Le problème qu'il présente est le temps de latence élevé des processeurs, à cause du séquencement des différents traitements à effectuer et du temps pris par ces traitements. Par exemple, la division

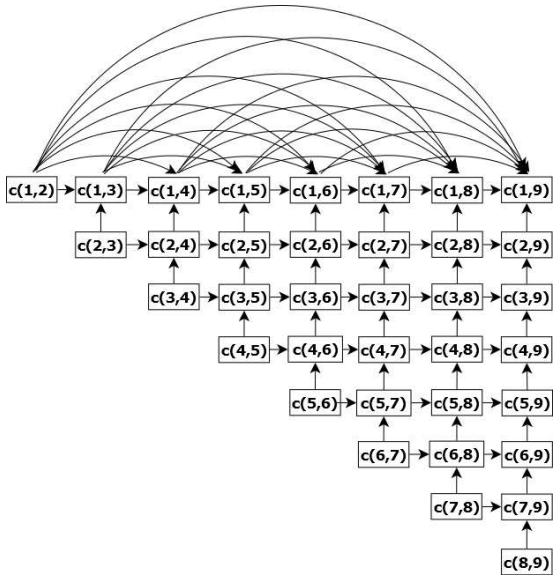


FIGURE 4.1 – DAG correspondant aux problèmes de la classe considérée avec $n = 9$. Chaque diagonale représente un niveau. Les dépendances entre les n niveaux sont représentées par des arêtes verticales et horizontales. Chaque nœud reçoit une arête (i.e. des données) de chacun des nœuds des niveaux précédents qui sont dans sa ligne ou sa colonne. Pour permettre la lisibilité, nous n'avons affiché que les dépendances horizontales de la première ligne ; $c(i,j) = Cost(i,j)$.

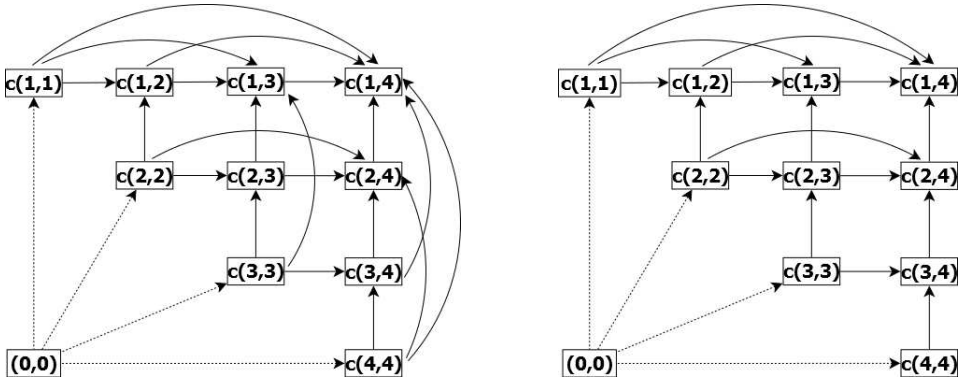
des tâches et la distribution des données sur les processeurs nécessitent $O(n) + O(n \ln n)$ calculs locaux pour un problème d'ordre n. L'algorithme séquentiel utilisé est celui de Godbole, donc, pas le plus performant.

4.3 Algorithme de plus court chemin one-to-all dans le graphe dynamique

Les problèmes de programmation dynamique sont parfois résolus via des problèmes de plus courts chemins sur leur DAG-multi-niveaux [Gen96]. En effet, Bradford [Bra94] a proposé un modèle graphique qu'il a baptisé *Graphe Dynamique* pour la résolution du problème MCOP. Un graphe dynamique D est un DAG pondéré [4].
Bradford a montré qu'à tout problème de parenthésage optimal d'une chaîne de n matrices, correspond un graphe dynamique D_n, où le coût du plus court chemin partant d'un sommet virtuel ajouté $(0,0)$ vers un sommet (i,j) quelconque, noté $SP(i,j)$, est égal à $Cost(i,j)$. Ainsi, le plus court chemin du sommet (0,0) vers le sommet $(1,n)$ dans D_n ré-

4. C'est-à-dire qu'à chaque arc de ce graphe est associé un poids.

sout $Cost(1, n)$ [5]. Le graphe dynamique D_4, correspondant à un problème de parenthésage optimal d'un produit de 4 matrices est représenté dans la figure 4.2a.



(a) Graphe dynamique D_4 (b) Graphe dynamique D'_4

FIGURE 4.2 – Les graphes dynamiques D_4 et D'_4 pour un produit de 4 matrices.

Comme pour le graphe des tâches de la figure 3.6 à la section 3.4.1, nous référençons chaque sommet du graphe dynamique D_n par le couple d'entiers $(i, j)/1 \leq i \leq j \leq n$, qui indique sa position dans le graphe. Nous disons que deux sommets (i, j) et (k, m) sont sur la même ligne (respectivement sur la même colonne) si $i = k$ (respectivement $j = m$). Ils sont sur la même diagonale $(j - i + 1)$ si $(j - i) = (m - k)$. D_n contient n diagonales de sommets, n lignes de sommets et n colonnes de sommets.

Le graphe dynamique D_n est défini par un ensemble de sommets $S = \{(i, j) : 1 \leq i \leq j \leq n\}$ $\cup \{(0, 0)\}$, et un ensemble d'arcs A construit à partir des deux sous-ensembles d'arcs suivants :

1. $\{(i, j) \rightarrow (i, j + 1) : 1 \leq i \leq j < n\} \cup \{(i, j) \uparrow (i - 1, j) : 1 < i \leq j \leq n\} \cup$ $\{(0, 0) \nearrow (i, i) : 1 \leq i \leq n\}$; dit ensemble d'arcs unitaires et,
2. $\{(i, j) \Rightarrow (i, t) : 1 \leq i < j < t \leq n\} \cup \{(s, t) \Uparrow (i, t) : 1 \leq i < s < t \leq n\}$; dit ensemble de sauts.

Dans le graphe D_n, chaque chemin entre le nœud $(0, 0)$ et un nœud (i, j) correspond à l'un des parenthésages possibles du sous-produit $P_{i,j}$ et chaque arc [6] de ce chemin définit une partie du parenthésage (quatre parenthèses) de $P_{i,j}$. C'est le plus court chemin qui correspond au parenthésage optimal.

5. Qui est, comme vu à la section 3.3.2, la solution du problème à résoudre.
6. À l'exception des arcs unitaires diagonaux.

La longueur d'un chemin vers le sommet (s, m) est définie par la somme des poids des arcs le constituant. $w((s, m), (f, l))$ désigne le poids d'un arc partant du sommet (s, m) au sommet (f, l). Ces poids sont évalués comme suit :

- Le poids des arcs unitaires constitué de l'ensemble $\{(0, 0) \nearrow (i, i) : 1 \leq i \leq n\}$ est nul : $w((0, 0), (i, i)) = 0$;
- L'arc unitaire $(i, j) \rightarrow (i, j+1)$ définit les parenthèses suivantes : $((M_i, \ldots, M_j)M_{j+1})$ et pèse : $w((i, j), (i, j + 1)) = U(i, j, j + 1) = d_{i-1} \times d_j \times d_{j+1}$;
- L'arc unitaire $(i, j) \uparrow (i-1, j)$ définit les parenthèses suivantes : $(M_{i-1}(M_i, \ldots, M_j))$ et pèse : $w((i, j), (i - 1, j)) = U(i - 1, i - 1, j) = d_{i-2} \times d_{i-1} \times d_j$;
- Le plus court chemin vers le nœud (i, j) via les sauts $(i, k) \Rightarrow (i, j)$ et $(k+1, j) \Uparrow (i, j)$ définit les mêmes parenthèses : $((M_i, \ldots, M_k)(M_{k+1}, \ldots, M_j))$, et ces sauts pèsent respectivement :
  - $w((i, k), (i, j)) = SP(k + 1, j) + U(i, k, j) = SP(k + 1, j) + d_{i-1} \times d_k \times d_j$;
  - $w((k + 1, j), (i, j)) = SP(i, k) + U(i, k, j) = SP(i, k) + d_{i-1} \times d_k \times d_j$;

Étant donné un produit P de n matrices et le graphe dynamique D_n lui correspondant, Bradford [Bra94] a montré que :

Proposition 1 *Si le plus court chemin entre les sommets* $(0, 0)$ *et* (i, k) *contient le saut* $(i, j) \Rightarrow (i, k)$*, alors il existe un plus court chemin dual qui contient le saut* $(j + 1, k) \Uparrow (i, k)$*.*

La Proposition 1 est fondamentale pour les algorithmes BSP/CGM présentés dans ce chapitre. Elle permet d'éviter la redondance de calcul lors de la recherche de la valeur du plus court chemin pour les sommets du graphe D_n. En effet, pour tout sommet (i, j), parmi tous ses plus courts chemins contenant des sauts, seuls ceux qui ne contiennent que des sauts horizontaux sont évalués. Le graphe d'entrée des algorithmes BSP/CGM est donc un sous-graphe de D_n noté D'_n, dans lequel l'ensemble A' des arcs est constitué de : {l'ensemble des arcs unitaires de D_n} $\cup$ $\{(i, j) \Rightarrow (i, t) : 1 \leq i < j < t \leq n\}$. Ainsi, dans le graphe D'_n, un sommet (i, j) a des sauts uniquement vers les sommets suivant sur la même ligne.

De cette façon, Bradford montre que l'algorithme séquentiel de Godbole, pour un produit de n matrices, est équivalent à un algorithme de plus court chemin one-to-all sur le graphe dynamique D'_n. Dans cet algorithme, les calculs se font, diagonale après diagonale, de la première diagonale à la dernière, calculant la valeur du plus court chemin du sommet virtuel $(0, 0)$ vers chacun des sommets de D'_n. La figure 4.2b présente le graphe D'_4 et la figure 4.3 présente le graphe des tâches (la table de trace) ou matrice de plus

court chemin, noté $SP(9,9)$, associé au graphe dynamique de la figure 4.1. Chaque entrée de ce graphe contient la longueur (le coût) du plus court chemin vers chacun des sommets du graphe D'_n.

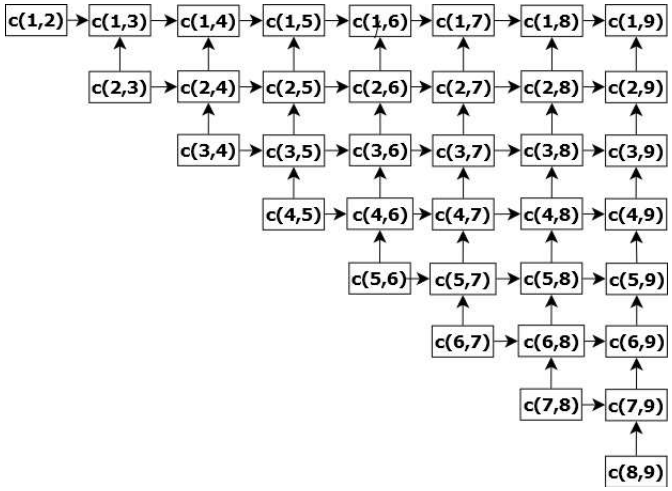


FIGURE 4.3 – Matrice de plus court chemin $SP(9,9)$ (encore appelée graphe des tâches). Elle est utilisée pour calculer $c(1,9) = Cost(1,9)$.

Note 3 *La suite de cette thèse est consacrée à l'étude de la parallélisation des problèmes de la classe abordée suivant le modèle de calcul parallèle BSP/CGM.*

4.4 Travaux de Kechid-Myoupo et leurs insuffisances

Dans cette section, nous présentons l'algorithme parallèle BSP/CGM proposé par Kechid-Myoupo [Kec09] pour le calcul du plus court chemin (one-to-all) dans le graphe pondéré D'_n, sur un Coarse-Grained Multicomputer de p processeurs, chacun muni d'une mémoire locale de taille $O(n^2/p)$. Cet algorithme permet de résoudre un problème MCOP d'ordre n. Pour cela, les traitements à effectuer sur le graphe D'_n sont divisés en un ensemble de tâches. Ensuite, une étude des dépendances entre les différentes tâches est effectuée, et conduit à l'élaboration des différentes étapes de l'algorithme parallèle (calculs et communication). Enfin, les différentes tâches sont distribuées sur l'ensemble des processeurs.

4.4.1 Partitionnement du graphe de tâches

Comme nous l'avons dit plus haut, les valeurs des plus courts chemins vers les nœuds du graphe dynamique D'_n, sont stockées dans une matrice triangulaire supérieure qu'on

appelle *matrice des plus courts chemins*. Nous notons cette matrice $SP(n,n)$. Une case $SP(i,j)$ contient à tout moment la valeur temporaire du plus court chemin vers le sommet (i,j) du DAG D'_n. La matrice $SP(n,n)$ est initialisée à 0, et à la fin de l'algorithme, chaque case doit contenir la valeur finale du plus court chemin vers le sommet de D'_n qui lui correspond.

4.4.1.1 Stratégie de partitionnement

Puisque la résolution du problème revient au calcul des valeurs des plus courts chemins vers chacun des sommets du DAG D'_n, la division de la tâche à effectuer se résume au partitionnement de D'_n (c'est-à-dire la matrice $SP(n,n)$) en sous-graphes (blocs matriciels).

Le graphe D'_n est divisé en $\frac{d\times(d+1)}{2}$ sous-graphes, tel que n/d soit entier. Chacun de ces sous-graphes est référencé par un couple $(i,j)/1 \leq i \leq j \leq d$.

Nous appelons $B_{i,j}$ le bloc de la matrice $SP(n,n)$ qui contient les cases correspondantes au sous-graphe $D'_n(i,j)$. La figure 4.4 montre la division de la matrice $SP(n,n)$ en $\frac{d\times(d+1)}{2}$ blocs.

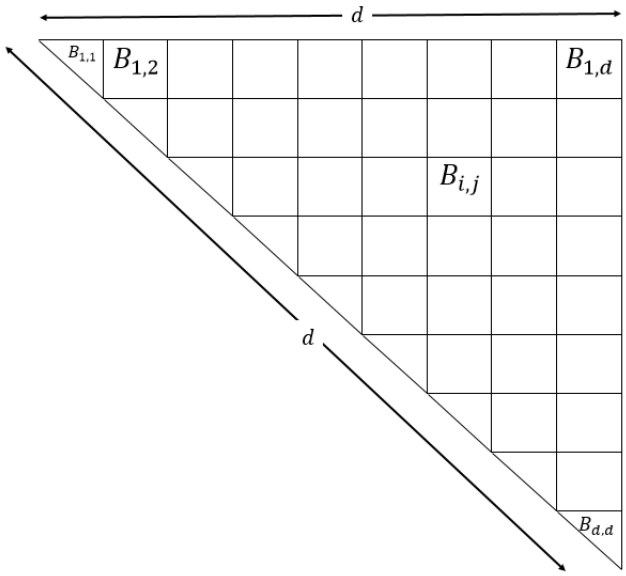


FIGURE 4.4 – Division de la matrice $SP(n,n)$ en $(d \times (d+1)/2)$ blocs.

Terminologie 3 *Considérons $D'_n(i,j)$ et $D'_n(l,m)$ deux sous-graphes de la division présentée par la figure 4.4, et les blocs $B_{i,j}$ et $B_{l,m}$ de $SP(n,n)$ qui leurs correspondent. Nous disons que $D'_n(i,j)$ et $D'_n(l,m)$ (c'est-à-dire les blocs $B_{i,j}$ et $B_{l,m}$) sont sur la même ligne i ssi $i = l$. Nous disons qu'ils sont sur la même colonne j , ssi $j = m$. Nous disons qu'ils sont sur la même diagonale $(j-i+1)$ ssi $(j-i) = (m-l)$.*

À partir de cette terminologie, nous pouvons déduire que le graphe D'_n (c'est-à-dire la matrice $SP(n,n)$) est décomposé en d lignes et d colonnes de sous-graphes (blocs). Le nombre de sous-graphes dans la ligne i est $(d-i+1)$ et le nombre de sous-graphes dans la colonne j est j. Le nombre de sous-graphes dans la diagonale k est $d-k+1$.

Remarque 3 (différentes dimensions des blocs de $SP(n,n)$) .

1. *Comme la matrice des plus courts chemins $SP(n,n)$, les blocs de la première diagonale sont des matrices triangulaires supérieures ;*
2. *Les blocs de la première ligne peuvent compter plus de lignes que les autres (c'est-à-dire plus de sommets) ;*
3. *Les autres blocs sont des matrices carrées pleines, contenant $(n/d)\times(n/d)$ sommets.*

La figure 4.5 donne le nombre de sommets contenus dans les différents sous-graphes de $SP(n,n)$ et l'égalité 4.3 montre les entrées délimitant un bloc $B_{i,j}$ $(1<i<j\leq d)$.

$$B_{i,j}=\begin{pmatrix} SP\left((i-1)(n/d)+2,(j-1)(n/d)+1\right) & \dots & SP\left((i-1)(n/d)+2,j(n/d)\right) \\ . & \dots & . \\ . & \dots & . \\ . & \dots & . \\ SP\left(i(n/d)+1,(j-1)(n/d)+1\right) & \dots & SP\left(i(n/d)+1,j(n/d)\right) \end{pmatrix} \tag{4.3}$$

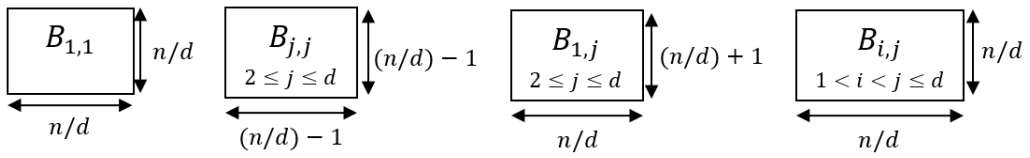


FIGURE 4.5 – Les dimensions des différents blocs de la matrice $SP(n,n)$ divisée en $(d\times(d+1)/2)$ blocs.

La figure 4.6 montre la division de la matrice $SP(n,n)$ correspondant au graphe D'_{18} avec $d=9$. Les diagonales de blocs sont mises en évidence à travers une alternance de couleurs.

4.4.1.2 Dépendance de données

D'après le schéma de dépendances entre les sommets de $SP(n,n)$ (figure 4.2), l'évaluation d'un bloc $B_{i,j}$ nécessite les valeurs déjà calculées des plus courts chemins vers des sommets d'autres blocs.

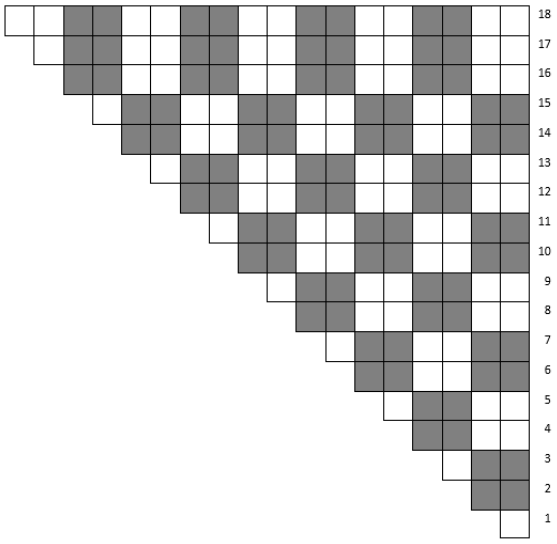


FIGURE 4.6 – La division de la matrice $SP(18, 18)$ avec $d = 9$.

Lemme 3 (Dépendance de sommets) *Le calcul du plus court chemin vers un sommet* $(i, j)/1 \leq i < j \leq n$ *du graphe* D'_n *nécessite la valeur du plus court chemin vers chacun des sommets* $(i, i), \ldots, (i, j-1)$ *et* $(i+1, j), \ldots, (j, j)$.

Preuve. Pour calculer le plus court chemin vers un sommet (i, j), on a besoin, pour chacun de ses arcs entrants, d'avoir la valeur du plus court chemin vers le sommet de départ de l'arc et le poids de cet arc.
Dans le graphe D'_n, les arcs qui arrivent sur un sommet (i, j) proviennent de l'ensemble de sommets : $S^l_{i,j} = (i, i), \ldots, (i, j-1) \cup (i+1, j)$ [7] . Le poids d'un saut $(i, t) \Rightarrow (i, j)/t < j$ est $W((i, t) \Rightarrow (i, j)) = SP(t+1, j) + U(i, t, j)$. Le calcul des poids des arcs provenant des sommets de $S^l_{i,j}$ nécessite les valeurs des plus courts chemins vers les sommets $S^c_{i,j} = (i+1, j), \ldots, (j, j)$ de la colonne j. Les valeurs produites par la fonction U dépendent des dimensions des matrices de la chaîne à parenthéser. Chacun des processeurs détiendra localement ces dimensions pour toutes les entités de la chaîne. Ainsi, le calcul de $SP(i, j)$ nécessite les valeurs des plus courts chemins vers les sommets des groupes $S^l_{i,j}$ et $S^c_{i,j}$. ■

Lemme 4 (Poids des sauts d'un sous-graphe) *Le calcul des poids des sauts partants des sommets d'un sous-graphe* $D'_n(i, k)$ *vers les sommets d'un sous-graphe* $D'_n(i, j)/1 \leq i \leq k \leq j$ *nécessite uniquement les valeurs des plus courts chemins vers le sous-graphe* $D'_n(k, j)$.

7. Les sauts verticaux ne sont pas considérés en vertu de la proposition 1.

Preuve. Le calcul des poids des sauts qui partent d'un ensemble de sommets successifs dans la ligne i : $(i,k),(i,k+1),\ldots,(i,h)$, vers un sommet $(i,m)/i \leq k < h < m$ ne nécessite que les valeurs du plus court chemin vers les sommets $(k+1,m),\ldots,(h+1,m)$ qui est un ensemble de sommets successifs de la colonne m. Ainsi, le calcul des poids des sauts de toute ligne i' du sous-graphe $D'_n(i,k)$, i.e. $(i',(k-1)\times(n/d)+1),\ldots,(i',k\times(n/d))$ vers chacun des sommets (i',j') de la ligne i' du sous-graphe $D'_n(i,j)$, i.e. $(i',(j-1)\times(n/d)+1),\ldots,(i',j'),\ldots,(i',j\times(n/d))$, ne nécessite que les valeurs du plus court chemin vers l'ensemble de sommets : $((k-1)\times(n/d)+2,j'),\ldots,(k\times(n/d)+1,j')$.
Or cet ensemble est exactement l'ensemble des sommets de la colonne j' du sous-graphe $D'_n(k,j)$ avec j' appartenant à $(j-1)\times(n/d)+1,(j-1)\times(n/d)+2,\ldots,j\times(n/d)-1$, $j\times(n/d)$. Ainsi le calcul des poids des sauts de chacune des lignes de $D'_n(i,k)$ ne nécessite que les valeurs du plus court chemin des sommets du sous-graphe $D'_n(k,j)$. ■

Il découle des lemmes 3 et 4 que :

Théorème 1 (Dépendance entre les sous-graphes) *Le calcul des plus courts chemins vers les sommets de $D'_n(i,j)$ nécessite la valeur du plus court chemin vers chacun des sommets des sous-graphes : $D'_n(i,i),\ldots,D'_n(i,j-1)$ et $D'_n(i+1,j),\ldots,D'_n(j,j)$.*

La figure 4.7 montre les blocs de D'_n desquelles dépendent les bloc $B_{i,k}$, $B_{h,j}$ et $B_{f,m}$ (correspondant respectivement aux sous-graphes $D'_n(i,k)$, $D'_n(h,j)$ et $D'_n(f,m)$).

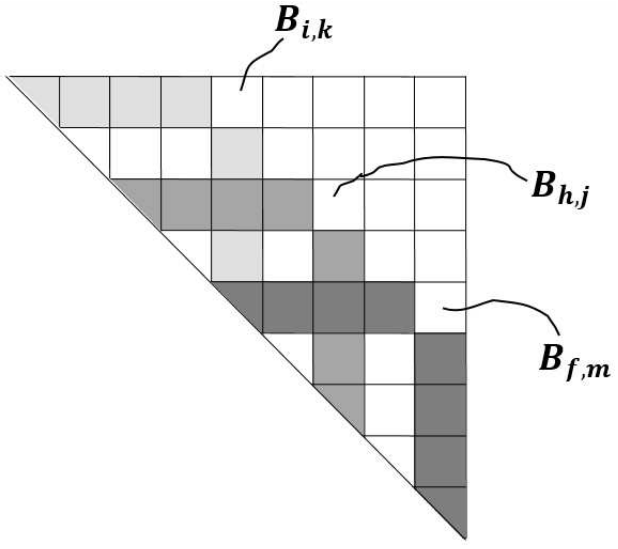


FIGURE 4.7 – Blocs nécessaires au calcul des valeurs des entrées des blocs $B_{i,k}$, $B_{h,j}$ et $B_{f,m}$.

Corollaire 2 *Les relations de dépendance entre les sous-graphes montrent que :*

- *Le calcul des plus courts chemins vers les sommets appartenant à un sous-graphe d'une diagonale donnée demande les valeurs des plus courts chemins vers les sommets appartenant aux sous-graphes de toutes les diagonales précédentes ;*
- *Les sous-graphes de la même diagonale sont indépendants les uns des autres. Ainsi, les calculs des plus courts chemins vers les sommets dont ils sont constitués peuvent se faire en parallèle.*

4.4.2 Algorithme CGM et ses défauts

A cause des dépendances des données entre les blocs du graphe D'_n, l'évaluation de ceux-ci doit se faire suivant un ordre bien adapté. En effet, les valeurs des plus courts chemins vers les sommets des blocs d'une diagonale k ne peuvent pas s'obtenir avant celles des sommets contenus dans chacun des blocs dont ils dépendent sur les diagonales précédentes (les diagonales $1, 2, \ldots, k-1$ d'après le théorème 1). Cependant, la nature de ces calculs (minimum de valeurs) permet de les entamer avant la fin de l'évaluation des blocs de la diagonale $k-1$. Ainsi, il y a deux approches possibles pour le séquencement des étapes de l'algorithme :

1. commencer l'évaluation des blocs d'une diagonale aussitôt que possible (c'est-à-dire dès que les premières données nécessaires sont disponibles) ; ou bien,
2. commencer l'évaluation des blocs d'une diagonale k après celle des blocs de la dernière diagonale de blocs desquels ils dépendent, c'est-à-dire celles de la diagonale $k-1$.

Il y a donc deux versions de cet algorithme BSP/CGM générique. Une première version dans laquelle, l'évaluation des blocs est faite de manière progressive et commence le plus tôt possible. Et une seconde version dans laquelle, cette évaluation ne commence que lorsque toutes les valeurs desquelles ces blocs dépendent sont présentes. Dans cette thèse, nous ne présenterons que la version 1, et la contribution que nous apporterons (voir section 4.4.3) sera valable pour les deux versions.

Le théorème 2 indique à quel moment les calculs des valeurs des plus courts chemins vers les sommets d'un sous-graphe $D'_n(i,j)$ peuvent commencer.

Théorème 2 *Pour tout sous-graphe $D'_n(i,j)$, après le calcul de chaque diagonale de sous-graphes (blocs) h tel que $\lceil (j-i/2) \rceil + 1 \leq h \leq j-i+1$, au moins deux valeurs possibles du plus court chemin vers chacun des sommets du sous-graphe $D'_n(i,j)$ sont calculables.*

Preuve. Pour évaluer les plus courts chemins vers les sommets d'un sous-graphe $D'_n(i,j)$ de la diagonale $k = j-i+1$, on a besoin pour chaque sommet d'avoir le coût des différents arcs entrants et la valeur du plus court chemin vers le sommet source de chacun de ces arcs. $D'_n(i,j)$ reçoit des arcs à partir des sous-graphes $D'_n(i,j')/j' \in [i, j-1]$, et les coûts des arcs en provenance d'un sous-graphe $D'_n(i,j')$ peuvent être évalués dès que les valeurs du plus court chemin des sommets de $D'_n(j',j)$ sont calculés (d'après le lemme 4). Ce qui n'est possible qu'à partir de la diagonale $t/(t \geq j'-i+1)$ et $(t \geq j-j'+1)$. Ceci implique que : $(t \geq \lceil j-i/2 \rceil + 1)$. Ainsi, après l'évaluation de la diagonale $h/(t \leq h \leq j-i+1)$, les blocs $(i, h+i-1), (h+i-1, j), (i, j-h+1)$ et $(j-h+1, j)$ sont évalués. Et donc, deux valeurs possibles du plus court chemin vers chacun des sommets de $D'_n(i,j)$ sont calculables. Ceux dont les arcs entrants proviennent de $D'_n(i, h+i-1)$ et $D'_n(i, j-h+1)$.
■

4.4.2.1 Structure générale de l'algorithme BSP/CGM

Nous sommes maintenant prêts à dévoiler les différentes étapes de l'algorithme BSP/CGM. Il consiste en une succession de d étapes similaires. Dans chacune de ces étapes, les traitements pour tous les blocs (sous-graphe) d'une diagonale qu'on appelle *« la diagonale en cours »* sont effectués en parallèles. Ces traitements commencent par les blocs de la première diagonale, à partir de la gauche, puis la deuxième et ainsi de suite jusqu'à la dernière (la diagonale d). L'algorithme 7 donne la structure globale de cette approche. Il permet d'obtenir le coût optimal, pour un problème de parenthésage optimal d'une chaine de n matrices. C'est-à-dire le coût $Cost(1,n)$ du plus court chemin vers le sommet $(1,n)$.

Cette approche consiste à calculer *progressivement* les valeurs des plus courts chemins vers les sommets de chacun des sous-graphes (blocs). Le calcul de la valeur du plus court chemin vers chacun des sommets d'un sous-graphe de la diagonale k commence dès que la diagonale $\lceil k/2 \rceil$ est évaluée (théorème 2), les diagonales étant numérotées de gauche à droite de 1 à d. Après la finalisation de l'évaluation des blocs d'une diagonale k (phase 1 de l'itération k de l'algorithme 7), chacun de ses blocs est communiqué (phase 3 de l'itération k de l'algorithme 7) aux processeurs détenant les blocs qui en ont besoin pour la mise à jour (phase 2) ou la finalisation (phase 1) de l'évaluation de leurs blocs dans la prochaine itération (étape), c'est-à-dire l'itération $k+1$. En effet, à l'itération $k+1$, deux tâches doivent être effectués :

1. Pour chaque entrée d'un bloc d'une diagonale m $(k+2 \leq m \leq 2 \times k)$, de nouvelles valeurs peuvent être calculées et des mises à jour de valeurs temporaires sont effec-

tuées (*principe de relaxation de chemin* [LRSC01]). Ce traitement correspond à la phase 2 de l'itération $k+1$ de l'algorithme 7 ;

2. Pour chacune des entrées d'un bloc de la diagonale $k+1$, deux nouvelles valeurs possibles sont calculées et la mise à jour finale des valeurs temporaires de ces entrées est effectuée. Ce traitement correspond à la phase 1 de l'itération $k+1$ de l'algorithme 7.

Algorithme 7 : Structure générale de l'algorithme BSP/CGM pour un problème de la classe abordée.

Entrées : La matrice $SP(n,n)$ initialisée à 0, stockée sur un CGM(n,p).
Sorties : La valeur du plus court chemin vers chacun des sommets de D'_n.

1 **pour** $k = 1$ **à** d **faire**
2 | **Phase 1 : Finalisation** du calcul des valeurs des plus courts chemins vers les sommets des sous-graphes de la diagonale k ;
3 | **Phase 2 : Mise à jour** des valeurs des plus courts chemins vers les sommets de chacun des sous-graphes des diagonales ($k+1$, $k+2$, ..., $\min\{2\times(k-1), d\}$);
4 | **Phase 3 : Communication** de tout bloc $B_{i,j}$, correspondant à un sous-graphe de la diagonale en cours, aux processeurs qui détiennent ses sous-graphes supérieurs [8]et ses sous-graphes droits ;
5 **fin**

Ce processus est répété à chaque itération jusqu'à l'itération d où seule la phase 1 de l'algorithme est exécutée. À l'itération 1 de l'algorithme, seules les phases 1 et 3 sont exécutées.
On peut observer aisément que le calcul de la longueur optimale des chemins (en faisant des sauts qui partent de $D'_n(i,k)$) allant vers les sommets du sous-graphe $D'_n(i,j)$, est équivalent à la multiplication de matrices $(+, min)$ [9] des deux blocs $B_{i,k}$ et $B_{k,j}$. Ainsi, la structure des traitements effectués dans les phases 2 et 1 de cet algorithme est donnée, respectivement, dans les algorithmes 8 et 9.

Une fois l'itération $(j-i)$ de l'algorithme 7 terminée, les seuls chemins non encore évalués pour les sommets du sous-graphe $D'_n(i,j)$ sont ceux dont le dernier arc est :

a) soit un arc unitaire (vertical ou horizontal) ;

b) soit un saut horizontal qui provient d'un sommet interne à $D'_n(i,j)$;

8. Un sous-graphe $D'_n(i,j)$ est supérieur à un sous-graphe $D'_n(k,l)$ ssi ($i<k$ et $j=l$). Un sous-graphe $D'_n(i,j)$ est droit à un sous-graphe $D'_n(k,l)$ ssi ($i=k$ et $j>l$).

9. La multiplication de matrices (+,min) est une multiplication de matrices dans laquelle l'opération de multiplication est remplacée par l'addition et la somme par la fonction min.

Algorithme 8 : Mise à jour du bloc (i, j) à l'itération $h+1$ ($\lceil (j-i/2) \rceil + 1 \leq h < j-i+1$).

début
  $M_1 \leftarrow Multiplication\_matrices(+, min)(B_{i,h+i-1}, B_{h+i-1,j})$;
  $M_2 \leftarrow Multiplication\_matrices(+, min)(B_{i,j-h+1}, B_{j-h+1,j})$;
  $B_{i,j} \leftarrow \min\{M_1, M_2, B_{i,j}\}$;
fin

Algorithme 9 : Finalisation de calcul du bloc (i, j) à l'itération $j-i+1$.

pour $k = (j-i-1) \times (n/d) + 1$ **à** $(j-i+1) \times (n/d) - 1$ **faire**
  pour chaque *sommet s de la diagonale k appartenant à* $D'_n(i, j)$ **faire**
    $SP(s) \leftarrow \min$ {$SP(s)$, poids des chemins dont l'arc final est un saut en provenance de $D'_n(i, i)$, poids des chemins dont l'arc final est un saut interne, poids des chemins dont l'arc final est un arc unitaire interne};
  fin
fin

c) soit un saut horizontal qui provient d'un sommet de $D'_n(i, i)$.

Dans tous les cas, le calcul du poids du chemin induit par chacun de ces arcs, vers un sommet (i', j') de $D'_n(i, j)$, nécessite la valeur du plus court chemin d'un des sommets (e', f') de $D'_n(i, j)/f' - e' < j' - i'$ (lemme 4). Dans les cas (a) et (b), cette valeur est nécessaire pour le calcul du plus court chemin du sommet de départ du dernier arc, et dans le cas (c) elle est nécessaire pour le calcul du poids du dernier arc. Ainsi, l'algorithme 9 est un algorithme classique de plus court chemin sur le sous-graphe $D'_n(i, j)$ dans lequel chaque sommet peut recevoir un arc unitaire ou un saut en provenance d'un sommet interne à $D'_n(i, j)$ ou appartenant à $D'_n(i, i)$. Pour chaque sommet de $D'_n(i, j)$, la valeur finale prise pour son plus court chemin est produite après l'exécution de cet algorithme.

Lemme 5 *La complexité temporelle des algorithmes 8 et 9 pour l'évaluation d'un bloc est* $O(n^3/d^3)$.

Preuve. La finalisation des traitements (algorithme 9) sur un bloc est similaire à l'algorithme de Godbole pour un problème d'ordre (n/d). Ainsi elle nécessite un temps en $O(n^3/d^3)$. L'algorithme 8 permet d'effectuer une multiplication (+,min) pour deux matrices d'ordre $(n/d) \times (n/d)$, c'est-à-dire en $O(n^3/d^3)$ temps d'exécution. D'où les algorithmes 8 et 9 nécessitent un temps en $O(n^3/d^3)$. ■

De tout ceci, nous pouvons dire que :

Corollaire 3 *Puisque le nombre de messages communiqués dans chaque itération est indépendant de n, et la taille des messages communiqués dans chaque itération croît en fonction de n, Cet algorithme BSP/CGM générique pour la résolution des problèmes de la classe considérée est extensible*[10].

Chaque bloc $B_{i,j}$ de $SP(n,n)$, pour $i < j$, représente un sous-problème. Après avoir décomposé le problème à résoudre en sous-problèmes de même type (sous-graphes de même forme), ceux-ci doivent être distribués sur p processeurs. Ces processeurs calculeront les valeurs des plus courts chemins vers chacun des sommets des sous-graphes qu'ils détiendront.

4.4.2.2 Distribution des blocs aux processeurs et complexité

La première diagonale de blocs du graphe D'_n (après partitionnement) contient le plus grand nombre de blocs. Ce nombre est supposé égal à p. Il diminue d'une unité d'une diagonale à l'autre (il n'y a qu'un seul bloc à la diagonale d). Ainsi, le choix des processeurs à utiliser lorsque le nombre de blocs de la diagonale en cours est inférieur à p est une tâche très importante, car elle permet de :

1. minimiser le coût effectif des communications ;
2. équilibrer les charges entre les différents processeurs ;
3. faciliter l'implémentation de l'algorithme ;
4. faciliter la prédiction du coût d'exécution de l'algorithme.

Plusieurs distributions sont possibles :

Une distribution horizontale (respectivement verticale) qui consiste à affecter tous les blocs d'une ligne (respectivement d'une colonne) i au processeur P_i. Le schéma de communication correspondant est facile à réaliser et préserve les communications horizontales (respectivement verticales) mais induit un grand déséquilibre de charge entre les processeurs. En effet, le processeur P_i (P_j) évalue tous ses blocs en $O(n/p)^3 \times (1 + 2 + \ldots + (p - i) + (p - i + 1))$ temps de calcul ;

Une distribution aléatoire équitable qui consiste à attribuer les blocs de chaque diagonale aux processeurs de manière aléatoire, en se rassurant que le nombre de blocs qu'ils détiennent soit équilibré. Cette méthode ne prend pas en compte la dépendance de donnée entre les blocs, et complique grandement la définition d'un

10. C'est-à-dire qu'il ne perd pas ses performances avec l'accroissement de la taille du problème (voir section 2.4.3).

schéma de communication clair et efficace entre les processeurs. Elle complique donc : la prédiction de coût de l'algorithme, son implémentation, etc.

Le mapping (la distribution de données) choisi est appelé « **Distribution par Projection Bidirectionnelle Alternative** » (DPBA). Il facilite la prédiction du coût et l'implantation de l'algorithme tout en essayant d'équilibrer les charges entre les processeurs. Son principe est illustré dans la figure 4.8. Pour distribuer les $p(p+1)/2$ blocs du graphe dynamique D'_n sur les p processeurs, on affecte chacun des p blocs de la première diagonale à l'un des processeurs. Le bloc (i,i) est affecté au processeur P_i. Les blocs des autres diagonales sont projetés sur la première diagonale, de manière alternative, une fois horizontalement et une fois verticalement. Un bloc d'une diagonale k qui est projeté sur le bloc (i,i) est affecté au processeur P_i. Ainsi, deux blocs qui sont projetés sur le même bloc de la première diagonale seront évalués (traités) par le même processeur. Les blocs $B_{1,1}$, $B_{1,3}$ et $B_{1,5}$ seront évalués par le processeur P_1.

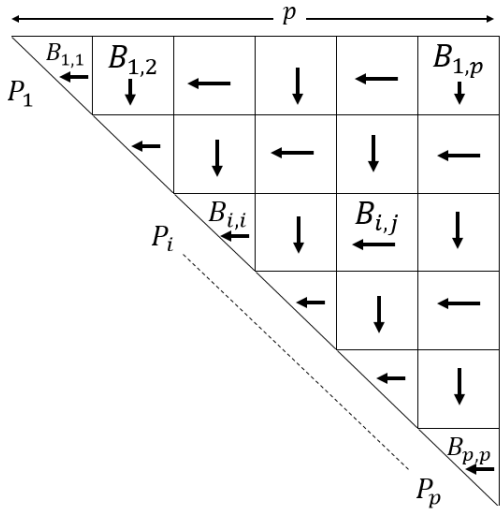


FIGURE 4.8 – Distribution par Projection Bidirectionnelle Alternative

Théorème 3 *En utilisant la Distribution par Projection Bidirectionnelle Alternative, l'algorithme 7 utilise p rondes de communication avec $O(n^3/p)$ calculs séquentiels sur chaque processeur.*

Preuve. L'algorithme 7 effectue p itérations. Chacune des itérations ne demande pas plus d'une communication globale entre les processeurs actifs (phase 3). Ainsi, le nombre de rondes de communication est égal à p.
Comme vu à la section 4.4.1.2, un bloc appartenant à une diagonale k nécessite plus de

calculs qu'un bloc appartenant à une diagonale h si $(h < k)$. Le processeur du bloc de la diagonale k nécessite $k-1$ exécutions de l'algorithme 8 et une exécution de l'algorithme 9, et celui du bloc de la diagonale h exécute $h-1$ fois l'algorithme 8 et une fois l'algorithme 9.

Ainsi, puisque le nombre de blocs à évaluer par chacun des processeurs est le même $((\lceil p+1 \rceil /2)$ ou $(\lfloor p+1 \rfloor /2))$, le processeur qui fait le moins de calculs est celui qui passe à l'état inactif[11] en premier (le processeur $P_{p/2}$) et le dernier à passer à l'état inactif (le processeur P_p) est celui qui fait le plus de calculs. Le processeur $P_{p/2}$ calculera un bloc de chacune des $(p/2)$ premières diagonales. Ainsi, il accomplira $p(p+2)/8$ exécutions en $O(n^3/p^3)$, et ces calculs auront une complexité temporelle en $O(n^3/p)$. Le processeur P_p traitera un bloc pour chaque couple de diagonales contiguës, ainsi il accomplira $p(p+2)/4$ exécutions en $O(n^3/p^3)$. Ces calculs auront donc une complexité temporelle en $O(n^3/p)$. D'où, chacun des processeur réalise un calcul séquentiel en $O(n^3/p)$. ■

4.4.2.3 Avantages et inconvénients de la DPBA

Avantages :

1. La clarté et la simplicité d'implémentation du schéma de communication ;
2. La diminution de moitié du nombre de messages communiqués dans l'algorithme :
   - en diminuant d'un demi le nombre de processeurs auxquels un bloc $B_{i,j}$ est envoyé une fois calculé. En effet, la moitié des blocs qui sont supérieurs (respectivement à droite) à un bloc $B_{i,j}$ sont détenus par le même processeur (le processeur P_i ou P_j) ;
   - en diminuant d'un quart le nombre de messages communiqués pour l'évaluation de chaque bloc. En effet, tout bloc $B_{i,j}$ est évalué par un processeur qui détient en local $(j-i+1)/2$ des blocs nécessaires à son évaluation.

Inconvénients :

1. Le processeur $P_{\lceil p/2 \rceil}$ deviens inactif à l'étape $\lceil p/2 \rceil + 1$, et à chacune des étapes suivantes, deux nouveaux processeurs deviennent inactifs. Ainsi, la DPBA favorise l'oisiveté des processeurs utilisés.
2. Comme la charge de calculs induite par les blocs du DAG D'_n est de plus en plus grande lorsqu'on passe d'une étape à l'autre (c'est-à-dire d'une diagonale à l'autre), celles des blocs des dernières diagonales sont plus grandes que celles

11. Processeur ayant terminé ses calculs dans l'algorithme.

des premières (celle du bloc de la diagonale d est la plus grande). De ce fait, la DPBA cause un grand déséquilibre de charge de calcul entre les processeurs. Nous allons montrer dans la section suivante que ce déséquilibre augmente drastiquement avec le nombre de processeurs utilisés.

Corollaire 4 *La Distribution par Projection Bidirectionnelle Alternative favorise l'oisiveté des processeurs à partir de l'étape* $\lceil p/2 \rceil + 1$ *de l'algorithme. Certains processeurs à partir de cette étape peuvent avoir une grande charge de calcul pendant que d'autres sont inactifs.*

4.4.3 Notre contribution : équilibrage de charges

Dans cette section, nous proposons une technique de distribution des blocs aux processeurs permettant de palier aux défauts majeurs de la DPBA. Cette technique peut être appliquée à tout graphe partitionné de façon similaire au nôtre. L'objectif est d'équilibrer les charges entre les processeurs, réduire au minimum l'oisiveté des processeurs tout en préservant les communications. Pour mieux comparer notre approche avec la DPBA, nous gardons les mêmes terminologies et le même partitionnement du graphe D'_n, c'est-à-dire en $p(p+1)/2$ blocs de même taille.

4.4.3.1 Hypothèse et principe

Nous proposons une méthode de distribution progressive des données dans laquelle, les blocs sont attribués aux processeurs diagonale après diagonale. Cette méthode repose sur deux informations fondamentales :

- La prédiction, à chaque instant, de la charge de calculs courante des différents processeurs (nombre d'opérations à effectuer), pour mieux équilibrer leur charge ; et,
- La prise en compte de la localisation des blocs déjà alloués au moment de l'allocation d'un nouveau bloc, pour minimiser les communications.

Elle peut être mise en œuvre suivant deux approches :

1. Une approche ascendante dans laquelle l'on alloue les blocs aux processeurs en commençant par ceux de la première diagonale et en allant vers la dernière ; et,
2. Une approche descendante dans laquelle les blocs sont alloués à partir de la dernière diagonale vers la première.

Nous supposons qu'en fonction du problème à résoudre, nous pouvons, après le partitionnement du DAG D'_n correspondant, estimer le nombre de calculs élémentaires néces-

saire pour l'évaluation de chaque bloc. Ainsi, utiliser cette prédiction (estimation) pour évaluer la charge de calcul d'un processeur qui détient un ensemble de blocs spécifique [12].

Notre principe est le suivant : nous allouons les blocs aux processeurs, diagonale après diagonale, en s'assurant qu'un processeur n'aura pas plus d'un bloc dans la même diagonale [13]. Au départ, nous allouons un bloc à chaque processeur. Ensuite, suivant la charge induite par le bloc détenu par chaque processeur, nous allouons les blocs de la diagonale suivante (ou bien ceux de la diagonale en cours non encore alloués) aux processeurs ayant les plus faibles charges courantes. Nous procédons de la même façon jusqu'à la dernière diagonale. Ainsi, lors de l'allocation de chaque bloc, nous essayons de réduire le déséquilibre de charge induit par les allocations précédentes.

D'autre part, pour minimiser les communications, les blocs d'un même processeur sont concentrés autant que faire se peut, soit sur la même ligne, soit sur la même colonne.

Dans la suite de cette section, nous montrons comment estimer la charge courante d'un processeur, pour la résolution d'un problème d'ordre n de la classe abordée. Ensuite, nous présentons chacun des deux scénarios de notre méthode d'allocation de tâches ainsi que leurs avantages et inconvénients. Nous terminons par une brève comparaison de notre approche descendante avec la DPBA.

4.4.3.2 Estimation de charges

À partir de la forme générale des équations fonctionnelles de la classe de problèmes abordée dans cette thèse (équation 3.2 de la section 3.3), le calcul du plus court chemin vers chaque sommet de la diagonale d du DAG D'_n requiert $3(d-1)$ opérations arithmétiques. Il s'agit comme le montre l'équation 3.2, de $(d-1)$ traitements constitués de deux opérations d'addition et d'une opération de minimum.

Posons $n/p = m$. Comme en générale chaque bloc de $SP(n,n)$ a une taille de $m \times m$ (voir section 4.4.1), chaque bloc de la première diagonale de blocs induira une charge de :

$$\alpha = 3 \times \sum_{k=1}^{m} k(m-k) = 3 \times \frac{((m-1) \times m \times (m+1))}{6}$$

De même, chaque bloc de la diagonale $i/i > 1$ induira à son tour une charge de :

$$3 \times \left(\sum_{k=1}^{m} k((i-2)m+k) + \sum_{k=1}^{m-1} k(im-k) \right) = 3 \times (i-1)m^3$$

12. Les blocs d'un même processeur peuvent induire des charges de calculs très différentes.
13. Le nombre maximum de blocs d'une diagonale est supposé égal à p, voir section 4.4.2.2

Si nous posons $\beta = 3m^3$, alors, un processeur qui évalue un bloc de la deuxième diagonale effectuera β calculs et un autre qui évalue les blocs $B_{1,1}$ et $B_{1,4}$ aura une charge globale de $\alpha + 3\beta$. En définitive, la charge de calculs induite par un bloc de la première diagonale est α tandis que celle d'un bloc de la diagonale $i/i > 1$ est $(i-1)\beta$.

4.4.3.3 Distribution ascendante des données

Adoptons les terminologies suivantes :

Terminologie 4 .

- *Un processeur est dit* éligible *pour une diagonale i s'il fait partie des $p - i + 1$ processeurs ayant les plus faibles charges ;*
- *Un processeur éligible est dit* disponible *si on ne lui a pas encore attribué un bloc ;*
- *Un bloc est* libre *s'il n'est pas encore alloué à un processeur ;*
- *Le* suivant *d'un processeur P_s (noté suivant(s)) est le processeur P_{s+1} si $s < p$ et le processeur P_1 si $s = p$.*

Cette distribution part de la première diagonale et évolue diagonale après diagonale jusqu'à la dernière. Après avoir attribué chaque bloc de la première diagonale à un des processeurs (de 1 à p), à chacune des diagonales $i/i > 1$ suivantes, on commence par choisir les processeurs éligibles à qui allouer les blocs de la diagonale en cours. Ensuite, on alloue chaque bloc $B_{l,c}$ de cette diagonale, soit au processeur l, soit au processeur c. Dans le cas où ces deux processeurs ne sont pas éligibles, ou bien ne sont pas disponibles, on attribue le bloc $B_{l,c}$ à n'importe quel processeur disponible. L'idéal est d'attribuer ce bloc à un processeur ayant déjà un bloc sur la même ligne (ou colonne), ceci pour gagner une communication. On passe à la diagonale suivante lorsque tous les blocs de la diagonale en cours sont alloués.

La structure globale de cette distribution est formalisée par l'algorithme 10. Dans cet algorithme, la variable *numProc* désigne le numéro du dernier processeur à qui un bloc a été alloué. Elle est initialisée à p.

Les blocs d'une même diagonale induisent tous la même charge. Cette charge augmente de β lorsqu'on passe d'une diagonale i à une diagonale $i+1$. Ainsi, à chaque fois que l'on attribue un bloc à un processeur, celui-ci devient le processeur ayant la plus grande charge, et le choix le plus naturel pour continuer est son suivant. D'où, l'attribution ascendante des blocs aux processeurs se fait de façon cyclique sur leur numéro. L'algorithme 11 permet d'effectuer la distribution des blocs d'une diagonale d.

Algorithme 10 : Distribution ascendante des tâches

Entrées : Nombre de blocs et leurs dimensions.
Sorties : Schéma d'allocation des blocs aux processeurs.

```
1 numProc ← p ;
2 pour d = 1 à p faire
3 |  Distribuer les blocs de la diagonale d ;
4 fin
```

Lemme 6 *L'allocation des blocs du graphe des tâches $SP(n, n)$ aux processeurs par la distribution ascendante de données se fait dans le pire des cas en $O(p^2)$ temps d'exécution.*

Preuve. Bien qu'il y ait imbrication des boucles Pour et Tant que (lignes 11 et 12) dans l'algorithme 11, l'ensemble des blocs de la diagonale d et l'ensemble des processeurs sont chacun parcouru une et une seule fois par ces deux boucles. Donc ces deux boucles sont exécutées en $O(p)$ temps d'exécution. De ce fait, puisque la première phase de l'algorithme 11 se réalise en $O(p)$ temps d'exécution, il nécessite dans le pire des cas $O(p)$ temps d'exécution. Ainsi, l'algorithme 10 réalise le schéma d'allocation des blocs aux processeurs dans le pire des cas en $O(p^2)$ temps d'exécution. ■

Théorème 4 *Avec la distribution ascendante de données, le déséquilibre de charges entre les processeurs est bornée par $(p-1)\beta$.*

Preuve. Le processeur de plus faible charge P_k choisi pour le dernier bloc du graphe (celui de la dernière diagonale) se verra augmenter une charge de $(p-1)\beta$. Et donc aura au maximum cette quantité de charge en plus que le processeur P_{k+1} (le nouveau processeur ayant la plus faible charge). La différence de charge entre les processeurs dépend donc du nombre p de processeurs. Et est bornée par $(p-1)\beta$. En particulier, elle vaut 7β pour $p = 8$. ■

Cette distribution est illustrée dans la figure 4.9 avec $p = 7$. Elle représente la subdivision du graphe dynamique D'_n. Chacune de ses cases représente un bloc et contient le numéro du processeur à qui le bloc est attribué.

Le tableau 4.1 présente à chaque étape (diagonale) l'estimation de la charge globale courante de chaque processeur. Il montre en plus le déséquilibre de charge final entre ceux-ci. Et pour chaque processeur, le nombre de ses blocs qui se trouvent sur une même ligne ou sur une même colonne qu'un autre bloc qui lui est alloué (permet de gagner une communication) ainsi que le nombre total de ses blocs.

Algorithme 11 : Distribution des blocs de la diagonale d.

Entrées : le numéro $numProc$ du dernier processeur à qui un bloc a été alloué et le numéro d de la diagonale en cours.

Sorties : les numéros des processeurs qui évalueront chaque bloc de la diagonale d.

```
num ← suivant(numProc) ;
pour i = 1 à p − d + 1 faire
    si ((num − d + 1) ≥ 1) et (B_{num−d+1,num} libre) alors
        attribuer le bloc B_{num−d+1,num} à P_num ;
    sinon si (num ≤ p − d + 1) et (B_{num,num+d−1} libre) alors
        attribuer le bloc B_{num,num+d−1} à P_num ;
    fin
    num ← suivant(num) ;
fin
num ← suivant(numProc) ;
pour i = 1 à p − d + 1 faire
    tant que B_{i,i+d−1} libre faire
        si (P_num disponible) alors
            attribuer le bloc B_{i,i+d−1} à P_num ;
        fin
        num ← suivant(num) ;
    fin
fin
```

p	Etapes (ou diagonale)							Bilan	
	1	2	3	4	5	6	7	Total	Msg
1	α	α + β	α + 3β	α + 6β				**α + 6β**	4/4
2	α	α + β	α + 3β		α + 7β			**α + 7β**	4/4
3	α	α + β	α + 3β		α + 7β			**α + 7β**	4/4
4	α	α + β	α + 3β		α + 7β			**α + 7β**	4/4
5	α	α + β		α + 4β		α + 9β		**α + 9β**	4/4
6	α	α + β		α + 4β		α + 9β		**α + 9β**	4/4
7	α		α + 2β	α + 5β			α + 11β	**α + 11β**	4/4

TABLE 4.1 – Évolution des charges des processeurs et nombre de blocs qui permettent de gagner une communication (message) : Cas de 7 processeurs.

Le déséquilibre de charges dans ce cas est de 5β. Le processeur P_7 est celui qui effectue le plus de calculs ($\alpha + 11\beta$) et P_1 est celui qui en effectue le moins ($\alpha + 6\beta$). D'autre part, chaque processeur évalue des blocs situés sur une ligne ou une colonne sur laquelle il détient déjà un autre bloc.

Les blocs sont répartis sur les processeurs avec la même régularité sur le graphe. Ainsi,

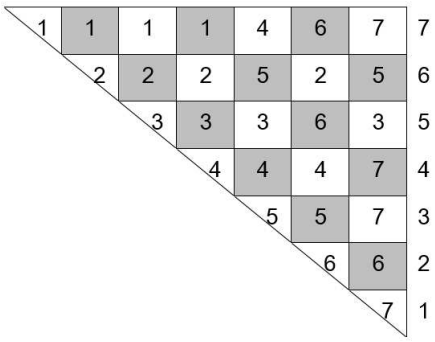


FIGURE 4.9 – Distribution ascendante des tâches avec p=7.

l'oisiveté des processeurs est limitée dans tout le processus.

Résultat 1 *La distribution ascendante des données se fait dans le pire des cas en* $O(p^2)$ *temps d'exécution et le déséquilibre de charges entre les processeurs est borné par* $(p-1)\beta$ *opérations arithmétiques. En plus, cette distribution minimise les communications de manière satisfaisante.*

4.4.3.4 Distribution descendante des données

La différence fondamentale entre cette distribution et la distribution ascendante est qu'elle débute à la dernière diagonale (la diagonale p) et progresse en vague vers la première.

Puisqu'on commence la distribution par les blocs induisant les plus fortes charges ($(p-1)\beta$ pour l'unique bloc de la diagonale p), certains processeurs pourront être plusieurs fois éligibles sans que d'autres ne le soient, ceci à cause de leur charge courante. Ainsi, le choix des $(p-i+1)$ processeurs éligibles d'une diagonale i donnée n'est plus évident. Il devient donc important de définir une méthode d'identification des processeurs éligibles à chaque diagonale avant de commencer l'attribution des blocs.

L'attribution d'un premier bloc à chaque processeur permet de parcourir les δ dernières diagonales de blocs du graphe. En effet, l'attribution commence par le processeur $P_{\lceil p/2 \rceil}$ qui reçoit une charge de $(p-1)\beta$ et le dernier qui reçoit son premier bloc est $P_{\lceil p/2 \rceil+1}$. Ce bloc est situé sur la diagonale $(p-\delta+1)$. Il induit une charge de calcul $(p-\delta)\beta$ (c'est la plus faible charge courante). δ est la plus petite valeur de m tel que $p \leq m(m+1)/2$. Cette valeur est égale à : $\delta = \left\lceil \left(-1+\sqrt{1+8p}\right)/2 \right\rceil$.

La distribution descendante des données est formalisée par l'algorithme 12. Dans cet algorithme, la variable *« numProc »* représente le numéro du dernier processeur éligible de la diagonale précédente, et la variable *« chargeMin »* représente la plus grande charge

possible d'un processeur éligible à un moment donné, elle est initialisée à $(p-\delta-1)$ ($(p-\delta-1)$ représente une charge courante de $(p-\delta-1)\beta$).

Algorithme 12 : Distribution descendante des tâches.

Entrées : Nombre de blocs et leurs dimensions.
Sorties : Schéma d'allocation des blocs aux processeurs.

```
1 chargeMin ← (p − δ − 1) /* éventuelle charge minimale courante   */ ;
2 numProc ← ⌈p/2⌉ + 1 /* On veut commencer par le processeur P_⌈p/2⌉  */ ;
3 pour d = p à 1 faire
4 |   Rechercher les processeurs éligibles pour la diagonale d ;
5 |   Distribuer les blocs de la diagonale d ;
6 fin
```

Terminologie 5 .

- *Nous disons que le précédent d'un processeur P_s (noté* precedent(s)*) est le processeur P_{s+1} si $s > 1$ et le processeur P_p si $s = 1$;*
- *Si on suppose que les processeurs éligibles sont ordonnés suivant leur numéro, nous disons que le successeur d'un processeur P_s (noté* succ(s)*) est son suivant dans cet ordre.*

L'algorithme 13 permet de rechercher les processeurs éligibles pour une diagonale d donnée.

L'algorithme 14 permet d'attribuer les blocs de la diagonale en cours aux processeurs éligibles. Il est presque identique à celui de la distribution ascendante (algorithme 11).

Lemme 7 *L'allocation des blocs du graphe des tâches aux processeurs par la distribution descendante de données se fait dans le pire des cas en $O(p^3)$ temps d'exécution.*

Preuve. L'algorithme 14 tout comme l'algorithme 11 requiert dans le pire des cas un temps en $O(p)$.

L'algorithme 13 a une complexité temporelle dans le pire des cas en $O(p^2)$. En effet, pendant la recherche des processeurs éligibles pour une diagonale donnée, l'algorithme 13 doit parcourir plusieurs fois l'ensemble des processeurs. Après l'attribution d'un bloc à chaque processeur au début de la distribution des données, l'écart maximal de charges entre les processeurs est de $(\delta-1)\beta$. Si tous les processeurs de la diagonale $(p-\delta+1)$ ne possèdent qu'un seul bloc, alors la différence de charge maximale est de $(\delta-1)\beta$. Et l'algorithme 13 fera au plus $(\delta-1)$ parcours de l'ensemble des processeurs.

Algorithme 13 : Recherche des processeurs éligibles pour la diagonale d.

Entrées : La valeur de $chargeMin$, le numéro d de la diagonale en cours, la valeur de $numProc$ et la charge courante de tous les processeurs.

Sorties : Liste ordonnée des processeurs éligibles de cette diagonale.

```
1  nombreProc ← 0 /* Nombre de processeurs éligibles déjà recensés  */ ;
2  num ← precedent(numProc) ;
3  tant que nombreProc < (p − d + 1) faire
4  |  si (charge(num) ≤ chargeMin) alors
5  |  |  etat(P_num) ← eligible ;
6  |  |  nombreProc ← nombreProc + 1 ;
7  |  fin
8  |  num ← precedent(num) ;
9  |  si (num = p) alors /* Incrémentation de la charge                */
10 |  |  chargeMin ← chargeMin + 1 ;
11 |  fin
12 fin
```

Si par contre d'autres processeurs possédant déjà un bloc sur une autre diagonale [14] évaluent aussi un bloc de cette diagonale $(p-\delta+1)$, alors ces processeurs auront une charge courante de $(2(p-\delta)+1)\beta$ et la différence de charge maximale sera de $(p-\delta+1)\beta$. Cette quantité tend vers p pour de grandes valeurs de p.

La recherche des processeurs éligibles pour une diagonale nécessitera au plus p parcours de l'ensemble des processeurs. Ainsi, l'algorithme 13 recherche les processeurs éligibles en $O(p^2)$ temps d'exécution.

On peut conclure de tout ceci que l'algorithme 12 permet d'effectuer la distribution descendante de données en $O(p^3)$ temps d'exécution. ■

Théorème 5 *La différence de charge maximale entre les processeurs avec la distribution descendante des données est bornée par* $(\delta-1)\beta$.

Preuve. Puisque l'idée de cette distribution est de diminuer au fil des diagonales la différence de charge entre les processeurs, si nous montrons qu'après la diagonale $(p-\delta+1)$, cette différence est inférieure ou égale à $(\delta-1)\beta$, alors à la fin de la distribution, il en sera toujours de même.

La suite de cette preuve est similaire à celle du lemme 7 ■

La figure 4.10 et le tableau 4.2 montrent cette distribution pour $p=7$. Pour ce cas, le déséquilibre de charge est de 3β, les processeurs qui effectuent le plus de calculs sont P_7

14. Il s'agit de certains processeurs éligibles de la diagonale $(p-\delta+2)$, car ayant les plus faibles charges courantes.

Algorithme 14 : Distribution des blocs de la diagonale d.

Entrées : L'ensemble trié des processeurs éligibles pour la diagonale en cours et le numéro d de cette diagonale.

Sorties : Les numéros des processeurs qui évalueront chaque bloc de la diagonale d.

```
1  num ← numéro du premier processeur éligible ;
2  pour i = 1 à p − d + 1 faire
3  |  si ((num − d + 1) ≥ 1) et (B_{num−d+1,num} libre) alors
4  |  |  attribuer le bloc B_{num−d+1,num} à P_num ;
5  |  sinon si (num ≤ p − d + 1) et (B_{num,num+d−1} libre) alors
6  |  |  attribuer le bloc B_{num,num+d−1} à P_num ;
7  |  fin
8  |  num ← succ(num) ;
9  fin
10 num ← numéros du premier processeur éligible ;
11 pour i = 1 à p − d + 1 faire
12 |  tant que B_{i,i+d−1} libre faire
13 |  |  si (P_num disponible) alors
14 |  |  |  attribuer le bloc B_{i,i+d−1} à P_num ;
15 |  |  fin
16 |  |  num ← succ(num) ;
17 |  fin
18 fin
```

et P_4 et celui qui effectue le moins de calculs est P_5. Les processeurs évaluent exactement le même nombre de blocs.

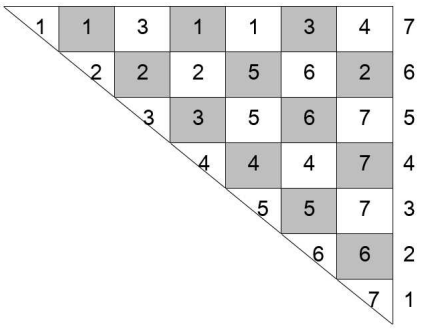


FIGURE 4.10 – Distribution descendante des tâches avec $p = 7$.

Chaque processeur possède des blocs sur les premières et les dernières diagonales du graphe. Et comme l'évaluation d'un bloc commence dès que les premières données utiles sont disponibles, on peut dire que *cette distribution ne favorise pas l'oisiveté des processeurs.*

p	Etapes (ou diagonale)							Bilan	
	7	6	5	4	3	2	1	Total	Msg
1			4β	7β		8β	8β +α	α +8β	4/4
2		5β			7β	8β	8β +α	α +8β	4/4
3		5β			7β	8β	8β +α	α +8β	3/4
4	6β				8β	9β	9β +α	α +9β	3/4
5				3β	5β	6β	6β +α	α +6β	4/4
6			4β	7β		8β	8β +α	α +8β	4/4
7			4β	7β	9β		9β +α	α +9β	3/4

TABLE 4.2 – Évolution des charges des processeurs et nombre de blocs qui permettent de gagner une communication (message) : Cas de 7 processeurs.

Cette distribution minimise assez bien les communications, et équilibre beaucoup mieux les charges entre les processeurs.

Résultat 2 *La distribution descendante des données se fait dans le pire des cas en* $O(p^3)$ *temps d'exécution et le déséquilibre de charges entre les processeurs est borné par* $(\delta - 1)\beta$ *opérations arithmétiques. En plus, cette distribution minimise les communications de manière satisfaisante.*

4.4.3.5 Étude comparative avec la DPBA

Avec les mêmes hypothèses que celles qui sont développés dans notre approche, nous déduisons qu'avec la DPBA, Le processeur $P_{\lceil p/2 \rceil}$ qui effectue le minimum de calculs a une charge inférieure à $(p(p+2)/8)\beta$, et celle du processeur qui effectue le plus de calculs (c'est-à-dire P_1 ou P_p) est supérieure ou égale à $(p^2/4)\beta$. Ainsi, la différence de charges entre les processeurs dans le pire des cas est $(p(p-2)/8)\beta$. Pour $p \in \{4, 5, 6, 7\}$, cette différence de charge est plus petite que celle obtenue avec notre distribution descendante. Mais lorsque $p > 7$, $(p(p-2)/8) > (\delta - 1)$, et ainsi, notre distribution descendante équilibre mieux les charges entre les processeurs que la DPBA. De même, pour $p > 9$, $(p(p-2)/8) > (p-1)$ et de ce fait, notre distribution ascendante équilibre mieux les charges que la DPBA.

Nous présentons dans la figure 4.11 et le tableau 4.3, pour $p = 10$, les différents résultats obtenus avec la DPBA. Nous y constatons que le déséquilibre de charges est de 15β, alors que avec notre distribution descendante, il n'est que de 2β.

Nous pouvons finalement dire que :

1. Comme l'écart entre $(p(p-2)/8)$ et $(\delta - 1)$ (ou encore $(p-1)$) augmente grandement avec le nombre p de processeurs, notre approche descendante (respectivement

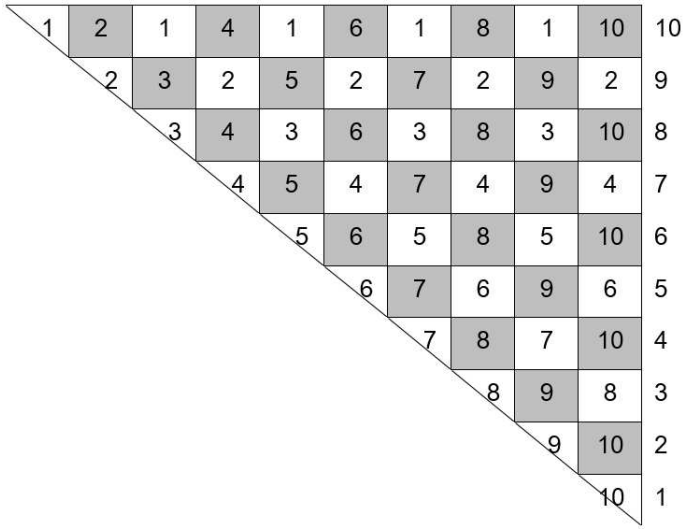


FIGURE 4.11 – Distribution des tâches avec la DPBA pour $p = 10$.

p	Etapes (ou diagonale)										Bilan	
	1	2	3	4	5	6	7	8	9	10	Total	Msg
1	α		β +α		5β +α		11β +α		19β +α		**α + 19β**	5/5
2	α	β +α	3β +α		7β +α		13β +α		22β +α		**α + 22β**	6/6
3	α	β +α	3β +α		7β +α		13β +α				**α + 13β**	5/5
4	α	β +α	3β +α	6β +α	10β +α		16β +α				**α + 16β**	6/6
5	α	β +α	3β +α	6β +α	10β +α						**α + 10β**	5/5
6	α	β +α	3β +α	6β +α	10β +α	15β +α					**α + 15β**	6/6
7	α	β +α	3β +α	6β +α		11β +α					**α + 11β**	5/5
8	α	β +α	3β +α	6β +α		11β +α		18β +α			**α + 18β**	6/6
9	α	β +α		4β +α		9β +α		16β +α			**α + 16β**	5/5
10	α	β +α		4β +α		9β +α		16β +α		25β +α	**α + 25β**	6/6

TABLE 4.3 – Évolution des charges des processeurs et nombre de blocs qui permettent de gagner une communication (Messages) pour la DPBA : Cas de 10 processeurs.

l'approche ascendante) minimise d'avantage la différence de charge entre les processeurs ;

2. Avec la DPBA, le processeur $P_{\lceil p/2 \rceil}$ fini ses calculs après la diagonale $\lceil p/2 \rceil$, et dans chacune des diagonales suivantes, deux processeurs supplémentaires terminent aussi leurs calculs. Avec nos deux approches, tous les processeurs possèdent des blocs des premières diagonales aux dernières. Ainsi, elles minimisent l'oisiveté des processeurs ;
3. La DPBA minimise mieux les communications. Tous les blocs d'un processeur P_i sont toujours soit sur la ligne i, soit sur la colonne i ;
4. La DPBA est bien plus simple à réaliser ;

5. Dans les deux distributions, le nombre de blocs des processeurs est presque le même.

Nous avons proposé dans cette section une méthode de distribution de données qui permet d'attribuer les blocs aux processeurs de proche en proche en essayant à chaque moment d'atteindre des objectifs spécifiques. Dans le cas présent il s'agissait de la différence des charges entre les processeurs et la quantité des communications. Cette distribution peut se réaliser suivant deux scénarios différents : une distribution ascendante plus facile à réaliser mais moins performante et une distribution descendante plus complexe mais avec de meilleures performances.

Le problème non encore résolu dans cet algorithme BSP/CGM générique, même avec l'un de nos scénarios de distribution de données, est le fait que *le nombre de blocs d'un processeur n'est pas borné*. En effet, avec le partitionnement du graphe D'_n considéré, un processeur peut détenir jusqu'à $O(p)$ blocs, et ceci peut être à l'origine de mauvaises performances. Dans la section 4.5, nous proposons une solution dépourvu de cet inconvénient.

4.5 Résolution du problème de parenthésage minimale sur le modèle CGM

Dans cette section, nous proposons un nouvel algorithme BSP/CGM générique pour la résolution de la classe de problèmes abordée. Il a la même structure globale que celui de Kechid-Myoupo [Kec09] mais avec de meilleures performances. Cet algorithme BSP/CGM équilibre mieux les charges des processeurs, minimise le nombre de messages à communiquer, et surtout le nombre de rondes de communication. Il borne à 2 le nombre de blocs d'un processeur, et nécessite $O(n^3/p)$ temps d'exécution avec $\lceil (2p)^{1/2} \rceil$ rondes de communication sur p processeurs.

Nous allons dans la suite de cette section présenter comment nous subdivisons le graphe des tâches en blocs, et ensuite, montrer la dépendance entre les différents blocs. Nous allons terminer par la présentation des différentes étapes de notre algorithme BSP/CGM, ainsi qu'une analyse des résultats expérimentaux obtenus.

4.5.1 Partitionnement du graphe de tâches

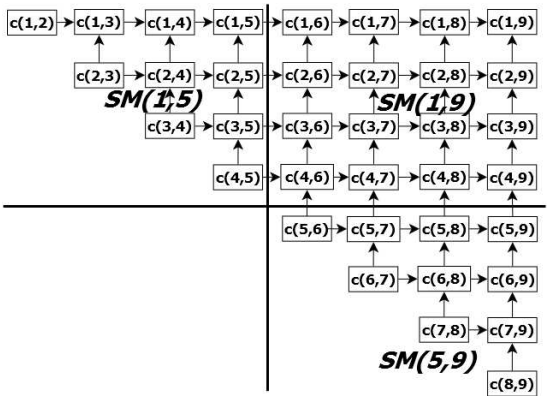
4.5.1.1 Stratégie de partitionnement

En gardant les mêmes notations que dans les sections précédentes, nous divisons le graphe D'_n (la matrice $SP(n,n)$) en $\left(\left(\lceil (2p)^{1/2} \rceil + 1\right) \times \lceil (2p)^{1/2} \rceil\right)/2$ sous-matrices (sous-

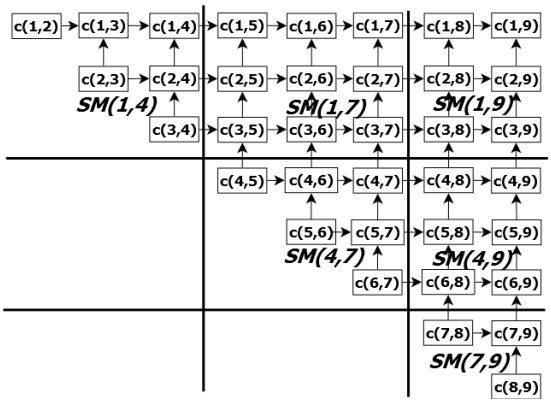
graphes) $SM(i,j)$, ayant chacun $\lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$ lignes et $\lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$ colonnes. La figure 4.12 montre des exemples de ce partitionnement pour $n = 9$ et $p \in \{2,3,4,5\}$.

Terminologie 6 *Chaque sous-matrice de $SP(n,n)$ est appelée bloc. Nous désignons par $SM(i,j)$ un bloc (une sous-matrice) dont l'évaluation permet d'obtenir le meilleur parenthésage possible pour la sous-chaine de matrices $(M_i \times \ldots \times M_j)$ (voir figure 4.12). $SM(1,n)$ désigne le bloc qui permet d'obtenir la solution finale (c'est-à-dire $Cost(1,n)$).*

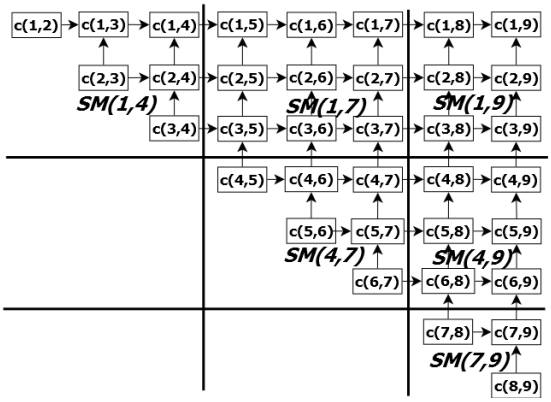
Notation 1 *Dans la suite ce cette section, nous utiliserons les notations suivantes : $f(p) = \lceil (2p)^{1/2} \rceil$ et $\theta(p,n) = \lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$.*



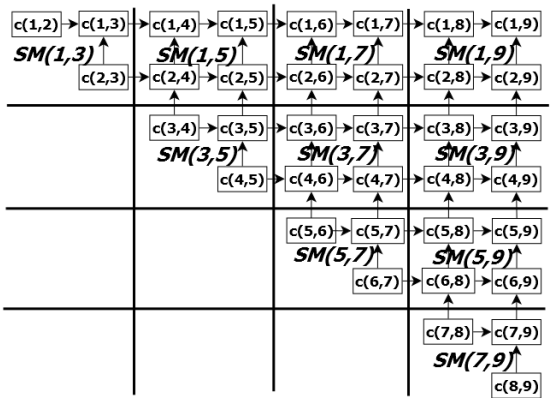
(a) Pour $n = 9$ et $p = 2$, la matrice des plus courts chemins est partitionnée en 3 blocs.



(b) Pour $n = 9$ et $p = 3$, la matrice des plus courts chemins est partitionnée en 6 blocs.



(c) Pour $n = 9$ et $p = 4$, la matrice des plus courts chemins est partitionnée en 6 blocs.



(d) Pour $n = 9$ et $p = 5$, la matrice des plus courts chemins est partitionnée en 10 blocs.

FIGURE 4.12 – Partitionnement de la matrice des plus courts chemins pour $n = 9$ et $p \in \{2,3,4,5\}$

De ce partitionnement résulte $f(p) \times f(p)$ matrices dans lesquelles chaque nœud $SP(i,j), 1 \leq i < j \leq n$ contient la valeur temporaire du plus court chemin vers le nœud (i,j) de D'_n. L'égalité 4.4 montre les entrées de la table $SP(n,n)$ délimitant un bloc $SM(i,j), 1 \leq i < j < n$.

$$SM(i,j) = \begin{pmatrix} SP(i, j-\theta(p,n)+1) & \dots & SP(i,j) \\ SP(i+1, j-\theta(p,n)+1) & \dots & SP(i+1,j) \\ . & \dots & . \\ . & \dots & . \\ SP(i+\theta(p,n)-1, j-\theta(p,n)+1) & \dots & SP(i+\theta(p,n)-1, j) \end{pmatrix} \quad (4.4)$$

Remarque 4 .

1. *Les blocs de la première diagonale, comme la matrice $SP(n,n)$, sont des matrices triangulaires supérieures de $\theta(p,n)$ lignes et $\theta(p,n)$ colonnes ;*
2. *Un bloc est plein s'il est une matrice non triangulaire de taille $\theta(p,n) \times \theta(p,n)$;*
3. *En générale, les blocs de la dernière colonne de blocs (la colonne $f(p)$) dans $SP(n,n)$ ne sont pas plein (ceci est illustré dans la figure 4.12b) ;*
4. *Un bloc $SM(i,j)$ se trouve sur la diagonale de blocs $\lceil (j-i)/\theta(p,n) \rceil$.*

Nous pouvons, de tout ceci, dériver le lemme suivant :

Lemme 8 *Après partitionnement, le graphe D'_n (la table $SP(n,n)$) contient b sous-graphes (blocs matriciels) tel que $p < b \leq 2p$. Ainsi, chaque processeur a à évaluer au plus 2 blocs.*

Preuve. On a $b = \left(\lceil (2p)^{1/2} \rceil + 1\right)\left(\lceil (2p)^{1/2} \rceil\right)/2$ blocs après le partitionnement. Il est clair que $p < b \leq 2p$, et donc, si la distribution des blocs se fait de manière équitable sur les p processeurs, chacun d'eux aura au moins un bloc à évaluer, mais ne pourra pas en avoir plus de deux. ■

4.5.1.2 Dépendance de données

Lemme 9 (Poids des sauts d'un sous-graphe) *Le calcul des poids des sauts partants des sommets d'un sous-graphe $D'_n(i,k)$ vers les sommets d'un sous-graphe $D'_n(i,j)/1 \leq i \leq k \leq j$ nécessite uniquement les valeurs des plus courts chemins vers le sous-graphe $D'_n(k-\theta(p,n)+2, j)$.*

Preuve. Le calcul des poids des sauts qui partent d'un ensemble de sommets successifs dans la ligne i : $\{(i,k),(i,k+1),\ldots,(i,h)\}$, vers un sommet $(i,m)/i \leq k < h < m$ ne nécessite que les valeurs du plus court chemin vers les sommets $\{(k+1,m),\ldots,(h+1,m)\}$ qui est un ensemble de sommets successifs de la colonne m. Ainsi, le calcul des poids des sauts de toute ligne i' du sous-graphe $D'_n(i,k)$, i.e. $(i',k-\theta(p,n)+1),(i',k-\theta(p,n)+2),\ldots,(i',k)$ vers chacun des sommets (i',j') de la ligne i' du sous-graphe $D'_n(i,j)$, i.e. $(i',j-\theta(p,n)+1),(i',j-\theta(p,n)+2),\ldots,(i',j'),\ldots,(i',j)$, ne nécessite que les valeurs du plus court chemin vers l'ensemble de sommets : $\{(k-\theta(p,n)+2,j'),(k-\theta(p,n)+3,j'),\ldots,(k+1,j')\}$.
Or cet ensemble est exactement l'ensemble des sommets de la colonne j' du sous-graphe $D'_n(k-\theta(p,n)+2,j)$ avec $j' \in \{(j-\theta(p,n)+1),(j-\theta(p,n)+2),\ldots,(j-1),j\}$.
Ainsi le calcul des poids des sauts de chacune des lignes de $D'_n(i,k)$ ne nécessite que les valeurs du plus court chemin des sommets du sous-graphe $D'_n(k-\theta(p,n)+2,j)$. ■

Le théorème 6 formalise la dépendance des données entre les blocs de $SP(n,n)$. Il se distingue du théorème 1 par la taille des blocs.

Théorème 6 (dépendance entre blocs) *soit u un entier tel que $u = \lceil (j-i)/\theta(p,n) \rceil$. Le coût du plus court chemin vers chacun des nœuds des blocs $SM(i,j-\theta(p,n))$, $SM(i,j-2\times\theta(p,n))$,... ,$SM(i,j-(u-1)\times\theta(p,n))$, et $SM(i+\theta(p,n),j)$, $SM(i+2\times\theta(p,n),j)$, ..., $SM(i+(u-1)\times\theta(p,n),j)$ sont nécessaires pour évaluer les coûts des plus court chemins vers les nœuds du bloc $SM(i,j)$.*

Preuve. Elle est similaire à celle du théorème 1. ■

La figure 4.7 illustre cette dépendance (voir section 4.4.1.2).

Avec notre nouveau partitionnement du graphe D'_n, les blocs d'une même diagonale sont indépendants. Par conséquent, ils peuvent être évalués en parallèle. L'évaluation des nœuds d'un bloc d'une diagonale donnée, nécessite les valeurs des plus courts chemins vers les nœuds des blocs de toutes les diagonales précédentes.

4.5.2 Algorithme CGM

Notre algorithme BSP/CGM se distingue de celui de Kechid-myoupo [Kec09] essentiellement à deux niveaux : *au niveau du partitionnement du graphe des tâches, et au niveau du schéma de communication de données.* Le théorème 2 est aussi valable pour notre approche. Il permet, comme dans la précédente, d'obtenir deux variantes de notre algorithme BSP/CGM dans lesquelles on peut :

1. commencer l'évaluation des blocs d'une diagonale aussitôt que possible (c'est-à-dire à l'étape $\lceil k/2 \rceil + 1$ pour un bloc de la diagonale k) ; ou bien,
2. commencer l'évaluation des blocs d'une diagonale k après l'évaluation des blocs de la diagonale $k-1$, qui est la dernière diagonale de blocs desquels ils dépendent.

Dans cette thèse, nous présentons uniquement la première variante. La seconde variante peut être aisément déduite de la première.

4.5.2.1 Structure générale de l'algorithme BSP/CGM

Notre algorithme BSP/CGM est une succession de $f(p)$ étapes (itérations) similaires. Dans chacune d'elles, les blocs d'une diagonale (la diagonale en cours) sont évalués en parallèle. L'évaluation commence par la première diagonale, ensuite la seconde, et progresse d'une diagonale à l'autre jusqu'à la dernière. Cet algorithme BSP/CGM dont la structure globale est donnée par l'algorithme 15, permet à la fin des traitements, d'obtenir la valeur de $Cost(1, n)$.

Algorithme 15 : Structure générale de notre algorithme BSP/CGM pour un problème de la classe abordée.

Entrées : La matrice $SP(n, n)$ initialisée à 0, stockée sur un $CGM(n, p)$. (Chaque processeur possède au plus 2 blocs (i.e. $O(n^2/p)$ nœuds) et les valeurs initiales du sous-problème correspondant).

Sorties : Valeur du plus court chemin (i.e. la solution optimale) vers chacun des sommets des sous-graphes détenus par chaque processeur.

```
1 pour k = 1 à f(p) faire
2     Phase 1 : Finalisation du calcul des valeurs des plus courts chemins vers les
      sommets des sous-graphes de la diagonale k ;
3     Phase 2 : Mise à jour des valeurs des plus courts chemins vers les sommets de
      chacun des sous-graphes des diagonales (k+1, k+2, ..., min{2×(k−1), f(p)});
4     Phase 3 : Communication de tout bloc SM(i, j), correspondant à un
      sous-graphe de la diagonale en cours, aux processeurs qui détiennent ses
      sous-graphes supérieurs et ses sous-graphes droits ;
5 fin
```

Comme nous l'avons vu dans la section 4.4, et conformément au théorème 6, les mises à jours pour un bloc $SM(i, j)$ (qui concernent les sauts provenant du bloc $SM(i, k)$) sont équivalentes à une multiplication de matrices $(+, min)$ des matrices $SM(i, k)$ et $SM(k - \theta(p, n) + 2, j)$. L'évaluation d'un bloc de la diagonale k commence dès que la diagonale $\lceil k/2 \rceil$ est évaluée, c'est-à-dire à l'étape $\lceil k/2 \rceil + 1$ de l'algorithme.

La mise à jour d'un bloc $SM(i,j)$ (étape 2 de l'algorithme 15) à l'itération $h+1$ ($\lceil u/2 \rceil \leq h < u$ où $u = \lceil (j-i)/\theta(p,n) \rceil$) est réalisée par l'algorithme 8 (voir section 4.4.2.1) pour des matrices d'ordre $\theta(p,n)$. L'étape 1 est réalisée par l'algorithme 16.

Algorithme 16 : Finalisation de calcul du bloc $SM(i,j)$ à l'itération $u = \lceil (j-i)/\theta(p,n) \rceil$

pour $k = (j-i-\theta(p,n))$ **à** $(j-i)$ **faire**
  pour chaque *sommet s de la diagonale k appartenant à* $D'_n(i,j)$ **faire**
    $SP(s) \leftarrow \min$ {$SP(s)$, poids des chemins dont l'arc final est un saut en provenance de $D'_n(i, i+\theta(p,n))$, poids des chemins dont l'arc final est un saut interne, poids des chemins dont l'arc final est un arc unitaire interne};
  fin
fin

Lemme 10 *La complexité temporelle de l'algorithme 16 pour l'évaluation d'un bloc est en* $O(n^3/(2p)^{3/2})$.

Preuve. La finalisation des calculs à l'étape 1 est similaire à l'algorithme générique séquentiel pour un problème de taille $\theta(p,n) = \lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$. Ainsi, sa complexité temporelle dans le pire des cas est en $O(n^3/(2p)^{3/2})$. ■

Remarque 5 *nous avons dit dans la remarque 4 que les blocs de la dernière colonne de blocs pouvaient avoir moins de colonnes de sommets que les autres. Ainsi, si un bloc* $SM(i,n), 1 \leq i \leq n$ *de cette colonne de blocs n'est pas plein, les traitements qui lui sont associés à chaque étape (multiplication de matrices* $(+, min)$ *ou bien finalisation des calculs) nécessiteront moins de* $O(n^3/(2p)^{3/2})$ *calculs locaux, et de ce fait,* $O(n^3/(2p)^{3/2})$ *est la complexité dans le pire des cas.*

4.5.2.2 Distribution des blocs aux processeurs et complexité

Avec notre algorithme, le nombre maximal de blocs d'une diagonale du graphe D'_n est $f(p)$, et comme $f(p) < p$ le choix des processeurs pour l'évaluation des blocs de chaque diagonale doit être judicieux pour remplir les critères définis dans la section 4.4.2.2 (être simple à réaliser, équilibrer les charges des processeurs, préserver les communications, etc).

Nous distribuons les blocs aux processeurs suivant un schéma proche de notre distribution ascendante de données (voir section 4.4.3.3). Cette distribution (mapping) de

données est schématisée dans la figure 4.13. Elle est simple à implémenter et permet d'obtenir de bonnes prédictions. Dans ce mapping, tous les blocs de la première diagonale sont alloués aux processeurs de la gauche vers la droite. Le processus est repris dans la diagonale suivante en commençant par le processeur $P_{f(p)+1}$, et ainsi de suite jusqu'à ce que chaque processeur possède un bloc. L'allocation recommence par le processeur 1, et se poursuit en *serpentant* l'ensemble des diagonales jusqu'à la diagonale $f(p)$.

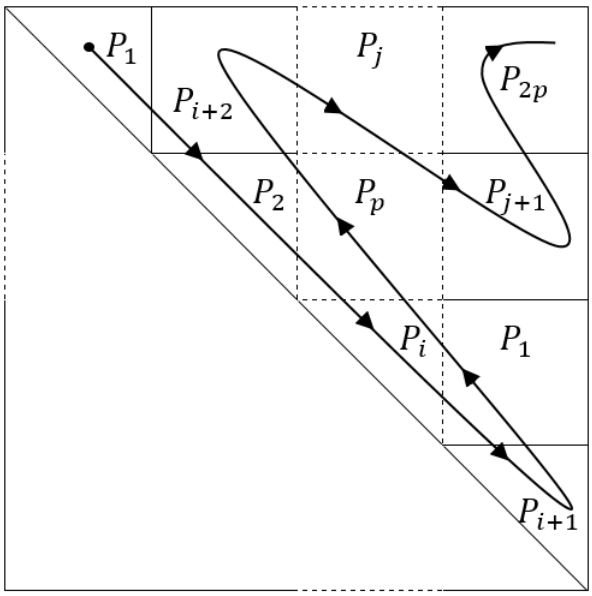


FIGURE 4.13 – Distribution des blocs aux processeurs de manière serpentée.

Théorème 7 *Avec notre subdivision des tâches et notre schéma de communication, l'algorithme 15 nécessite $O(n^3/p)$ temps d'exécution avec $\lceil (2p)^{1/2} \rceil$ rondes de communication.*

Preuve. Dans le pire des cas, chaque étape de calcul est soit un calcul séquentiel pour la finalisation de l'évaluation d'un bloc (algorithme 16), c'est-à-dire similaire à la résolution d'un problème de programmation dynamique d'ordre $\lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$, soit une multiplication de deux matrices d'ordre $\lfloor n/\lceil (2p)^{1/2} \rceil \rfloor$ (algorithme 8). Ainsi, un processeur réalise au plus $O(n^3/(2p)^{3/2})$ calculs locaux à chaque étape. Comme l'algorithme 15 est constitué de $\lceil (2p)^{1/2} \rceil$ étapes, il a une complexité temporelle en $O(n^3/p)$. ■

Notre algorithme BSP/CGM présente les avantages suivants :

1. Le nombre de rondes de communication est réduit à $f(p)$. Comme ce nombre représente les performances de la phase de communication (d'après le modèle de coût des algorithmes CGM, section 2.4.2), notre algorithme minimise le temps de communication ;

2. Le schéma de communication est simple et facile à réaliser. Il permet de faire de bonnes prédictions ;
3. Chaque processeur possède un bloc sur les premières et les dernières diagonales. Ainsi, notre distribution des tâches minimise l'oisiveté des processeurs ;
4. Les processeurs possèdent presque le même nombre de bloc (un ou deux blocs d'après le lemme 8), et comme cette distribution de données est équivalente à notre distribution ascendante (voir section 4.4.3.3) en termes d'équilibrage de charge, on peut dire qu'elle équilibre bien les charges des processeurs ;
5. L'augmentation de la taille du problème augmente la taille des messages à communiquer, mais n'affecte pas leur nombre.

Résultat 3 *Avec notre subdivision des tâches et notre schéma de communication, notre algorithme BSP/CGM permet de résoudre un problème d'ordre n de la classe abordée en* $O(n^3/p)$ *temps d'exécution avec* $\lceil (2p)^{1/2} \rceil$ *étapes. D'après les avantages sus cités, on peut dire que notre algorithme est extensible à l'augmentation de la taille des données et du nombre de processeurs.*

4.5.3 Résultats des simulations

Nous présentons dans cette section les résultats issus de l'implémentation de notre algorithme BSP/CGM, pour la résolution d'un problème MCOP d'ordre n. Ces résultats sont en adéquation avec les prédictions théoriques que nous avions faites.

4.5.3.1 Environnement et outils d'implémentation

Nous avons implémentés cet algorithme sur une machine constituée comme suit :

1. 10 ordinateurs : HP Compaq dx7500 Microtower dual processors, Pentium (R) Dual-Core CPU E5200@2.50GHz, 2.00 GB DDR ECC Registered PC2100, 250 GB hard drive ;
2. Réseau : Ethernet 100 Mbit/s.

Le langage de programmation utilisé est le *C*, sur le système d'exploitation Ubuntu 10.04. La communication inter-processeurs est réalisée via la bibliothèque MPI (version *Open MPI*).

Note 4 *L'environnement de simulation et les outils d'implémentation que nous venons de décrire sont les mêmes pour tous les algorithmes parallèles de cette thèse.*

4.5.3.2 Analyse des simulations

Afin de bien explorer les performances de notre algorithme, les résultats présentés ici, sont issus de son exécution pour différentes valeurs du couple (n, p) dans lesquelles :

- p représente le nombre de processeurs, et prend ses valeurs dans l'ensemble $\{1, 2, 3, 4, 6, 10, 14, 20\}$;
- n représente la taille du problème (nombre de données), et prend ses valeurs dans l'ensemble $\{300, 900, \ldots, 3900\}$. Les n valeurs prises en entrée du problème sont tirées aléatoirement.

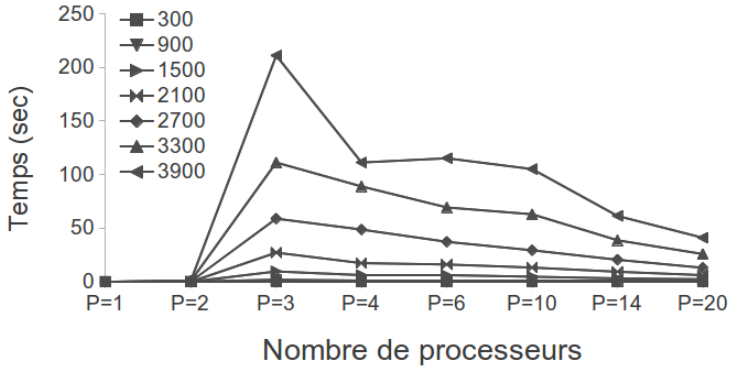


FIGURE 4.14 – Temps global de communication en fonction de p, avec $p \in \{1, 2, 3, 4, 6, 10, 14, 20\}$ et $n \in \{300, 900, \ldots, 3900\}$.

Ces résultats sont compatibles avec ceux prévus à la conception. En effet, la figure 4.14 nous montre que pour $p > 3$, le temps de communication décroît lorsque le nombre de processeurs augmente. Ces résultats révèlent aussi que pour le même nombre de processeurs, ce temps augmente raisonnablement avec la taille du problème. Nous pouvons conclure que cet algorithme est extensible à l'accroissement du nombre de processeurs.

Nous observons des courbes de la figure 4.16 que, le temps globale de communication n'explose pas avec l'accroissement de la taille du problème en entrée. La figure 4.15 indique que, bien qu'il soit plus grand que le temps global de calcul, le temps global de communication affecte de moins en moins le temps global d'exécution de l'algorithme lorsque n croît. Nous pouvons dire que notre algorithme est extensible à l'accroissement de la taille du problème.

En ce qui concerne l'équilibrage de charges, la figure 4.17 montre que 50% des processeurs ont une charge supérieure à la charge moyenne. Les charges en dessous de la moyenne lui sont très proches, et celles qui sont en dessus sont celles des processeurs qui évaluent les blocs des dernières diagonales du graphe D'_n. Elles sont néanmoins plus éloignées de la

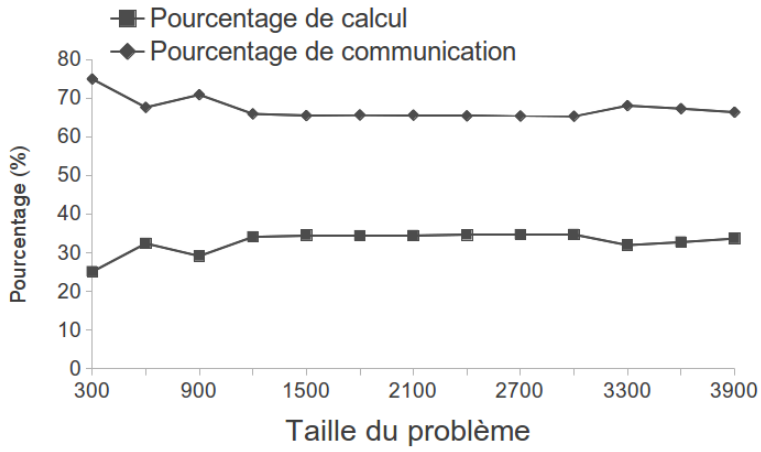


FIGURE 4.15 – Pourcentage de calcul vs pourcentage de communication pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 3900\}$

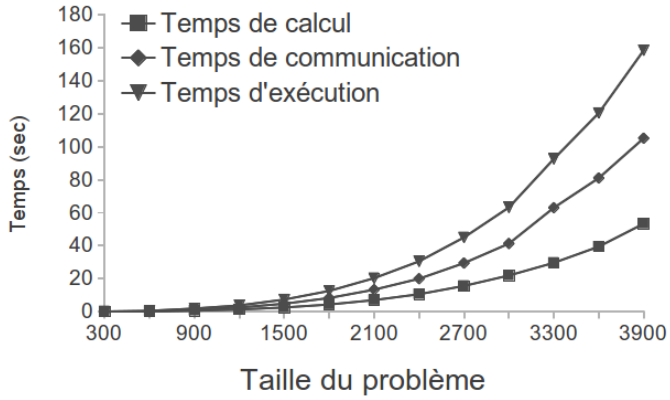


FIGURE 4.16 – Évolution du temps total d'exécution, du temps global de communication et du temps global de calcul pour $p = 10$ et $n \in \{300, 600, 600, \ldots, 3900\}$.

charge moyenne. Ceci était prévisible, car la charge induite par les blocs augmente avec le numéro de la diagonale sur laquelle ils se trouvent (voir section 4.4.3). Nous pouvons donc conclure, en vertu de la proximité des charges des processeurs, que notre algorithme équilibre bien les charges des processeurs.

Remarque 6 *La figure 4.17 permet aussi de valider la méthode de rééquilibrage de charge que nous avons proposés à la section 4.4.3.*

4.6 Synthèse

Nous avons présenté dans ce chapitre deux algorithmes BSP/CGM génériques pour une classe de problèmes de programmation dynamique de type *polyadique non-serial* où

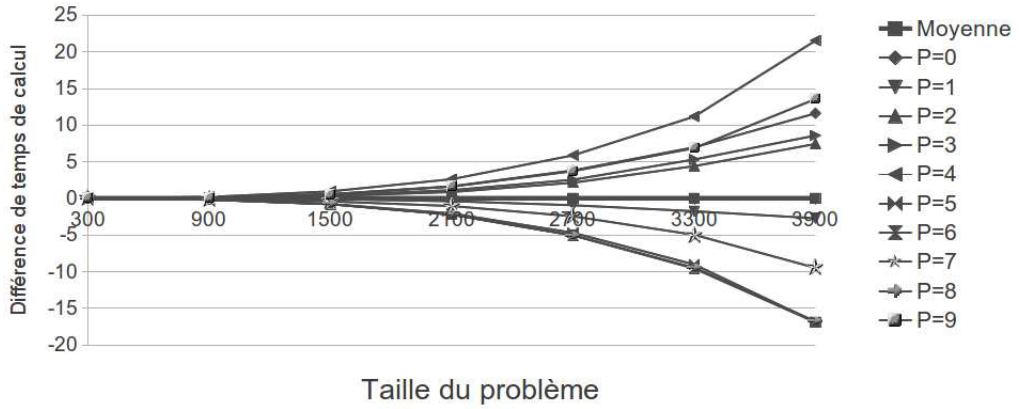


FIGURE 4.17 – Différence de charge relative à la charge moyenne pour $p = 10$ et $n \in \{300, 600, \ldots, 2100\}$.

il y a une forte dépendance entre les sous-problèmes. Le premier peut résoudre un tel problème en $O(n^3/p)$ temps de calcul avec $O(p)$ rondes de communication sur p processeurs, mais équilibre peu les charges de ceux-ci. Nous avons proposés une méthode d'allocation des blocs aux processeurs permettant d'équilibrer significativement leurs charges. Elle se décline suivant deux scénarios : *une distribution ascendante facile à réaliser, mais moins efficace, et une distribution descendante plus complexe, mais avec de meilleures performances*.

Nous avons proposés par la suite un algorithme BSP/CGM qui a le même temps de calcul par processeur que le précédent, mais avec $\lceil (2p)^{1/2} \rceil$ rondes de communication. Il est donc meilleur que le précédent qui en compte $O(p)$. Avec cet algorithme, le nombre de messages à communiquer n'explose ni avec l'augmentation de la taille des données, ni avec celle du nombre de processeurs utilisé. Ainsi, il est extensible à l'accroissement de la taille du problème et du nombre de processeurs.

Notre approche peut être plus performante si chaque processeur utilise l'algorithme de Coppersmith-Winograd [CW90] pour la multiplication matricielle. Ainsi, le temps de calcul sera en $O(n^{2.376}/p)$. Un autre moyen est d'utiliser des algorithmes optimaux dans les phases de calculs locaux. Les *« accélérations en programmation dynamique »* constituent dans ce cas une solution naturelle.

Chapitre 5

Accélération en programmation dynamique

Sommaire

5.1 Introduction

Dans ce chapitre nous proposons deux algorithmes BSP/CGM dans lesquelles, les calculs locaux sont réalisés à l'aide d'algorithmes séquentiels accélérés. L'objectif étant de réduire le temps global de calcul des processeurs et ainsi obtenir de meilleures performances.

Dans la deuxième section, nous proposons un algorithme BSP/CGM, basé sur l'algorithme séquentiel de Knuth [Knu71], pour la résolution du problème de la recherche de l'arbre binaire de recherche optimal. Cet algorithme nécessite $O(n^2/p)$ temps d'exécution avec $\lceil (2p)^{1/2} \rceil$ rondes de communication sur p processeurs, chacun étant muni de $O(n^2/p)$ espace mémoire. Dans la section 3, en utilisant des spécificités liées à l'accélération de

Knuth (voir section 3.4.2), nous proposons une méthode permettant de réduire le temps de latence des processeurs et ainsi, améliorer les performances de notre algorithme. Dans la section 4, pour le problème d'ordonnancement de produit de chaine de matrices, nous proposons un algorithme BSP/CGM basé sur l'accélération de Yao [Yao82]. Il nécessite au plus $(p+1)$ rondes de communication avec $O(n^2/p)$ calculs séquentiels sur chaque processeur.

5.2 Résolution du problème de la recherche de l'Arbre Binaire de Recherche Optimal sur le modèle CGM

Les contraintes de parallélisation liées à l'algorithme séquentiel de Knuth (section 3.4.2) rendent difficile la conception d'un algorithme BSP/CGM qui tire efficacement profit de l'accélération de Knuth. A notre connaissance, il existe peu d'algorithmes parallèles qui ont réussi à hériter de cette accélération [Kec09].

Le graphe des tâches pour un problème OBST d'ordre $n = 7$ est présenté dans la figure 5.1.

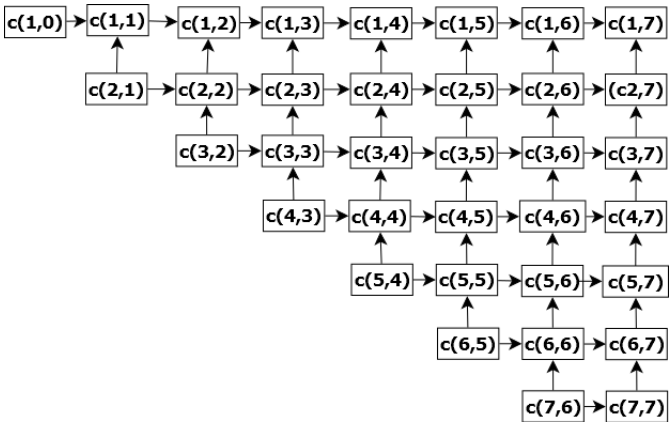


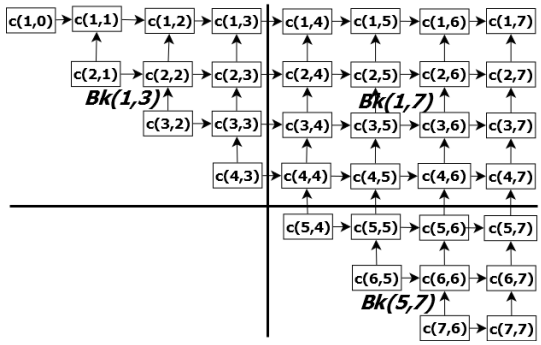
FIGURE 5.1 – Graphe des tâches pour un problème OBST d'ordre $n = 7$, avec $c(i,j) = Cost(i,j)$.

5.2.1 Partitionnement du graphe de tâches

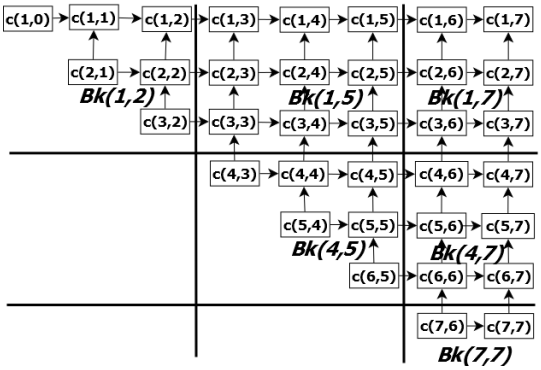
5.2.1.1 Stratégie de partitionnement

Notation 2 *Dans cette section, nous utiliserons les notations suivantes :* $f(p) = \lceil (2p)^{1/2} \rceil$ *et* $g(p,n) = \lceil (n+1)/\lceil (2p)^{1/2} \rceil \rceil$.

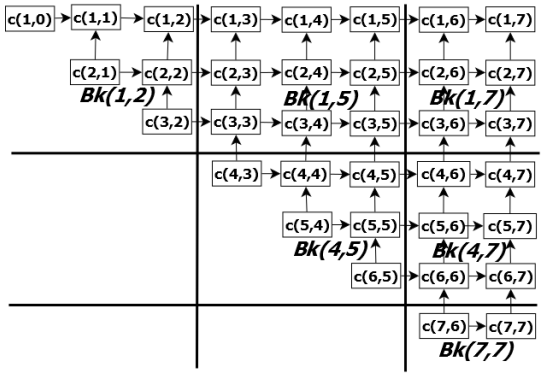
Nous partitionnons la matrice triangulaire supérieure de la figure 5.1 (ou graphe des tâches) en $\frac{(f(p)+1)\times f(p)}{2}$ sous-matrices. Ces sous-matrices sont encore appelées blocs, et notées $Bk(i,j)$. Ces blocs sont des matrices de dimensions $g(p,n) \times g(p,n)$ exceptés ceux de la dernière colonne. La figure 5.2 montre des exemples de ce partitionnement pour $n = 7$ et $p \in \{2,3,4,5\}$.



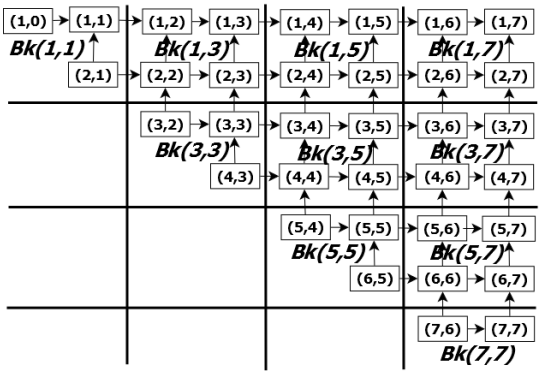
(a) Pour $n = 7$ et $p = 2$, la table de programmation dynamique est partitionnée en 3 blocs.



(b) Pour $n = 7$ et $p = 3$, la table de programmation dynamique est partitionnée en 6 blocs.



(c) Pour $n = 7$ et $p = 4$, la table de programmation dynamique est partitionnée en 6 blocs.



(d) Pour $n = 7$ et $p = 5$, la table de programmation dynamique est partitionnée en 10 blocs.

FIGURE 5.2 – Partitionnement du graphe des tâches pour $n = 7$ et $p \in \{2,3,4,5\}$

Remarque 7 .

1. *Les blocs de la première diagonale, sont des matrices triangulaires supérieures de $g(p,n)$ lignes et $g(p,n)$ colonnes ;*
2. *Un bloc est plein s'il est une matrice non triangulaire de taille $g(p,n) \times g(p,n)$;*
3. *En générale, les blocs de la dernière colonne de blocs (la colonne $f(p)$) ne sont pas plein (ceci est illustré dans les figures 5.2b et 5.2c) ;*

4. Un bloc $Bk(i,j)$ se trouve sur la diagonale de blocs $\lceil (j-i+1)/g(p,n) \rceil$.

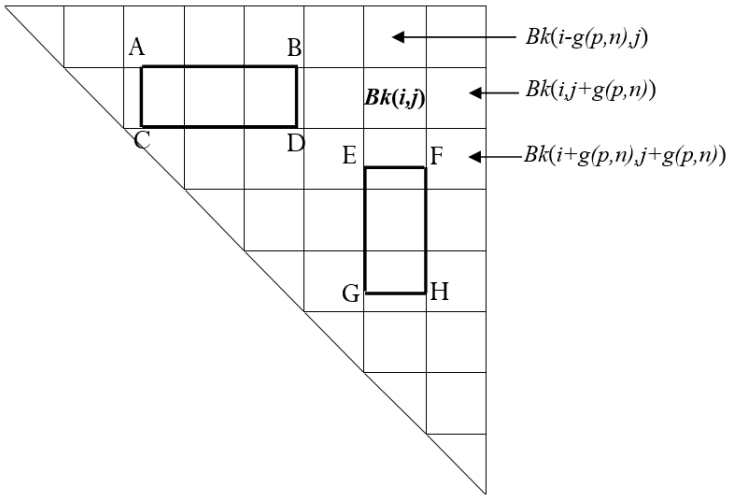
Ceci nous permet d'énoncer le lemme suivant :

Lemme 11 *Il y a au plus $2p$ blocs après le partitionnement du graphe des tâches. Ainsi, chaque processeur a à évaluer au plus 2 blocs.*

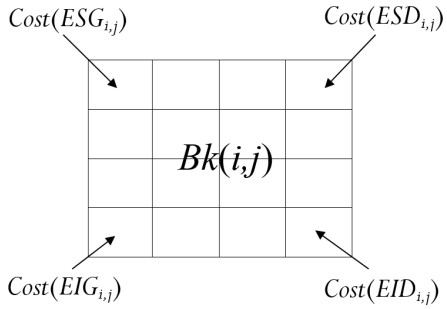
5.2.1.2 Dépendance de données et allocation des tâches

Dépendance de données :

Les figures 5.3a et 5.3b montrent les dépendances entre le bloc $Bk(i,j)$, $i < j \leq n$ et les autres blocs du graphe. On constate qu'ici, l'évaluation du bloc $Bk(i,j)$ de la diagonale $u = \lceil (j-i+1)/g(p,n) \rceil$ ne dépend pas, comme dans les cas précédents, des blocs de toutes les diagonales inférieures à u.



(a) Dépendance des données pour le bloc $Bk(i,j)$.



(b) Entrées du graphe des tâches qui délimitent le bloc $Bk(i,j)$.

FIGURE 5.3 – Dépendance de données et entrées du graphe des tâches qui délimitent un bloc $Bk(i,j)$.

Les entrées du graphe des tâches qui délimitent le bloc $Bk(i,j)$ sont désignées comme suit :

- $ESG_{i,j}$ signifie « entrée supérieure gauche de $Bk(i,j)$ » ;
- $ESD_{i,j}$ signifie « entrée supérieure droit de $Bk(i,j)$ » ;
- $EIG_{i,j}$ signifie « entrée inférieure gauche de $Bk(i,j)$ » ;
- $EID_{i,j}$ signifie « entrée inférieure droit de $Bk(i,j)$ ».

Elles sont définies comme suit :

- $ESG_{i,j} = Cost(i, j - g(p,n) + 1)$;
- $ESD_{i,j} = Cost(i, j)$;
- $EIG_{i,j} = Cost(i + g(p,n) - 1, j - g(p,n) + 1)$;
- $EID_{i,j} = Cost(i + g(p,n) - 1, j)$.

Ceci permet d'énoncer le théorème suivant [Kec09] :

Théorème 8 (dépendance de données) *le calcul des valeurs des entrées du bloc $Bk(i,j)$ nécessite dans le pire des cas, les valeurs déjà calculées des entrées des deux blocs délimités par les extrémités (ABCD) et (EFGH) tel que :*
$A = Cost(i, Cut(ESD_{i,j-g(p,n)}) - 1)$
$B = Cost(i, Cut(i + g(p,n), j) - 1)$
$C = Cost(i + g(p,n) - 1, Cut(ESD_{i,j-g(p,n)}) - 1)$
$D = Cost(i + g(p,n) - 1, Cut(i + g(p,n), j) - 1)$
$E = Cost(Cut(ESD_{i,j-g(p,n)}) + 1, j - g(p,n) + 1)$
$F = Cost(Cut(ESD_{i,j-g(p,n)}) + 1, j)$
$G = Cost(Cut(i + g(p,n), j) + 1, j - g(p,n) + 1)$
$H = Cost(Cut(i + g(p,n), j) + 1, j)$

Preuve. D'après l'inégalité de Knuth (inégalité 3.7), on a :
Pour toute entrée $c(r,x)$ contenue dans $Bk(i,j) : Cut(ESG_{i,j}) \leq Cut(r,x) \leq Cut(EID_{i,j})$
or $Cut(ESD_{i,j-g(p,n)}) = Cut(i, j - g(p,n)) \leq Cut(ESG_{i,j})$ et
$Cut(EID_{i,j}) = Cost(i + g(p,n) - 1, j) \leq Cut(i + g(p,n), j)$
Ainsi, pour toute les entrées $c(r,x)$ appartenant au bloc $Bk(i,j)$, on a :
$Cut(ESD_{i,j-g(p,n)}) \leq Cut(r,x) \leq Cut(i + g(p,n), j)$
On en déduit directement que les extrémités A et D correspondent respectivement aux entrées :
$Cost(i, Cut(ESD_{i,j-g(p,n)}) - 1)$ et $Cost(i + g(p,n) - 1, Cut(i + g(p,n), j) - 1)$ du graphe des tâches.
ABCD étant un rectangle, les extrémités B et C peuvent être directement déduites. Elles correspondent aux entrées :
$Cost(i, Cut(i + g(p,n), j) - 1)$ et $Cost(i + g(p,n) - 1, Cut(ESD_{i,j-g(p,n)}) - 1)$
Un simple remplacement dans la récurrence de programmation dynamique de l'OBST donne pour les extrémités du bloc EFGH les mêmes cases données dans l'énoncé du théorème. ■

Le théorème 8 montre que les blocs de la même diagonale de blocs sont indépendants, donc peuvent être évalués en parallèle.

Distribution des blocs aux processeurs :

Nous distribuons les blocs aux processeurs avec notre distribution serpentée (voir section 4.5.2.2). Commençant à la première diagonale par le processeur P_1, ensuite P_2, jusqu'à $P_{f(p)}$, poursuivant sur la seconde diagonale par $P_{f(p)+1}$ et continuant de manière serpentée jusqu'à la diagonale $f(p)$.

5.2.2 Algorithme CGM

Notre algorithme BSP/CGM évalue les blocs du graphe des tâches diagonale après diagonale, commençant par la première diagonale jusqu'à la diagonale $f(p)$. Mais, à cause la nature des dépendances entre les blocs, on ne peut pas prédire, avant le début du calcul des valeurs des entrées du graphe, quelles seront les valeurs nécessaires à l'évaluation d'un bloc $Bk(i,j)$. En d'autres termes, les emplacements des extrémités des rectangles ABCD et EFGH ne sont pas connus avant le début des traitements. Ainsi, une évaluation progressive des blocs d'une diagonale k n'est plus possible[1]. De ce fait, notre algorithme débute et termine l'évaluation du bloc $Bk(i,j)$, appartenant à la diagonale $u = \lceil (j-i+1)/g(p,n) \rceil$, à l'étape u. À cette étape, d'après le théorème 8, les valeurs des entrées du graphe dont dépend l'évaluation du bloc $Bk(i,j)$ sont connues. L'algorithme 17 donne la structure globale de notre algorithme BSP/CGM. $TP(n,n)$ désigne la table de programmation dynamique.

Il est clair que l'algorithme 17 utilise $f(p)$ étapes (rondes de communication). Dans chacune d'elles, les blocs d'une diagonale sont évalués : au plus un bloc par processeur. L'algorithme utilisé lors des calculs locaux est l'algorithme séquentiel de Knuth (algorithme 4). Après l'évaluation de chaque diagonale k, les entrées des tables $TP(n,n)$ et $Cut(n,n)$ de cette diagonale sont communiqués aux processeurs des blocs des diagonales $\{k+1, k+2, \ldots, f(p)\}$ qui en ont besoin pour évaluer leur bloc.

Les performances de notre algorithme sont données par le théorème 9.

Théorème 9 *En utilisant p processeurs, notre algorithme BSP/CGM résout un problème de recherche de l'arbre binaire de recherche optimal d'ordre n avec $f(p)$ rondes de communication et $O(n^2/p)$ temps de calculs locaux sur chaque processeur.*

Preuve. L'algorithme séquentiel de Knuth utilisé dans les phases de calculs locaux de

1. A l'aide des multiplications de matrices $(+, min)$ comme au chapitre 4.

Algorithme 17 : Structure générale de notre algorithme BSP/CGM pour une instance du problème OBST de taille n sur p processeurs.

Entrées : Les tables $TP(n,n)$ et $Cut(n,n)$ et la liste des $(2n+1)$ probabilités. Chaque processeur possède au plus 2 blocs (i.e. $O(n^2/p)$ nœuds et les valeurs initiales du sous-problème correspondant).

Sorties : Les tables $TP(n,n)$ et $Cut(n,n)$ calculées.

pour $k = 1$ **à** $f(p)$ **faire**

1. Calculer les valeurs des entrées des blocs de la diagonale k en utilisant l'algorithme 4 ;
2. Communiquer les entrées (des tables $TP(n,n)$ et $Cut(n,n)$ correspondant aux blocs évalués) nécessaires à l'évaluation des blocs des diagonales de blocs $\{k+1, k+2, \ldots, f(p)\}$;

fin

notre algorithme CGM nécessite $O((n+1)^2/(2p)^{2/2}) = O(n^2/2p)$ calculs locaux pour chaque diagonale de blocs. Comme chaque processeur évalue au plus 2 blocs, on peut conclure que cet algorithme nécessite $O(n^2/p)$ temps de calculs locaux sur chaque processeur. ■

5.2.3 Résultats des simulations

Dans cette section, nous présentons les résultats des expérimentations effectués dans les mêmes conditions que ceux de la section 4.5.3.1. Ensuite nous faisons une comparaison des tests issus des simulations de notre algorithme BSP/CGM basé sur l'algorithme séquentiel de Knuth et notre algorithme BSP/CGM générique.

Remarque 8 .

1. *Le temps de communication considéré dans cette thèse est la somme du temps de transfert effectif des données et du temps d'attente des processeurs ;*
2. *Les tests sont effectués pour les différents nombre de processeurs utilisés sur les mêmes données. Ceci permet d'observer, pour le même ensemble de données, les résultats obtenus pour différents nombre de processeurs.*

Terminologie 7 *Nous désignons par « OBST-Sans-Knuth » notre algorithme BSP/CGM générique (c'est-à-dire sans l'accélération de Knuth) et « OBST-Avec-Knuth » notre algorithme basé sur l'accélération de Knuth.*

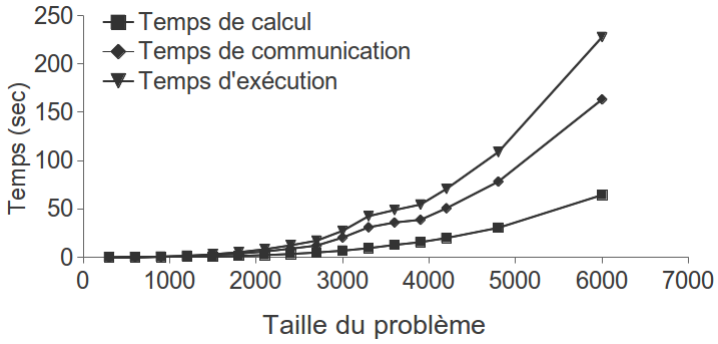


FIGURE 5.4 – Évolution du temps total d'exécution, du temps global de communication et du temps global de calculs pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 4200, 4800, 6000\}$.

Résultats obtenus pour notre algorithme basé sur l'accélération de Knuth

La figure 5.4 montre que le temps de calcul est très inférieur au temps de communication. De même, la figure 5.5 montre que le pourcentage de calcul est très inférieur au pourcentage de communication. Ceci est dû au gain de temps de calcul provenant de l'accélération de Knuth.

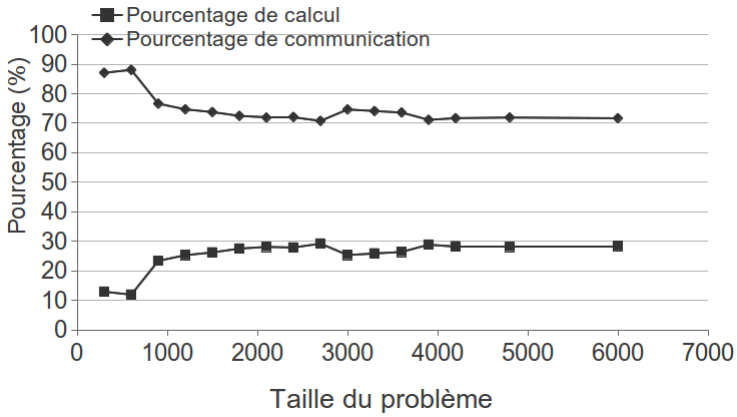


FIGURE 5.5 – Pourcentage de calcul vs pourcentage de communication pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 4200, 4800, 6000\}$.

Les fluctuations dans les courbes de la figure 5.5 sont dues au gain de calculs hérité de l'accélération de Knuth, qui varie de manière imprévisible pour les entrées d'une même diagonale du graphe des tâches. Les figures 5.4 et 5.5 montrent que notre algorithme *OBST-Avec-Knuth* garde de bonnes performances lorsque la taille tu problème augmente.

La figure 5.6 montre que la différence de charge entre les processeurs n'est pas grande (en fait, elle ne peut pas être plus grande que celle obtenue avec l'algorithme *OBST-Sans-*

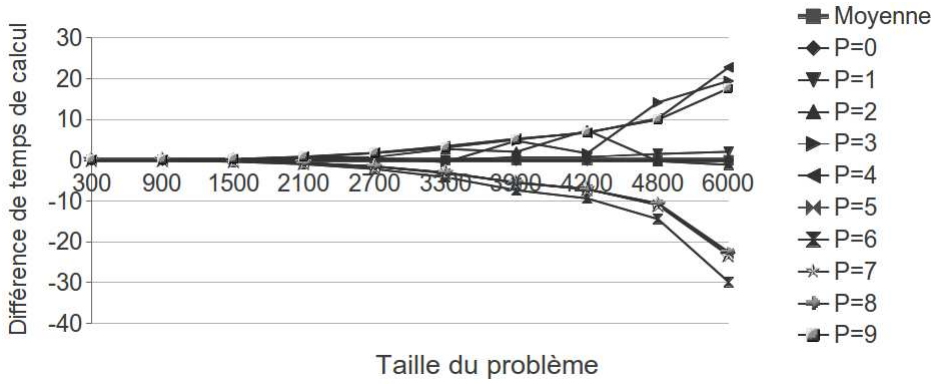


FIGURE 5.6 – Différence de charge par rapport à la charge moyenne pour $p = 10$ et $n \in \{300, 900, \ldots, 4200, 4800, 6000\}$.

Knuth). Les charges des processeurs inférieures à la charge moyenne évoluent de manière uniforme (elles sont toujours décroissantes), et celles au-dessus de la charge moyenne évoluent en dents de scie. Ceci vient du fait que les processeurs associés font partie de ceux qui évaluent plus d'un bloc du graphe. Ils évaluent un bloc des dernières diagonales de blocs où l'accélération de Knuth est le plus sensible [2]. L'accélération de Knuth à moins d'impact sur les blocs des premières diagonales.

Les courbes représentant le temps de communication (figure 5.7) sont similaires à celles obtenues sans l'accélération de Knuth (*OBST-Sans-Knuth*). Ceci est normal car les partitionnements des graphes des tâches utilisés sont similaires et le schéma de communication est le même.

Toutefois, ce qui est surprenant ici est que : le temps global de communication pour l'al-

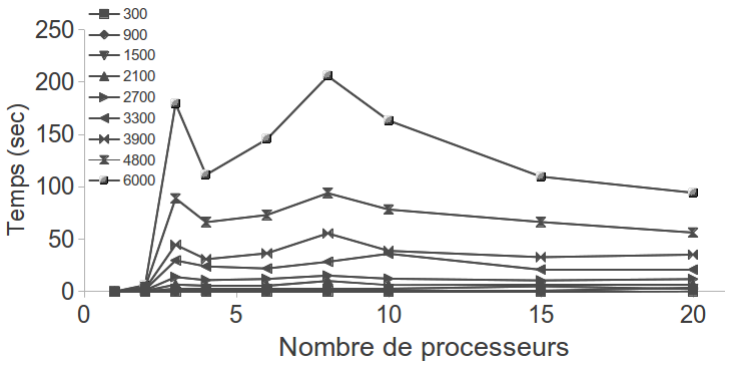


FIGURE 5.7 – Temps global de communication en fonction de p avec $p \in \{1, 2, 3, 4, 6, 8, 10, 15, 20\}$ et $n \in \{300, 900, \ldots, 3900, 4800, 6000\}$.

2. Le gain en nombre de calculs peut être plus grand pour les entrées des blocs des dernières diagonales du graphe des tâches que pour les entrées des blocs des premières.

gorithme *OBST-Avec-Knuth* est beaucoup plus faible que celui obtenu avec l'algorithme *OBST-Sans-Knuth* alors qu'il y a deux fois plus de données à transférer (les entrées des tables $TP(n,n)$ et $Cut(n,n)$). La raison est donnée par le théorème 10 :

Théorème 10 *Pour la classe de problèmes considérée, et pour le même partitionnement des données, l'augmentation du temps de calcul des différents blocs du graphe des tâches entraine l'augmentation du temps global de communication de l'algorithme.*

Preuve. Puisqu'un processeur P_i attend un processeur P_j parce qu'il n'a pas encore fini l'évaluation de son bloc dont P_i a besoin, le temps d'attente d'un processeur dépend aussi du temps de calcul des processeurs qui détiennent les blocs dont les valeurs des entrées lui sont utiles. Ce temps d'attente est très grand lorsque l'accélération de Knuth n'est pas utilisée [3]. Telle est le cas pour notre algorithme *OBST-Sans-Knuth*. Ainsi, le temps global de communication (temps d'attente et temps de transfert effectif des données) de l'algorithme *OBST-Sans-Knuth* est plus grand que celui de l'algorithme *OBST-Avec-Knuth* (dont le temps de transfert effectif des données est deux fois plus grand) à cause de la forte latence des processeurs. ■

Comparaison des résultats des expérimentations de nos algorithmes OBST-Avec-Knuth et OBST-Sans-Knuth

L'interprétation des courbes présentant les performances des algorithmes *OBST-Avec-Knuth* et *OBST-Sans-Knuth*, montre l'avantage que procure l'accélération de Knuth.

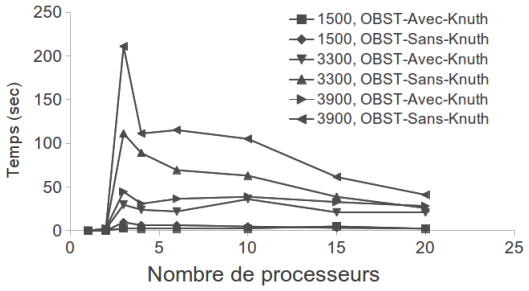
La figure 5.8a montre exactement ce qui a été prédit dans le théorème 10. Sur 10 processeurs, le temps de communication a diminué en moyenne de $42,147\%$.

La proportion de calcul est bien plus faible avec l'algorithme *OBST-Avec-Knuth* qu'avec l'algorithme *OBST-Sans-Knuth* (figure 5.8b). Ceci est essentiellement due à la diminution du temps de calcul qui vient de l'algorithme séquentiel de Knuth utilisé.

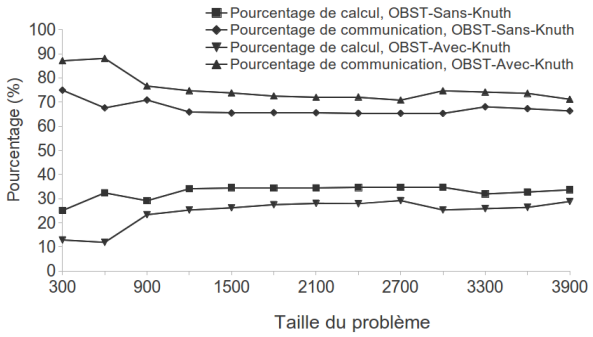
La figure 5.8c montre que l'algorithme *OBST-Avec-Knuth* a un temps d'exécution beaucoup plus faible que l'algorithme *OBST-Sans-Knuth* (en moyenne $53,475\%$ de moins pour $p = 10$). La figure 5.8d montre qu'il en est de même pour le temps de calcul (en moyenne $67,720\%$ de moins pour $p = 10$).

La figure 5.8e présente pour les deux algorithmes, les courbes représentant les différences de charges de calcul (chaque algorithme ayant sa propre charge moyenne). Ces courbes sont celles des processeurs ayant les plus grandes ou les plus petites charges de calcul. De cette figure, on observe que la différence de charge entre les processeurs avec

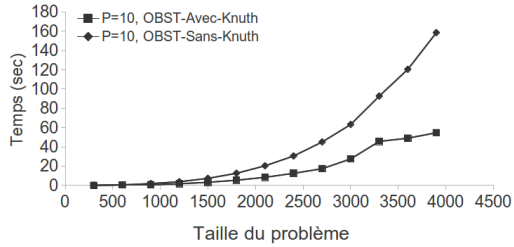
3. Cette accélération permet de diminuer énormément le nombre de calculs locaux effectués par un processeur, et ainsi, de diminuer son temps global de calcul.



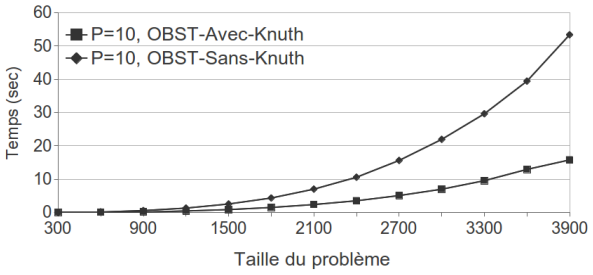
(a) Temps global de communication en fonction de p pour les algorithmes *OBST-Sans-Knuth* et *OBST-Avec-Knuth*, pour $p = 10$ et $n \in \{1500, 3300, 3900\}$.



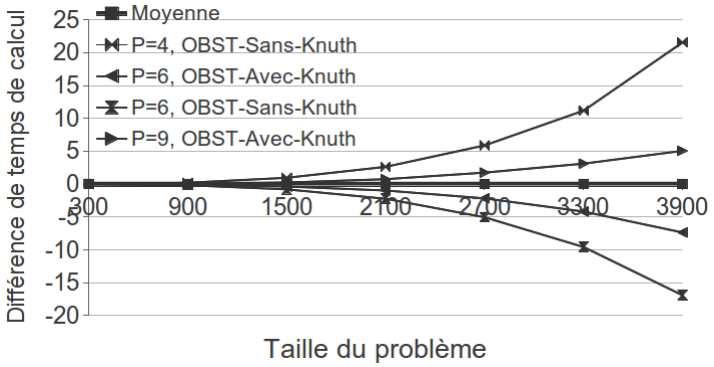
(b) Pourcentage de calcul vs pourcentage de communication pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 3900\}$ pour les algorithmes *OBST-Sans-Knuth* et *OBST-Avec-Knuth*.



(c) Temps total d'exécution pour les algorithmes *OBST-Sans-Knuth* et *OBST-Avec-Knuth* pour $p = 10$ et $n \in \{300, 600, \ldots, 3900\}$.



(d) Temps de calcul local pour les algorithmes *OBST-Sans-Knuth* et *OBST-Avec-Knuth* pour $p = 10$ et $n \in \{300, 600, \ldots, 3900\}$.



(e) Différence de charge relative à la charge moyenne pour les algorithmes *OBST-Avec-Knuth* et *OBST-Sans-Knuth* pour $p = 10$ et $n \in \{300, 900, \ldots, 3900\}$.

FIGURE 5.8 – Comparaison des performances des algorithmes *OBST-Sans-Knuth* et *OBST-Avec-Knuth*.

l'algorithme *OBST-Avec-Knuth* a diminuée en moyenne de $54,475\%$ par rapport à celle obtenue avec l'algorithme *OBST-Sans-Knuth*.

Remarque 9 *Avec l'algorithme OBST-Avec-Knuth, il n'est pas possible de prédire à l'avance quel processeur aura la plus faible ou bien la plus grande charge de calcul. En effet, ceci vient de l'accélération de Knuth qui cause l'irrégularité du nombre de calculs nécessaires à l'évaluation des entrées d'une même diagonale du graphe des tâches. Par exemple, dans la figure 5.8e, P_9 a la plus grande charge alors que P_4 aurait été attendu.*

Résultat 4 *Avec notre subdivision des tâches et notre schéma de communication des données, notre algorithme BSP/CGM basé sur l'algorithme séquentiel de Knuth résout le problème OBST en $O(n^2/p)$ temps de calculs sur chaque processeur avec $f(p)$ rondes de communication. La diminution du temps des calculs locaux a permis la diminution du temps d'attente des processeurs, et donc la diminution du temps de communication. Cet algorithme équilibre mieux les charges des processeurs à cause de l'algorithme séquentiel et du schéma de communication utilisés. Il est extensible à l'accroissement de la taille des données et du nombre de processeurs.*

Les analyses nous montrent que notre algorithme *OBST-Avec-Knuth* est efficace et extensible, mais peut être amélioré à plusieurs niveaux :

1. *Au niveau des communications* : les blocs sont communiqués entre les processeurs suivant le schéma de communication utilisé même s'ils ne sont pas finalement utilisés ;
2. *Au niveau des calculs* : le caractère non progressif des traitements [4] entraine une latence inutile des processeurs. En effet, pour commencer l'évaluation d'un bloc d'une diagonale k, un processeur attend que tous les blocs des diagonales inférieures à k soient évalués. Pourtant, il n'a pas besoin des données issues de toutes ces diagonales.

Nous proposons dans la section suivante une première piste de solution à ces problèmes.

5.3 Accélération de l'algorithme CGM par la minimisation de temps de latence des processeurs

L'objectif à atteindre dans cette section est double :

4. L'évaluation des blocs d'une diagonale k commence et se termine à l'étape k de l'algorithme.

1. réduire au minimum le nombre de blocs communiqués par les processeurs sans changer le schéma de communication ;
2. commencer l'évaluation d'un bloc aussitôt que les valeurs des entrées des blocs desquels il dépend sont disponibles.

Ces objectifs peuvent être atteints à l'aide de l'accélération de Knuth. En effet, l'évaluation des entrées d'un bloc $Bk(i,j)$ ne dépend pas en pratique des valeurs des entrées de tous les blocs $Bk(i,l)$, $i < l < j$ et $Bk(m,j)$, $i < m < n$ (se référer à la figure 5.3 de la section 5.2.1.2 pour plus de détails).

Hypothèse :

Notre méthode est basée sur le fait que quel que soit le bloc BR contenant le bloc $Bk(i,j)$, le parallélogramme $A'B'C'D'$ délimitant les entrées desquels dépend l'évaluation du bloc BR, contient le parallélogramme $ABCD$ qui délimite les entrées desquels dépend l'évaluation du bloc $Bk(i,j)$. La figure 5.9 montre que si le parallélogramme BR est constitué du bloc $Bk(i,j)$ et d'une partie du bloc $Bk(i, j-g(p,n))$, alors le parallélogramme $ABCD$ sera contenu dans le parallélogramme $A'BC'D$. Il en est de même pour les parallélogrammes $EFGH$ et $EFG'H'$.

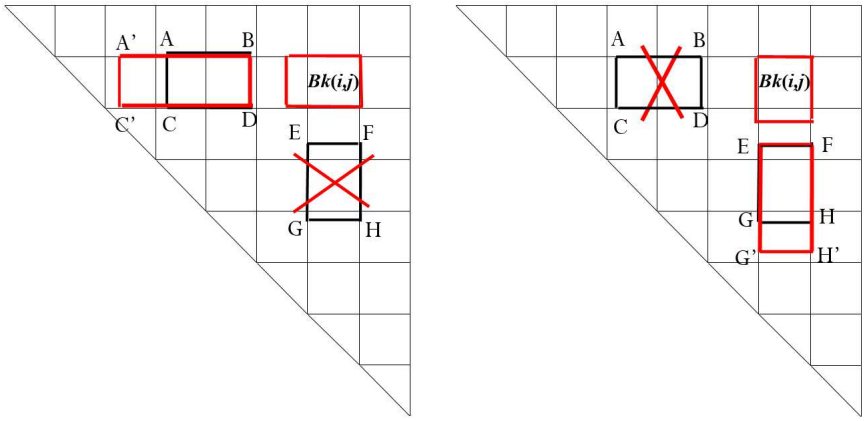


FIGURE 5.9 – Dépendance entre les blocs.

Terminologie 8 .

1. *Nous désignons par $Bk(i, j-g(p,n), j)$ le bloc constitué de l'union des blocs $Bk(i, j-g(p,n))$ et $Bk(i,j)$. Les parallélogrammes $ABCD$ et $EFGH$ délimitent les entrées du graphe des tâches dont les valeurs sont nécessaires à l'évaluation du bloc $Bk(i,j)$. Les parallélogrammes $A'B'C'D'$ et $E'F'G'H'$ délimitent celles du bloc $Bk(i, j-g(p,n), j)$;*

2. *Nous désignons par $P_{i,j}$ le processeur qui évalue le bloc $Bk(i,j)$;*
3. *Un bloc $Bk(i,j)$ se trouve sur la diagonale de blocs $u = \lceil (j-i+1)/g(p,n) \rceil$.*

A partir de la terminologie 8, on peut énoncer le lemme suivant :

Lemme 12 *Les parallélogrammes $ABCD$ et $EFGH$ sont contenus respectivement dans les parallélogrammes $A'B'C'D'$ et $E'F'G'H'$.*

Preuve. Il est évident que A et A' sont sur la même ligne i du graphe. Montrons que A' est à gauche de A.
En effet, $A' = Cost(i, Cut(ESD_{i,j-2\times g(p,n)}) - 1)$ et $A = Cost(i, Cut(ESD_{i,j-g(p,n)}) - 1)$ toutefois, $ESD_{i,j-2\times g(p,n)} = (i, j - 2 \times g(p,n))$ tandis que $ESD_{i,j-g(p,n)} = (i, j - g(p,n))$ d'après l'accélération de Knuth, $Cut(ESD_{i,j-2\times g(p,n)}) \leq Cut(ESD_{i,j-g(p,n)})$ et ainsi, A' est à gauche de A.
Il est claire que B est égale à B' et un raisonnement similaire montre que C' est à gauche de C. De même, G' et H' sont respectivement en dessous de G et H. ■

Lemme 13 (Poids des sauts d'un sous-graphe) *Lors de l'évaluation d'un bloc $Bk(i,j)$, les valeurs des entrées d'un bloc $Bk(i,m), 1 \leq i < m < j$ sont couplées avec celles du blocs $Bk(m - g(p,n) + 3, j)$.*

Preuve. Elle est similaire à celle du lemme 9 (section 4.5.1.2). ■

5.3.1 Amélioration des communications

Un processeur après avoir évalué un bloc $Bk(i,j)$ de la diagonale u, ne communique pas ses résultats à tous les processeurs des blocs supérieurs et blocs droits. Il effectue le transfert soit aux processeurs des blocs des deux diagonales suivantes (les diagonales $(u+1)$ et $(u+2)$), soit aux processeurs des autres diagonales uniquement s'il n'a reçu aucun message d'un autre processeur lui demandant de ne pas effectuer le transfert.

Le processeur $P_{i,j}$ après avoir fini d'évaluer le bloc $Bk(i,j)$ de la diagonale $u > 1$, calcule la valeur de $A' = Cost(i, Cut(ESD_{i,j}) - 1)$. Ensuite, calcule les coordonnées du bloc $Bk(i,m)$ dans lequel A' se trouve. Toutes les valeurs des blocs $Bk(i,l)$, $l < m$ ne seront pas utilisées pour l'évaluation des blocs $Bk(i,s)$, $j < s$. Le processeur $P_{i,j}$ envoie un message à tous les processeurs $P_{i,l}$, $l < m$ pour leur demander de ne pas diffuser leur bloc au processeurs $P_{i,s}$, $j < s$.

Similairement, comme les valeurs des blocs $Bk(i,l)$, $l < m$ et $Bk(l - g(p,n) + 3, s)$ sont couplées pour évaluer les blocs $Bk(i,s)$, $j < s$ (lemme 13), si le bloc $Bk(i,l)$, $l < m$

n'est pas utilisé, alors les blocs $Bk(l - g(p,n) + 3, s)$, $j < s$ ne le seront pas aussi, et en conséquence, ne doivent pas être distribués aux processeurs $P_{i,s}$, $j < s$. Ainsi, le processeur $P_{i,j}$ sachant que les blocs $Bk(i,l)$, $l < m$ ne seront pas utilisés pour l'évaluation des blocs $Bk(i,s)$, $j < s$, il envoie un message à chacun des processeurs $P_{l-g(p,n)+3,s}$, $j < s$ pour qu'il ne diffuse pas son bloc aux processeurs $P_{i,s}$, $j < s$.

Un raisonnement similaire permet au processeur $P_{i,j}$ lorsqu'il a fini d'évaluer le bloc $Bk(i,j)$, d'évaluer la valeur de $G' = Cost(Cut(ESD_{i,j}) + 1, j - g(p,n) + 1)$ Ensuite, calculer les coordonnées du bloc $Bk(e,j)$ qui contient G'. Et enfin, envoyer un message aux processeurs $P_{r,j}$, $r > e$ et $P_{x,r+g(p,n)-3}$, $x < i$ pour qu'ils ne communiquent pas les valeurs de leurs blocs au processeurs $P_{x,j}$, $x < i$.

En définitive, le processeur $P_{1,n}$ ne diffuse pas son bloc. Le processeur $P_{i,j}$ après avoir évalué son bloc $Bk(i,j)$ de la diagonale u, sait si ses résultats seront utilisés pour l'évaluation des blocs de la diagonale suivante (la diagonale $(u+1)$) ou pas. Si ce bloc n'est pas utile pour la diagonale suivante dans l'une ou l'autre direction (au-dessus ou à droite), il ne le diffuse pas. Sinon, s'il n'a pas encore reçu de message lui demandant de ne pas diffuser $Bk(i,j)$ dans l'une ou l'autre direction, il effectue la diffusion vers les processeurs des blocs des deux diagonales suivantes (les diagonales $(u+1)$ et $(u+2)$). À partir de là, et à chacune des ronde de communication, $P_{i,j}$ diffuse le bloc $Bk(i,j)$ aux processeurs des blocs des diagonales suivantes (la diagonale $(u+3)$, ensuite $(u+4)$ et ainsi de suite). La diffusion de $Bk(i,j)$ s'arrête dans l'une des direction, soit lorsque les processeurs des blocs de toutes les diagonales concernées sont servies, soit lorsque le processeur $P_{i,j}$ reçoit un message lui demandant de ne plus diffuser son bloc dans cette direction (au-dessus ou à droite).

Toutefois, lorsqu'un processeur ne doit pas envoyer son bloc à un autre, il le lui notifie en lui envoyant un *message d'information*. Après avoir reçu un tel message d'un processeur $P_{i,m}$, le processeur $P_{i,j}$ sait qu'il n'utilisera pas les données des blocs $Bk(i,l)$, $l < m$ et ceux des blocs $Bk(l-g(p,n)+3,j)$. De même, lorsque $P_{i,j}$ sait qu'il n'utilisera pas le bloc $Bk(e,j)$, $e > i$, il déduit qu'il n'utilisera pas les blocs $Bk(r,j)$, $r > e$ et $Bk(i,r+g(p,n)-3)$.

Comme le processeur $P_{i,j}$ peut savoir, à une étape $d < u$ de l'algorithme, qu'il possède déjà toute les données nésessaires à l'évaluation du bloc $Bk(i,j)$, il peut commencer et terminer cette évaluation à l'étape d (c'est-à-dire avant l'étape u).

5.3.2 Amélioration des calculs

Il a été démontré que l'évaluation d'un bloc de la diagonale u du graphe des tâches ne peut commencer qu'après l'étape $\lceil u/2 \rceil$ de l'algorithme (théorème 2, section 4.4.2).

À cause de l'accélération de Knuth, le processeur $P_{i,j}$, à une étape $d / \lceil u/2 \rceil < d \leq u$, peut déjà posséder toute les données dont il a besoin pour finaliser l'évaluation du bloc $Bk(i,j)$. Ainsi, en considérant les contenus des messages déjà reçus (*contenu des blocs* et *messages d'information*), si tous les blocs des diagonales $s > d$ ne sont pas utiles pour l'évaluation du bloc $Bk(i,j)$, le processeur $P_{i,j}$ peut commencer et terminer ses calculs à l'étape d.

Contrainte :

Le problème qui se pose est le suivant : les valeurs des entrées de la dernière colonne du bloc $Bk(i, j - g(p,n))$ [5] et celles de la première ligne du bloc $Bk(i + g(p,n), j)$ [6], non disponibles à l'étape $d < u$, sont nécessaires pour déterminer les plages de valeurs utiles permettant d'évaluer respectivement les entrées de la première ligne et de la dernière colonne du bloc $Bk(i,j)$. Dans cette section, nous proposons une méthode permettant de résoudre ce problème et commencer les calculs aussitôt que possible.

Terminologie 9 *Nous appelons « entrée critique » une entrée de la première colonne ou de la dernière ligne du bloc $Bk(i,j)$.*

Solution 1 :

Une solution possible est de considérer, pour chaque entrée critique du bloc $Bk(i,j)$, et pour l'une ou les deux extrémités de la plage de valeurs utiles, toutes les entrées (de la même ligne/colonne que cette entrée critique) de tous les blocs *reconnues* utiles.
Comme les extrémités des plages des valeurs utiles des entrées non critiques du bloc $Bk(i,j)$ sont déduites de celles des entrées critiques, il est possible de commencer l'évaluation des blocs d'une diagonale $u > 1$ avant l'étape u de l'algorithme.

L'inconvénient de cette première solution est le suivant : pour les entrées critiques, une ou les deux extrémités de la plage de valeurs utiles sont inconnues [7], et comme le nombre d'opérations à effectuer (définies par les extrémités de la plage de valeurs utiles) pour leur évaluation peut être très différent d'une entrée à l'autre, beaucoup de calculs peuvent être effectués inutilement.

5. Bloc qui précède $Bk(i,j)$ dans la ligne i.
6. Bloc qui suit $Bk(i,j)$ dans la colonne j.
7. Elles sont obtenues à partir des valeurs déjà calculées des entrées des blocs de la diagonale (u-1), or ils ne sont pas encore évalués.

Solution 2 :

Pour résoudre ce problème, considérons les bloc $Bk(i,m)$ et $Bk(e,j)$, les deux blocs desquels dépendent l'évaluation du bloc $Bk(i,j)$, situés sur les diagonales les plus élevées. Soit le couple (l,c) représentant respectivement le nombre de lignes et le nombre de colonnes du bloc $Bk(i,j)$. Considérons l'entrée critique $B_{i,j}(h,f)$ de $Bk(i,j)$ à évaluer. $BCut_{i,m}(h,c)$ désigne l'entrée (h,c) de la table $Cut(n,n)$ correspondant à l'entrée (h,c) du bloc $Bk(i,m)$.

À partir du lemme 12, on peut dire que : $BCut_{i,m}(h,c) \leq BCut_{i,j-g(p,n)}(h,c)$ pour $m \leq j - g(p,n)$ et $BCut_{i+g(p,n),j}(1,f) \leq BCut_{e,j}(1,f)$ pour $i + g(p,n) \leq e$. Ainsi, comme l'extrémité $BCut_{i,j-g(p,n)}(h,c)$ de la plage de valeurs des entrées critiques $B_{i,j}(h,1)$, $1 \leq h \leq l$ n'est pas disponible à l'étape m de l'algorithme, on peut utiliser l'extrémité $BCut_{i,m}(h,c)$. De même, comme l'extrémité $BCut_{i+g(p,n),j}(1,f)$ de la plage de valeurs des entrées critiques $B_{i,j}(l,f)$, $1 \leq f \leq c$ n'est pas disponible à l'étape m de l'algorithme, on peut utiliser $BCut_{e,j}(1,f)$.

Puisque ce sont les valeurs des blocs des diagonales $\{1,\ldots,m\}$ qui ont permis de délimiter les blocs utiles pour l'évaluation de $Bk(i,j)$, et comme les valeurs que nous proposons sont les plus proches des valeurs réelles qu'on peut obtenir à cette étape, nous pouvons dire que nous avons diminué au minimum le nombre de calculs qu'on peut effectuer à cette étape pour évaluer $Bk(i,j)$.

Note 5 *La solution utilisée dans la suite est la solution 2.*

5.3.3 Algorithme CGM

La structure globale de notre nouvel algorithme BSP/CGM que nous désignons *SpeedUp-OBST-Avec-Knuth* est donnée par l'algorithme 18.

Les calculs locaux sont effectués à l'aide de l'algorithme séquentiel de Knuth (algorithme 4). Un processeur commence et fini l'évaluation d'un bloc de la diagonale u à l'étape $k/k \leq u$. Une ronde de communication est divisée en trois phases (phases 0, 2 et 3).

À l'étape k de l'algorithme 18, l'estimation de la plage de valeurs non utiles pour l'évaluation des blocs des diagonales $\{k+1, k+2, \ldots, f(p)\}$ est effectuée par l'algorithme 19. Dans cet algorithme, les terminologies suivantes sont utilisées :

Terminologie 10 .

Algorithme 18 : Structure générale de notre algorithme BSP/CGM pour une instance du problème OBST de taille n sur p processeurs.

Entrées : Les tables $TP(n,n)$ et $Cut(n,n)$ et la liste des $(2n+1)$ probabilités.
Sorties : Les tables $TP(n,n)$ et $Cut(n,n)$ calculés.

1 Évaluation des blocs de la première diagonale en utilisant l'algorithme séquentiel de Knuth ;
2 Communication des blocs de la première diagonale aux processeurs des blocs supérieurs et blocs droits des diagonales 2 et 3 ;
3 **pour** $k = 2$ **à** $f(p)$ **faire**
4 | **Phase 1 : Réception** des blocs des diagonales inférieures et/ou des messages d'information, et mise à jour de la liste des blocs utiles ou non ;
5 | **Phase 2 : Calcul** des valeurs des entrées des blocs des diagonales $m \in \{k, k+1, k+2, \ldots, f(p)\}$ dont les valeurs utiles sont déjà disponibles, et qui ne sont pas encore évaluées;
6 | **Phase 3 : Estimation** de la plage de valeurs non utiles pour les blocs supérieurs et blocs droits des diagonales $\{m+1, m+2, \ldots, f(p)\}$ à partir des blocs évalués à la phase 2. Ensuite, envoie si nécessaire, des messages d'information aux processeurs concernés ;
7 | **Phase 4 : Communication** de tout bloc évalué à la phase 2, sauf contre indication, aux processeurs des blocs supérieurs et blocs droits des diagonales $m+1$ et $m+2$, et si nécessaires, ceux des diagonales $\{1, 2, \ldots, k-1\}$ aux processeurs concernés de la diagonale $k+2$.
8 **fin**

- *msg.abs.unused*$(Bk(i,l), Bk(i,j))$ *est un message indiquant que les valeurs des entrées des blocs* $Bk(i,m)$ *,* $m < l$ *ne seront utilisées pour aucun des blocs droit du bloc* $Bk(i,j)$ *(c'est-à-dire les blocs* $Bk(i,c)$ *,* $j < c$*);*
- *msg.ord.unused*$(Bk(e,s), Bk(i,s))$ *est un message indiquant que les valeurs des entrées des blocs* $Bk(y,s)$ *,* $y > e$ *ne seront utilisées pour aucun des blocs supérieurs du bloc* $Bk(i,s)$ *(c'est-à-dire les blocs* $Bk(v,s)$*,* $v < i$*);*
- *msg.bloc.unused*$(Bk(e,s), Bk(i,j))$ *est un message qui indique que les entrées du bloc* $Bk(e,s)$ *ne seront pas utilisées pour évaluer le bloc* $Bk(i,j)$*.*

Comme nous utilisons le partitionnement des tâches et la distribution des données de l'algorithme *OBST-Avec-Knuth*, chaque processeur évalue au plus deux blocs du graphe.

Théorème 11 *L'algorithme 18 résout un problème de recherche de l'arbre binaire de recherche optimale d'ordre n avec* $O(n^2/p)$ *calculs locaux et* $f(p)$ *rondes de communication sur p processeurs.*

Algorithme 19 : Estimation de la plage des données non utiles à partir du bloc $Bk(i,j)$.

$A' \leftarrow Cut(ESD_{i,j}) - 1$;
$G' \leftarrow Cut(ESD_{i,j}) + 1$;
$m \leftarrow \lceil (A'+1)/g(p,n) \rceil$;
pour chaque *processeur* $P_{i,d}/d < m \times g(p,n) - 1$ **faire**
  $msg.abs.unused(Bk(i, m \times g(p,n) - 1), Bk(i,j))$;
  pour chaque *processeur* $P_{d-g(p,n)+3,s}/j < s$ **faire**
    $msg.bloc.unused(Bk(d - g(p,n) + 3, s), Bk(i,s))$;
  fin
fin
$e \leftarrow \lceil (G')/g(p,n) \rceil$-1 ;
pour chaque *processeur* $P_{c,j}/c > e \times g(p,n) + 1$ **faire**
  $msg.ord.unused(Bk(e \times g(p,n) + 1, j), Bk(i,j))$;
  pour chaque *processeur* $P_{x,c+g(p,n)-3}/x < i$ **faire**
    $msg.bloc.unused(Bk(x, c + g(p,n) - 3), Bk(x,j))$;
  fin
fin

Preuve. L'algorithme 19, exécuté à la fin de l'évaluation d'un bloc, a une complexité temporelle de $O(p)$. Il est exécuté au plus deux fois par un processeur car chacun d'eux a au plus deux blocs du graphe à évaluer.
Le reste du déroulement de l'algorithme 18 est similaire à l'algorithme 17 dans lequel la quantité des données communiquées est réduite et les calculs pour l'évaluation d'un bloc commencent aussitôt que possible. ■

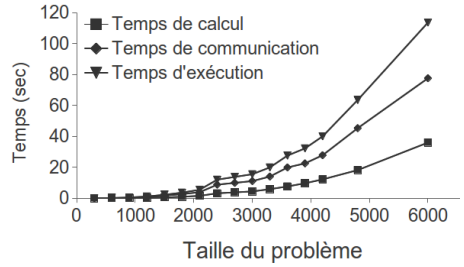
5.3.4 Résultats des simulations

Résultats obtenus pour notre algorithme SpeedUp-OBST-Avec-Knuth :

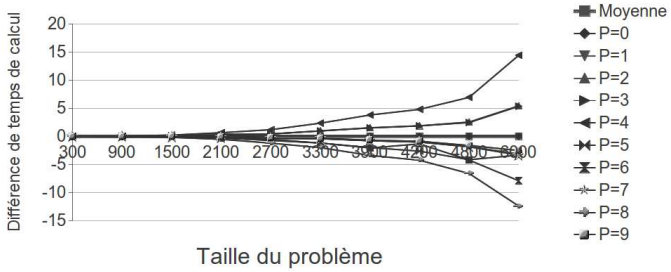
La figure 5.10 montre que les courbes relatives à l'algorithme *SpeedUp-OBST-Avec-Knuth* ont les mêmes propriétés et les mêmes progressions que celles de l'algorithme *OBST-Avec-Knuth*.

Comparaison des résultats des expérimentations de nos algorithmes OBST-Avec-Knuth et SpeedUp-OBST-Avec-Knuth :

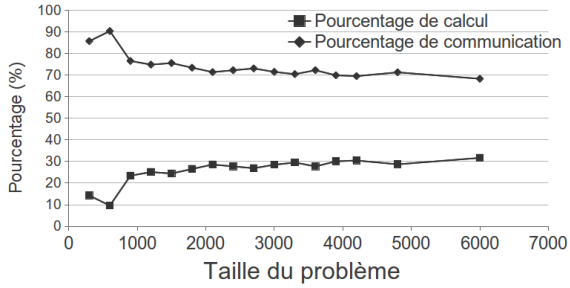
La figure 5.11d montre qu'avec l'algorithme *SpeedUp-OBST-Avec-Knuth* le temps de communication diminue en moyenne de $32,178\%$ par rapport à celui de l'algorithme *OBST-Avec-Knuth*. De plus, l'écart entre les courbes devient de plus en plus grand lorsque



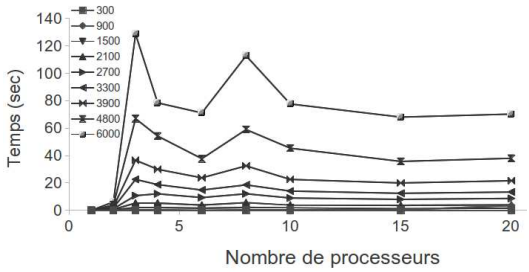
(a) Évolution du temps total d'exécution, du temps global de communication et du temps global de calculs pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 4200, 4800, 6000\}$.



(b) Différence de charge par rapport à la charge moyenne pour $p = 10$ et $n \in \{300, 900, \ldots, 4200, 4800, 6000\}$.



(c) Pourcentage de calcul vs pourcentage de communication pour $p = 10$ et $n \in \{300, 600, 900, \ldots, 4200, 4800, 6000\}$.



(d) Temps global de communication en fonction de p avec $p \in \{1, 2, 3, 4, 6, 8, 10, 15, 20\}$ et $n \in \{300, 900, \ldots, 3900, 4800, 6000\}$.

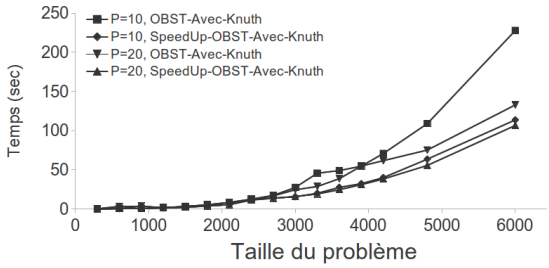
FIGURE 5.10 – Différentes courbes relatives à l'algorithme *SpeedUp-OBST-Avec-Knuth*.

le nombre de données augmente. Ceci montre que nos améliorations affectent d'avantage le temps de communication lorsque la taille du problème augmente.

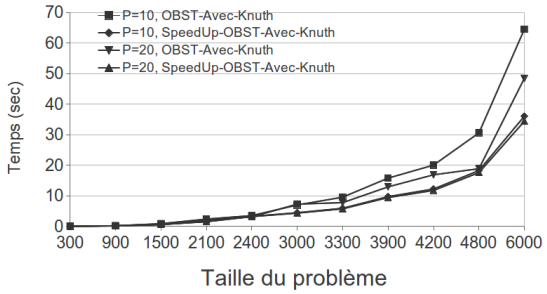
La figure 5.11b montre que le temps global de calcul a diminué en moyenne de $34,535\%$. Ceci était prévisible car avec notre exécution anticipée des blocs,*plusieurs diagonales de blocs qui étaient évalués séquentiellement, peuvent maintenant être évaluées en même temps par des processeurs différents.* Ainsi, plus il y aura de processeurs (donc de diagonales de blocs dans le graphe des tâches), plus il y aura des diagonales qui pourront être évaluées en même temps, et donc, le temps global de calcul pourra être diminué (par rapport au temps qu'on aurait obtenu avec l'algorithme *OBST-Avec-Knuth*).

La figure 5.11c montre que les pourcentages de calcul et de communication sont restés presque les mêmes. Ceci est normal car le temps global de communication et le temps global de calcul ont tous deux diminués.

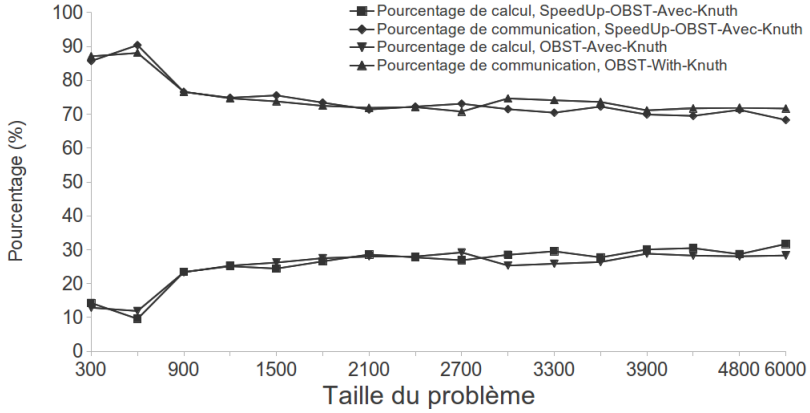
Résultat 5 *Notre méthode de diminution du nombre de communications inutiles permet*



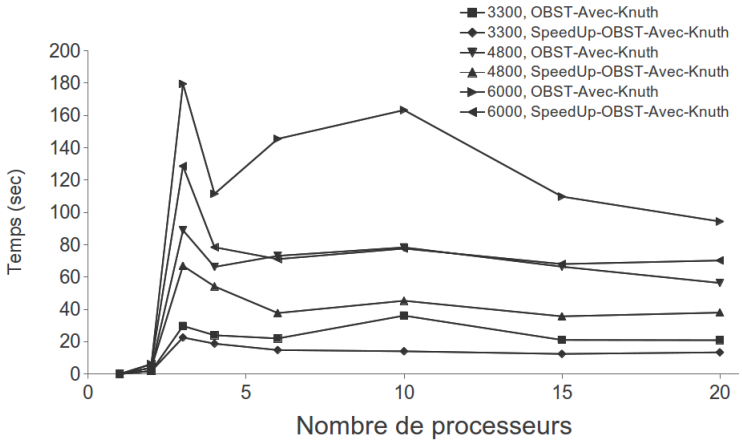
(a) Temps total d'exécution pour *SpeedUp-OBST-Avec-Knuth* et *OBST-Avec-Knuth* avec $p \in \{10, 20\}$ et $n \in 300, 900, 600, \ldots, 6000$.



(b) Temps global de calcul pour *SpeedUp-OBST-Avec-Knuth* et *OBST-Avec-Knuth* avec $p \in \{10, 20\}$ et $n \in \{300, 900, \ldots, 6000\}$.



(c) Pourcentage de calcul vs pourcentage de communication pour *SpeedUp-OBST-Avec-Knuth* et *OBST-Avec-Knuth* avec $p = 10$ et $n \in \{300, 600, 900, \ldots, 6000\}$.



(d) Temps global de communication en fonction de p pour *SpeedUp-OBST-Avec-Knuth* et *OBST-Avec-Knuth* avec $p \in \{1, 2, 3, 4, 6, 10, 15, 20\}$ et $n \in \{1500, 3300, 4800, 6000\}$.

FIGURE 5.11 – Comparaison des performances des algorithmes *SpeedUp-OBST-Avec-Knuth* et *OBST-Avec-Knuth*.

de réduire le temps de transfert effectif des données entre les processeurs, ainsi que le temps d'attente des processeurs. Elle permet aussi de débuter et terminer l'évaluation d'un bloc aussitôt que les données nécessaires sont disponibles. Ainsi il est possible d'évaluer plusieurs blocs des diagonales différentes en parallèle.
Tout ceci a permis de diminuer le temps global de communication et le temps global de calcul. D'où la diminution du temps global d'exécution de l'algorithme.

5.4 Résolution du problème d'Ordonnancement de Produit de Chaine de Matrices sur le modèle CGM

Pour la résolution du problème MCOP, l'algorithme séquentiel de Yao (vu à la section 3.4.3 du chapitre 3) repose sur la bijection qu'il a réussi à démontrer entre les problèmes MCOP et TCP. Dans cette section, nous proposons un algorithme BSP/CGM basé sur l'algorithme séquentiel de Yao, ainsi que quelques-unes de ses variantes.

5.4.1 Graphe de dépendance

L'accélération de Yao vient des spécificités du DAG-multi-niveaux utilisé, notons ce DAG G_n. G_n est en fait une forêt constituée de deux arbres et contient au plus $2n$ nœuds (sous-problèmes). L'évaluation d'un sous-problème (i, j) dépend des valeurs déjà calculées des sous-problèmes (i, k) et (k, j), fils de (i, j) dans G_n. Comme nous l'avons dit à la section 3.4.3, la difficulté induite par ce graphe est que sa structure est fonction de l'instance du problème à résoudre. Cette structure est différente même pour deux produits de matrices de même longueur si les matrices sont de dimensions différentes.
Par exemple, la figure 3.10 de la section 3.4.3 et la figure 5.12 montrent deux polygones de même taille avec des sommets de poids différents (deux produits de matrices de mêmes longueurs mais avec des matrices de dimensions différentes). Les DAG-multi-niveaux correspondant à ces polygones ont des structures très différentes.

Le graphe de la figure 5.12 avec les dimensions réelles des matrices est représenté dans la figure 5.13.

Dans la figure 5.13, l'évaluation du sous-problème $(5, 6)$ dépend uniquement des valeurs déjà calculées des sous-problèmes $(5, 7)$ et $(7, 6)$. Ainsi, les dépendances sont réduites en comparaison à celles du DAG introduit par Godbole. En conséquence, si G_n est partitionnée en sous-graphes (blocs) et ces sous-graphes attribués à des processeurs, ils ne s'échangeront qu'une petite partie des données qu'ils détiendront : celles des nœuds fils

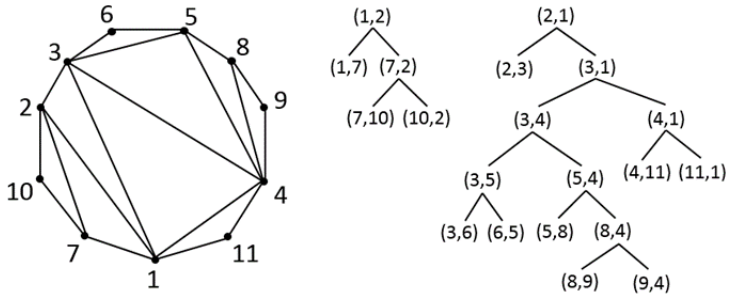


FIGURE 5.12 – Un polygone P de 11 sommets avec le graphe G_{11} correspondant (*les poids des sommets sont représentés par leurs indices*).

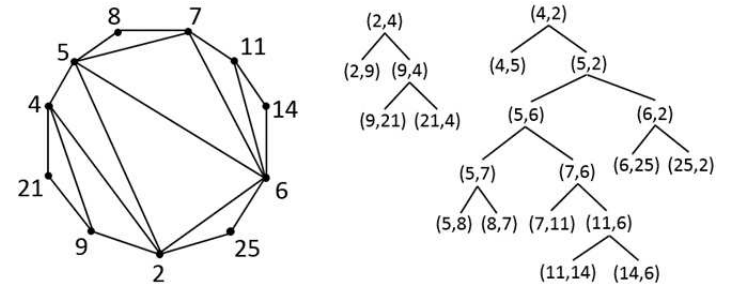


FIGURE 5.13 – Un polygone P de 11 sommets avec le graphe G_{11} correspondant (*les poids des sommets sont représentés par leurs valeurs réelles*).

des nœuds d'autres sous-graphes (nous les appelons aussi *nœuds racine* du sous-graphe).

5.4.2 Algorithme CGM

La complexité temporelle de la solution de Yao est en $O(n^2)$, $O(n)$ pour la phase 1 et $O(n^2)$ pour la phase 2 (voir section 3.4.3). Puisque en pratique ($n >> p$), c'est-à-dire $n^2/p >> n$, nous nous intéressons dans cette thèse uniquement à la parallélisation de la phase 2, source du coût en $O(n^2)$ de la solution de Yao.

L'algorithme BSP/CGM que nous proposons est une succession de trois parties. Sa structure globale est donnée par l'algorithme 20.

Partie 1 : *Construction du graphe G_n*

Dans cette partie, chaque processeur réalise la phase 1 de l'algorithme séquentiel de Yao (c'est-à-dire la génération des nœuds du graphe G_n). A la fin de cette partie, tous les processeurs ont le même DAG-multi-niveaux.

Partie 2 : *Partitionnement du graphe G_n en sous-graphes (blocs)*

Comme la structure du DAG G_n n'est pas connue avant le partitionnement, une conception classique (statique) de l'algorithme BSP/CGM implémentant la phase 2 de

Algorithme 20 : Algorithme BSP/CGM pour les problèmes MCOP et TCP (processeur P_i)

Entrées : Le nombre et la dimension des matrices.
Sorties : Solution optimale du sous-problème évalué par le processeur P_i.

```
1 début
2 |  Partie 1 : Construction du graphe G_n ;
3 |  Partie 2 : Partitionnement du graphe G_n ;
4 |  Partie 3 : Implémentation de la phase 2 de l'algorithme de Yao suivant la
  |  division/mapping choisi à la partie 2.
5 fin
```

l'algorithme séquentiel de Yao est impossible (voir section 5.4.2.1). Ainsi, le partitionnement du graphe G_n doit se faire de manière appropriée pour chaque instance des dimensions des matrices à parenthéser. Il faut donc automatiser le partitionnement du graphe et la distribution des données sur les processeurs.

Pour ce faire, dans la partie 2 de l'algorithme 20, chaque processeur exécute un algorithme que nous désignons *« algorithme de conception dynamique »*. Cet algorithme tire son efficacité du fait qu'*au moins la moitié des nœuds du graphe G_n sont des feuilles*. Il est structuré en deux phases :

1. *La phase de division des tâches* : l'objectif de cette phase est de diviser les calculs comptant pour la phase 2 de l'algorithme de Yao. Pour ce faire, le DAG G_n est divisé en S sous-graphes que nous appelons *bandes*. Ensuite, chaque bande B_i est subdivisée en NB_i sous-graphes que nous appelons *blocs*. Cette division doit garantir que les blocs de chaque bande B_i :
   - soient indépendants les uns des autres, et ainsi ils peuvent être évalués en parallèle ;
   - aient la même charge de calculs (T_{ci}) de sorte à faciliter l'équilibrage de charge pendant la phase de mapping.
2. *La phase de distribution des tâches (mapping)* : dans cette phase, les blocs de chaque bande B_i sont distribués sur les p processeurs, à raison de (NB_i/p) blocs par processeur.

Ces deux phases sont détaillées dans les deux sous-sections suivantes.

Partie 3 : *Implémentation de la phase 2 de l'algorithme séquentiel de Yao suivant la division/mapping choisi à la partie 2.*

Cette implémentation est effectuée en S rondes de calcul, chacune suivie d'une ronde de communication globale. Les calculs des valeurs des nœuds de la bande B_i s'effectuent

à la ronde de calcul i, à raison de (NB_i/p) blocs par processeurs.

Chaque ronde de calcul nécessite $((NB_i/p) \times T_{ci})$ opérations de calcul par processeur.

5.4.2.1 Partitionnement dynamique des tâches

La division dynamique des tâches doit être effectuée avec un minimum de temps, car elle est réalisée séquentiellement sur chaque processeur. Kechid et Myoupo [Kec09] ont proposés un algorithme de conception dynamique qui nécessite $O(n \times p)$ temps d'exécution pour paralléliser un algorithme ayant une complexité temporelle en $O(n^2)$. Bien que $n >> p$, le temps de partitionnement dynamique augmente énormément avec le nombre de processeurs utilisés. Ce qui n'est pas intéressant car l'algorithme doit être extensible à l'accroissement du nombre de processeurs.

Corollaire 5 *Si le temps de partitionnement dynamique des tâches doit être obligatoirement fonction de la taille n du problème, il doit dépendre le moins possible du nombre p de processeurs utilisés. L'idéal est qu'il soit totalement indépendant de p.*

Le modèle de coût BSP qui guide la conception de l'algorithme BSP/CGM de Kechid et Myoupo [Kec09] suppose que chaque bloc induira au plus n messages de diffusion après son évaluation, or rien ne permet de prédire le nombre exact de ces messages. Ceci peut conduire à de fausses prédictions théoriques, et donc, produire un mauvais partitionnement du graphe G_n. De plus, lorsque le même processeur évalue deux blocs dont l'un dépend totalement de l'autre, il n'y a pas de diffusion. Ce détail important n'est pas pris en compte par le modèle de coût BSP utilisé.
Nous proposons un partitionnement dynamique des tâches, qui repose uniquement sur les spécifications du modèle CGM. Ses objectifs sont les suivants :

1. être facile à réaliser ;
2. nécessiter un minimum de temps ;
3. ne faire aucune supposition sur le nombre de messages générés pour un bloc ;
4. utiliser le maximum de processeurs à chaque étape (bande) pour minimiser leur latence ;
5. avoir les blocs (sous-graphes) ayant les plus grandes tailles possibles dans chaque bande, de sorte à diminuer au minimum le nombre des nœuds racines des blocs, et ainsi, minimiser le temps de communication de l'algorithme ;
6. équilibrer les charges entre les processeurs.

Le DAG G_n compte au plus $2n$ nœuds parmi lesquels n feuilles (correspondants aux *segments côtés* du polygone). Comme ces feuilles sont toutes indépendantes, elles peuvent être évaluées en parallèle. Ainsi, elles doivent toute faire partie de la première bande.

Lemme 14 *Plus de la moitié de la charge de calcul est induite par les feuilles du graphe G_n.*

Preuve. Comme la charge de calcul induite par un nœud interne de G_n est plus faible que celle de chacun de ses deux fils, et comme il y a plus de feuilles que de nœuds interne, l'évaluation de chaque nœud interne requiert moins de calculs que celle d'une feuille distincte. ■

À partir du lemme 14, on peut dire que la bande la plus importante à construire est la première bande (la bande B_1). En effet, si elle contient toutes les feuilles du graphe G_n, alors elle induira une charge de calcul supérieure à la moitié de la charge globale. Et si $NB_1 \geq p$, alors chaque processeur évaluera au moins un bloc de la première bande et ainsi, *les p processeurs traiteront plus de la moitié du problème sans communiquer.*

Désignons par T_{cf} la charge de calcul de toutes les feuilles du DAG G_n et T_c la charge globale de calcul du problème ($T_c/2 < T_{cf} < T_c$). Si la charge induite par un bloc de la première bande est T_{cf}/p, on peut être sûr que chaque processeur détiendra au moins un bloc de la première bande.

La minimisation des communications et l'équilibrage des charges constituent deux objectifs contradictoires pour le partitionnement du graphe G_n :

Pour la minimisation les communications, les blocs doivent être les plus larges possibles. De cette façon, le nombre global des nœuds racines est minimisé et les processeurs s'échangent peu d'information. Ainsi, la charge des blocs de la première bande doit être au moins égale à T_{cf}/p. L'idéal pour la première bande est qu'elle contienne au moins p blocs ayant chacun le plus grand nombre de nœuds possibles (c'est-à-dire la plus grande charge possible). Donc, pour minimiser les communications, nous cherchons la charge totale de la plus grande première bande qui contient au moins p blocs.

L'inconvénient d'une telle solution est la possibilité d'un sérieux déséquilibre de charge entre les processeurs s'ils n'ont pas le même nombre de blocs.

Pour un bon équilibrage de charges entre les processeurs, les blocs doivent être de petite taille. Ainsi, si certains processeurs ont un bloc de plus que d'autres, comme les blocs ont une charge relativement faible, la différence de charge sera raisonnable.

L'inconvénient de cette approche est l'augmentation des nœuds racines des sous-graphes (blocs), qui induisent plus de communication entre les processeurs et ainsi, augmentent le temps global de communication.

Dans la suite, nous allons présenter plusieurs solutions parmi lesquelles, une sera axée sur la minimisation des communications, et une autre sera axée sur l'équilibrage des charges entre les processeurs. Les résultats des simulations effectués pour chacune des approches nous permettront de conclure.

Première idée :

Partitionner le DAG G_n avec des blocs ayant une charge T_c/p^2. Ensuite, calculer la charge globale T_g de tous les blocs des bandes ayant au moins p blocs (ou bien considérer le plus grand nombre de blocs k tel que $T_g = k \times (T_c/p^2)$). Construire la première bande avec des blocs ayant la charge $T_{c1} = T_g/p$.

Le problème avec cette première idée est que rien ne garantit qu'avec notre algorithme de construction de blocs (algorithme 22 ou 25 que nous présenterons plus loin dans cette section), la première bande contiendra au moins p blocs. Ceci peut causer un grave problème d'oisiveté des processeurs. Pour rendre cette idée utilisable, il faut un algorithme de construction de bandes qui assure que la première bande sera constituée d'au moins p blocs.

Deuxième idée : *axée sur l'équilibrage des charges entre les processeurs*

Notre principe est le suivant : si (i, k) et (k, j) sont deux feuilles du DAG G_n, alors leur père (i, j) peut faire partie de la première bande. En effet, tous les nœuds internes dont les deux fils sont des feuilles sont tous indépendants et donc peuvent être évalués en parallèle.

Désignons par T_{max} la somme des charges de ces nœuds internes et celle de toutes les feuilles ($T_{max} > T_{cf}$). De cette façon, une meilleure charge (comparé à T_{cf}/p) pour les blocs de la première bande est T_{max}/p.

Exemple : le polygone de la figure 5.14 montre que la charge T_{max} de la première bande est obtenue avec notre deuxième idée à partir des feuilles $\{(1,7),(7,10),(10,2),(2,3),(3,6),(6,5),(5,8),(8,9),(9,4),(4,11),(11,1)\}$ et des nœuds internes $\{(7,2),(3,5),(8,4),(4,1)\}$. Après cette bande B_1, il restera à évaluer les nœuds $\{(1,2),(5,4),(3,4),(3,1),(2,1)\}$. Ainsi, 15 des 20 nœuds de ce graphe appartiennent à la première bande.

Proposition 2 *Avec notre deuxième méthode de partitionnement des tâches, la première bande du graphe G_n est construite en $O(n)$ temps d'exécution.*

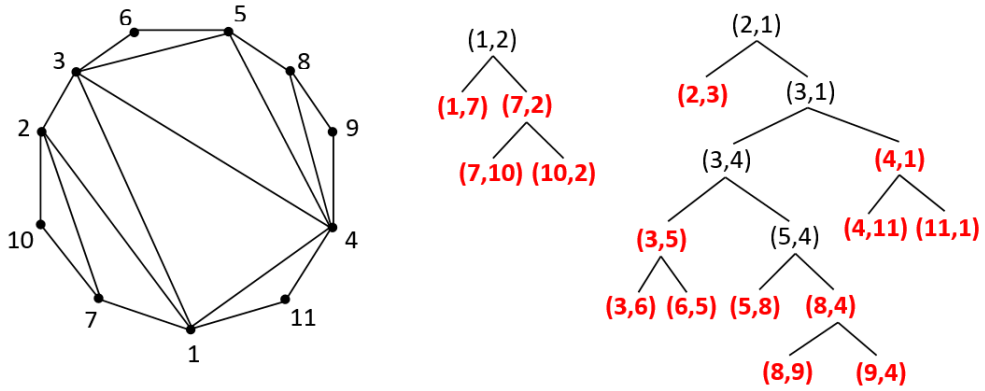


FIGURE 5.14 – Un polygone P de 11 sommets avec le graphe G_{11} correspondant (*les poids des sommets sont représentés par leurs valeurs réelles*).

Preuve. Cette construction requiert deux parcours des nœuds du graphe G_n. Un parcours pour calculer T_{max} et un autre pour construire la bande B_1. ■

Troisième idée : *axée sur la minimisation des communications*

Il est aussi possible de faire varier la charge des blocs de la première bande et observer leur nombre, pour obtenir la plus grande première bande possible.

Formellement, comme avec $T_{c1} = T_c/2p$ on est certain que le nombre de blocs de la première bande est supérieur ou égal à p, on peut faire varier T_{c1} dans l'intervalle $\left[\frac{T_c}{2p}, \ldots, \frac{T_c}{p}\right]$. Si la valeur T_c/p produit moins de p blocs dans B_1, l'on considère la valeur médiane $\frac{1}{2}\left(\frac{T_c}{p} + \frac{T_c}{2p}\right) = \frac{3Tc}{4p}$, et ainsi de suite, utilisant à chaque fois la valeur médiane entre la plus grande bonne valeur connu de T_{c1} (borne inférieure de l'intervalle) et la plus petite mauvaise valeur connue (borne supérieure de l'intervalle) pour optimiser le résultat. Le processus s'arrête lorsque l'intervalle de recherche atteint un certain seuil (par exemple $\frac{T_c}{p^{k+1}}$ où k est un entier positif), et la borne inférieure de l'intervalle de recherche est choisie. Ainsi, nous obtenons la charge des blocs de la plus grande première bande possible correspondant au seuil choisi.

Cette troisième méthode est satisfaisante, mais le temps d'exécution de l'algorithme permettant de la mettre en œuvre est fonction de n et de p.
Néanmoins, la proposition 3 suivante nous montre que cette solution a une complexité temporelle en $O(n \log_2 p)$. Ceci est acceptable car $\log_2 p << p << n$ et ainsi, cette solution est extensible.

Proposition 3 *Avec notre troisième partitionnement dynamique des tâches, la première bande du graphe G_n est construite en $O(n \log_2 p)$ temps d'exécution.*

Preuve. La distance initiale entre les bornes de l'intervalle dans lequel se trouve notre meilleure valeur de T_{c1} est $\frac{T_c}{p} - \frac{T_c}{2p} = \frac{T_c}{2p}$, cette distance est divisée par deux jusqu'à ce qu'elle atteigne le seuil $\frac{T_c}{p^{k+1}}$. Ainsi, le nombre de divisions de cet intervalle est la plus grande valeur de m tel que $\frac{T_c}{2^m p} \geq \frac{T_c}{p^{k+1}}$, c'est-à-dire $p^k \geq 2^m$ ou $k \times \ln p \geq m \ln 2$ et donc $m \leq k \times \frac{\ln p}{\ln 2}$.
Pour chaque valeur de T_{c1}, tous les nœuds du graphe G_n sont parcourus pour calculer la valeur de NB_1 correspondante. De tout ceci, on peut conclure que $O(n \log_2 p)$ calculs sont nécessaires pour déterminer la valeur finale de T_{c1} avec cette troisième idée. ■

Remarque 10 .

1. *Dans cette thèse, nous utilisons par défaut le seuil T_c/p^2, c'est-à-dire $k = 1$;*
2. *Pour gagner d'avantage de temps, ce partitionnement peut être effectué en parallèle sur l'ensemble des processeurs. Pour ce faire, chaque processeur calcule la meilleure valeur de T_{c1} pour une partie de la plage de valeurs possibles et diffuse son résultat à tous les autres. À la fin, chaque processeur garde la meilleure solution obtenu de tous.*

Corollaire 6 *Avec les idées 2 et 3, les charges des processeurs sur la première bande sont équilibrées de manière presque parfaite.*

Note 6 *Dans la suite, nous utiliserons les idées 2 et 3 ainsi que quelques-unes de leurs variantes. Nous les comparerons à partir des résultats expérimentaux que nous effectuerons.*

Il reste à déterminer comment partitionner le reste du graphe G_n, c'est-à-dire le sous-graphe $(G_n - B_1)$ constitué de moins de n nœuds.

Solution 1 : *axée sur l'équilibrage des charges*

Si nous gardons la même charge $T_c/2p$ pour tous les blocs des bandes $\{1, 2, \ldots, S\}$, alors la charge globale de tous les blocs de la première bande sera au moins $T_c/2$ et toutes les bandes $\{2, \ldots, S\}$ seront constituées d'au plus $(T_c/2)/(T_c/2p) = p$ blocs.

Si la charge de ces blocs est plutôt T_{max}/p, comme $T_{max}/p > T_c/2p$, il ne pourra pas avoir plus de p blocs sur les bandes $\{2, \ldots, S\}$. Le choix de partitionner le graphe G_n en blocs ayant une charge de T_{max}/p permet de limiter le nombre de blocs d'un processeur à 2. Ainsi, un processeur évalue soit deux blocs sur la première bande (si $NB_1 > p$) et aucun blocs sur les autres bandes, soit un bloc sur la bande B_1 et éventuellement un bloc sur l'ensemble des bandes $\{2, \ldots, S\}$.

Lemme 15 *Avec le partitionnement du graphe G_n en des blocs ayant une charge de T_{max}/p, le graphe G_n contiendra au plus $2p$ blocs et chaque processeur aura au plus deux blocs à évaluer.*

Preuve. $T_c > T_{max} > (T_c/2)$, ainsi, $(T_{max}/p) \times 2p = 2 \times T_{max} > T_c$. ■

Cette solution satisfait les points 1, 2, 3 et 6 des objectifs cités plus haut, mais peut causer beaucoup de communications (critères 4 et 5).

Lemme 16 *Cette première solution permet de partitionner le graphe G_n en $O(n)$ temps d'exécution.*

Preuve. Elle nécessite juste deux parcours du graphe G_n, un pour calculer T_{max} et un autre pour construire les S bandes. ■

Solution 2 : *axée sur la minimisation des communications*

Cette solution utilise la troisième idée de construction de la première bande. Désignons par T_{int} la charge globale de la première bande construite à partir de cette troisième idée. Les blocs du graphe G_n sont partitionnés en sous-graphes (blocs) ayant une charge T_{int}/p.

Ce partitionnement commence par la construction de la première bande comme susindiqué, et se poursuit par un seul parcours des nœuds du sous-graphe restant (le sous-graphe $G_n - B_1$) pour la construction des autres bandes.

Lemme 17 *Avec le partitionnement du graphe G_n en des blocs ayant une charge de T_{int}/p, le graphe G_n contiendra au plus $2p$ blocs et chaque processeur aura au plus deux blocs à évaluer.*

Preuve. $T_c > T_{int} > (T_c/2)$, ainsi, $(T_{int}/p) \times 2p = 2 \times T_{int} > T_c$. ■

Avec cette seconde solution, les processeurs sont plus oisifs sur les diagonales $\{2, \ldots, S\}$ par rapport à la première. Mais comme $T_{int} \geq T_{max}$ il y aura moins de bandes dans le graphe. L'avantage évident de cette nouvelle solution est qu'elle induit moins de communication que la première.

Lemme 18 *Cette seconde solution permet de partitionner le graphe G_n en $O(n \log_2 p)$ temps d'exécution.*

Preuve. Après la construction de la première bande (proposition 3), un seul parcours du reste du graphe G_n (le sous-graphe $G_n - B_1$) est effectué pour la construction des autres bandes. ■

Construction des bandes et blocs pour la solution 1 :

Algorithme 21 : Partitionnement du DAG G_n en bandes et blocs (charge d'un bloc T_{max}/p)

Entrées : DAG produit par la phase 1 de l'algorithme séquentiel de Yao.
Sorties : le DAG partitionné, c'est-à-dire contenant les bandes et les blocs.

```
1  début
2  |  T_c ← T_max ← 0 ;
3  |  pour chaque paire de nœuds (i,k) et (k,j) de G_n faire
4  |  |  T_c ← T_c + min{i,k} + min{k,j};
5  |  |  si estUneFeuille(i,k) alors T_max ← T_max + min{i,k};
6  |  |  si estUneFeuille(k,j) alors T_max ← T_max + min{k,j};
7  |  |  si estUneFeuille(i,k) et estUneFeuille(k,j) alors
8  |  |  |  T_max ← T_max + min{i,j}
9  |  |  fin
10 |  fin
11 |  T_c1 ← T_max/p ;
12 |  S ← 0 ;
13 |  j ← 1 ;
14 |  DAG_courant ← DAG produit par la phase 1 de l'algorithme de Yao ;
15 |  tant que DAG_courant ≠ ∅ faire
16 |  |  NB_j ← Construire_bande(DAG_courant, T_c1) ;
17 |  |  DAG_courant ← DAG_courant − B_j ;
18 |  |  S ← S + 1;
19 |  |  j ← j + 1;
20 |  fin
21 |  Retourner la division courante ;
22 fin
```

La partie 2 de notre algorithme BSP/CGM (algorithme 20) est effectuée par l'algorithme 21, qui effectue un parcours ascendant du DAG G_n, à partir des feuilles, pour construire les bandes une par une, par des appels successifs de la procédure *Construire_bande* (algorithme 22) qui prend en entrée la paire $(DAG\_courant, T_{cj})$ pour construire une bande. Au premier appel de la procédure *Construire_bande* le *DAG_courant* est le DAG G_n, ensuite, à chaque appel de cette procédure, la bande construite à l'appel précédent est élagué du *DAG_courant* (ligne 17 de l'algorithme 21). La division se termine lorsque le *DAG_courant* devient vide.

Pour construire une bande, la procédure *Construire_bande* fait des appels successifs de la procédure *Construire_bloc* (algorithme 23) qui prend en entrée la paire (DAG', T_{cj}). Celle-ci commence le parcours vers le haut par la feuille la plus à gauche. Elle utilise un compteur de charge C qui est initialisé à zéro, et incrémenté après chaque parcours d'un

nœud (k, l) par la taille de calcul induite par ce nœud (c'est-à-dire $\min\{k, l\}$). La procédure arrête le parcours lorsque C atteint T_{cj}.
Chaque nœud consommé (ajouté à un bloc) est coloré en *noir*. On consomme un nœud uniquement lorsque ses deux fils sont consommés. Ainsi, pendant le processus, si on remonte d'un fils à un père dont le second fils n'est pas noir, ce père n'est pas coloré en noir. Il est plutôt coloré en *gris* et le parcours vers le haut s'arrête. L'algorithme reprend, dans ce cas, le parcours à partir de la feuille non colorée la plus à gauche. S'il n'y en a aucune, alors la bande est achevée ainsi que le bloc courant (les différents compteurs sont alors mis à jours).

Pour garantir l'indépendance entre les blocs, le parcours de chaque bloc doit commencer par la feuille non colorée la plus à gauche du DAG dont le père n'est pas gris. En effet, au début de la construction d'un bloc, une feuille dont le père est gris est certainement une soeur d'une autre feuille colorée noire appartenant à un autre bloc de la même bande.

Nous verons dans la figure 5.16 un aperçu de la construction des bandes B_1 et B_2 pour un graphe G_n quelconque.

Algorithme 22 : Procédure $Construire\_bande(DAG, T_{cj})$, utilisée par l'algorithme 21, pour la construction des bandes du graphe G_n.

début
  $NB_j \leftarrow 0$;
  $DAG' \leftarrow DAG$;
  tant que *non* Fin_bande **faire**
    $bloc\_courrant \leftarrow Construire\_bloc(DAG', T_{cj})$;
    $NB_j \leftarrow NB_j + 1$;
    $DAG' \leftarrow DAG' - \{bloc\_courrant\}$;
  fin
  Retourner NB_j ;
fin

Note 7 *Les algorithmes 22 et 23 sont tirés de [Kec09].*

Construction des bandes et blocs pour la solution 2 :

L'algorithme 24 permet de construire les bandes et blocs avec notre solution 2. Comme la procédure de construction de la première bande est différente des autres, l'algorithme 24 fait un premier appel à la procédure $Construire\_premiere\_bande$ (algorithme 25) qui prend en entrée le couple $(DAG\_courant, T_c)$, ensuite recherche la charge de la meilleure

Algorithme 23 : Procédure $Construire\_bloc(DAG, T_{cj})$, utilisée par l'algorithme 22.

```
début
    nœud_courant ← la feuille non colorée la plus à gauche du DAG et dont le
    père n'est pas gris ou Ø au cas où une telle feuille n'existe pas ;
    si nœud_courant = Ø alors
        fin_bloc ← vrai ;
        fin_bande ← vrai;
    sinon
        C ← 0 ;
        tant que non fin_bloc faire
            colorer_noir(nœud_courant);
            C ← C+charge(nœud_courant);
            si (C ≥ T_cj) ou (est_racine(nœud_courant)) alors
                fin_bloc ← vrai ;
            sinon
                si (couleur(père(nœud_courant)) = gris) alors
                    nœud_courant ← père(nœud_courant);
                sinon
                    colorer_gris(père(nœud_courant)) ;
                    nœud_courant ← la feuille non colorée la plus à gauche du DAG
                    et dont le père n'est pas gris ou Ø au cas où une telle feuille
                    n'existe pas ;
                    si nœud_courant = Ø alors
                        fin_bloc ← vrai ;
                        fin_bande ← vrai;
                    fin
                fin
            fin
        fin
    fin
fin
```

première bande, la construit et retourne le nombre NB_1 de blocs de cette bande ainsi que leur charge T_{c1}. L'algorithme 24 appelle la procédure *Construire_bande* (algorithme 22) pour toutes les bandes suivantes (les bandes $\{2, \ldots, S\}$). Pendant la construction des bandes (avec les algorithmes 22 et 25) les blocs sont construits à l'aide de l'algorithme 23.

Algorithme 24 : Partitionnement du DAG G_n en bandes et blocs avec la solution 2 (charge d'un bloc T_{int}/p)

Entrées : DAG produit par la phase 1 de l'algorithme séquentiel de Yao.
Sorties : le DAG partitionné, c'est-à-dire contenant les bandes et les blocs.

```
1  début
2  |  T_c ← ∑ min{i, j} ;
3  |  DAG_courant ← DAG produit par la phase 1 de l'algorithme de Yao ;
4  |  (NB_1, T_b1) ← Construire_premiere_bande(DAG_courant, T_c);
5  |  DAG_courant ← DAG_courant − {B_1} ;
6  |  S ← 1 ;
7  |  j ← 2 ;
8  |  tant que DAG_courant ≠ ∅ faire
9  |  |  NB_j ← Construire_bande(DAG_courant, T_b1) ;
10 |  |  DAG_courant ← DAG_courant − B_j ;
11 |  |  S ← S + 1;
12 |  |  j ← j + 1;
13 |  fin
14 |  Retourner le DAG partitionné ;
15 fin
```

Lemme 19 *Avec nos algorithmes de partitionnement dynamique, le graphe G_n est partitionné en moins de p bandes (i.e. $S < p$).*

Preuve. Nos approches de partitionnement dynamique reposent sur la charge de calcul des feuilles du graphe. Le pire des cas survient lorsque les feuilles du graphe G_n nécessitent le moins de calculs, et qu'en même temps, les nœuds internes constituent une chaîne. Ceci est illustré dans la figure 5.15a pour un problème d'ordre $n = 11$.

Pour un problème d'ordre n, dans le pire des cas, la première bande a une charge de calcul de l'ordre de $\frac{n(n-1)}{2}$, et ainsi, un bloc a une charge de l'odre de $\frac{n(n-1)}{2p}$. Les nœuds internes du graphe ont une charge de $\frac{(n-2)(n-1)}{2}$ et cette charge n'est pas suffisante pour constituer p blocs.

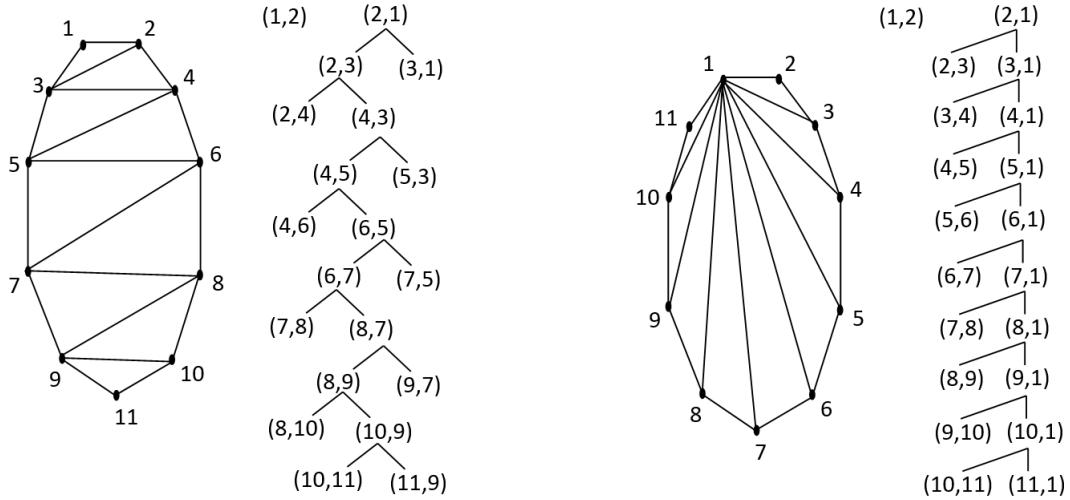
Dans le meilleur des cas, la charge des feuilles est maximale (Ceci est illustré dans la figure 5.15b pour $n = 11$). Ainsi, la charge des blocs est de l'ordre de $\frac{n(n-1)}{2p}$ et celle de

Algorithme 25 : Procédure $Construire\_premiere\_bande(DAG, T_c)$, utilisée par l'algorithme 24, pour la construction de la première bande du DAG G_n.

Entrées : DAG produit par la phase 1 de l'algorithme séquentiel de Yao.
Sorties : le nombre et la charge des blocs de la première bande.

```
1  début
2      sup ← Tc/p ;
3      inf ← Tc/2p ;
4      gap ← ⌈sup − inf⌉;
5      Tcj ← sup ;
6      tant que gap ≥ Tc/p² faire
7          NB1 ← 0 ;
8          DAG′ ← DAG ;
9          tant que non fin_bande faire
10             bloc_courant ← Construire_bloc(DAG′, Tcj) ;
11             NB1 ← NB1 + 1;
12             DAG′ ← DAG′ − {bloc_courant} ;
13         fin
14         si NB1 ≥ p alors
15             inf ← Tcj ;
16         sinon
17             sup ← Tcj;
18         fin
19         gap ← ⌈sup − inf⌉ ;
20         Tcj ← ⌈(sup+inf)/2⌉;
21     fin
22     Retourner (NB1, inf) ;
23 fin
```

tous les nœuds internes est de l'ordre de $(n-2)$. Comme $\frac{n}{p} > 1$, alors, $\frac{n(n-1)}{2p} > (n-2)$ et tous les nœuds internes constitueront un seul bloc. Dans ce cas, le graphe sera partitionné en 2 bandes exactement. ■



(a) Graphe G_n : Pire des cas

(b) Graphe G_n : Meilleur des cas

FIGURE 5.15 – Partitionnement dynamique du graphe G_n : le pire et le meilleur des cas.

5.4.2.2 Dépendance des données

Comme l'évaluation d'un nœud du DAG G_n ne dépend que des valeurs déjà calculées de ses deux fils, les valeurs communiquées par les processeurs sont celles des nœuds racines des blocs qu'ils détiennent, et ces nœuds ne sont pas nombreux. De plus, ces informations sont utilisées uniquement par les processeurs détenant les nœuds père des nœuds racines des blocs.
En d'autres termes, les valeurs des nœuds racine des blocs d'une bande B_j sont utilisées (donc communiquées) uniquement par les processeurs des blocs de la bande B_{j+1} qui en ont besoin.

Avec de telles dépendances entre les blocs du graphe G_n, quelque soit le partitionnement utilisé (parmi nos deux solutions), le temps de communication sera très faible.

Corollaire 7 *À cause du schéma de dépendances entre les blocs du graphe G_n, peu d'informations sont communiquées entre les processeurs. Ainsi, contrairement à tous les algorithmes BSP/CGM présentés jusqu'ici, dans celui-ci, le temps des calculs locaux constitue l'essentiel du temps global d'exécution.*

5.4.2.3 Distribution des blocs (mapping)

Une fois que le DAG G_n est partitionné en bandes et blocs, la tâche qui reste à effectuer est la distribution des différents blocs sur les processeurs de sorte que leurs charges soient équilibrées. Les NB_j blocs de la bande j sont distribués sur les p processeurs à raison de $\lceil NB_j/p \rceil$ blocs par processeur.

Si NB_j est inférieur à p, alors $(p - NB_j)$ processeurs seront oisifs lors de l'évaluation de la bande j. Pour réduire globalement cette oisiveté, nous distribuons les blocs sur les processeurs de manière cyclique. C'est-à-dire suivant la politique $FIFO$ *(First In First Out)*[8]. Ainsi, parmi les processeurs oisifs, celui qui est choisi est celui qui est passé en premier dans cet état.

Cette distribution peut être effectuée tel que pour chaque bande B_j, les $\lceil NB_j/p \rceil$ blocs d'un processeur P_i soient consécutifs. Mais cette caractéristique ne permet pas aux processeurs des blocs de la bande B_{j+1} de commencer et terminer l'évaluation de leurs blocs aussitôt que possible. En fait, avec nos algorithmes de partitionnement dynamique, $NB_1 \geq p$. Ainsi, les p premiers blocs de la bande B_1 sont attribués aux p processeurs et le reste des $(NB_1 - p)$ blocs sont attribués aux processeurs $1, \ldots, (NB_1 - p)$. Les blocs de la bande B_2 sont attribués aux processeurs en commençant par le processeur $(NB_1 - p + 1)$. Il en est de même jusqu'à la dernière bande S.
Après l'évaluation des p premiers blocs de la bande B_1, les valeurs des nœuds racines sont communiquées aux processeurs des blocs de la bande B_2[9], qui peuvent aussitôt commencer l'évaluation de leurs blocs. Parallèlement, les $(NB_1 - p)$ processeurs ayant un bloc supplémentaire sur la bande B_1 peuvent évaluer leur second bloc. Ainsi, comme les p premiers blocs de la bande B_1 déjà évalués à ce niveau son contiguës, tous les blocs de la bande B_2 desquels dépendent ces p blocs peuvent être évalués entièrement. Il en est de même pour les diagonales suivantes.

Nous pouvons donc voir une étape de notre algorithme BSP/CGM comme l'évaluation d'au plus p blocs contiguës (appartenant à la même bande ou non) et tous indépendants.

Si par contre les blocs d'un processeur P_i situés sur la même diagonale sont contiguës, alors, après que chaque processeur ait évalué son premier bloc de la bande B_1, les processeurs de la bande B_2 pourront commencer l'évaluation de leurs blocs sans pouvoir terminer, à cause des valeurs des nœuds racines non disponibles des blocs de la bande B_1.

8. Cette distriution de données est celle utilisée dans tous les algorithmes parallèles proposés dans cette thèse.

9. Ces processeurs de la bande B_2 n'ont qu'un seul bloc sur la bande B_1 car un processeur évalue au plus deux blocs du graphe.

La figure 5.16 illustre le partitionnement du graphe G_n et la distribution des blocs des bandes B_1 et B_2 sur 4 processeurs. Les nœuds colorés en *noir* sont ceux de la première bande, ceux colorés en *gris* sont ceux de la seconde bande et les nœuds non colorés sont ceux des autres bandes.

L'évaluation de ces bandes sera effectuée comme suit : à l'étape 1, les 4 premiers blocs de la bande B_1 seront évalués à raison d'un bloc par processeur. À l'étape 2, le dernier bloc de la bande B_1 sera évalué par le processeur P_1 pendant que les trois premiers blocs de la bande B_2 seront évalués par les processeurs P_2, P_3 et P_4. Le processus se poursuivra de cette façon jusqu'à la bande S.

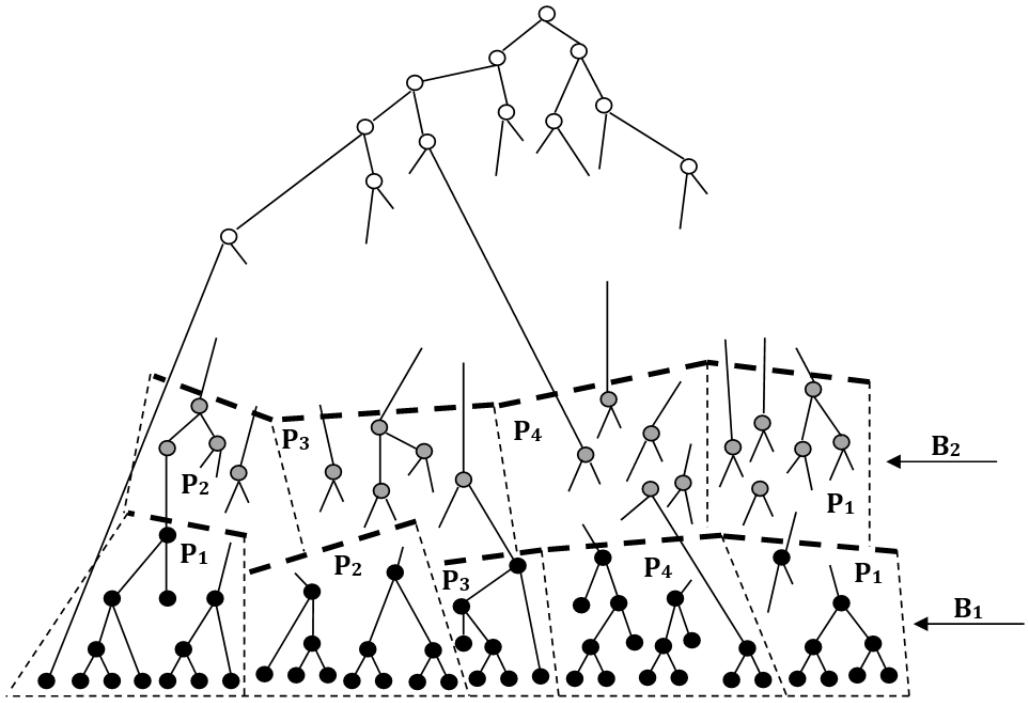


FIGURE 5.16 – Partitionnement du graphe G_n et distribution des blocs des bandes B_1 et B_2 sur 4 processeurs.

5.4.2.4 Complexité

Théorème 12 *Dans le pire des cas, notre algorithme BSP/CGM avec la solution 1 du partitionnement dynamique des tâches, résous un problème MCOP d'ordre n avec S rondes de communication tel que $S < p$ et $O(n^2/p)$ calculs locaux sur chaque processeur.*

Preuve. D'après le lemme 19, notre algorithme compte au plus $S < p$ bandes, c'est-à-dire $S < p$ rondes de communication.

La partie 1 de l'algorithme 20 requiert $O(n)$ temps de calcul.
D'après le lemme 16, la partie 2 requiert $O(n)$ calculs locaux.
Pour la partie 3, supposons que chaque nœud du graphe G_n s'évalue avec le maximum de temps $O(n)$. Ainsi, Chaque bloc du graphe aura au plus $\frac{2n}{p}$ nœuds, et comme la charge de chaque nœud est maximale, le temps de calcul d'un bloc sera en $O(n^2/p)$. Puisque chaque processeur évalue au plus 2 blocs du graphe, il effectue au plus $O(n^2/p)$ calculs locaux. ■

Théorème 13 *Dans le pire des cas, notre algorithme BSP/CGM avec la solution 2 du partitionnement dynamique des tâches, résous un problème MCOP d'ordre n avec S rondes de communication tel que $S < p$ et $O(n^2/p)$ calculs locaux sur chaque processeur.*

Preuve. Ceci s'obtient à partir des lemmes 17 et 18 et d'un raisonnement similaire à celui du théorème 12. ■

Théorème 14 *Avec nos deux solutions de partitionnement dynamique de tâches, le temps d'exécution de notre algorithme CGM est en $O\left(\frac{n^2}{p} \times S\right)$ avec $2 \leq S < p$.*

Preuve. Ceci se déduit des théorèmes 12 et 13. ■

5.4.3 Résultats des simulations

5.4.3.1 Résultats obtenus pour la solution 1 (charge des blocs : T_{max}/p)

Dans cette sous-section, nous présentons les résultats des simulations pour notre algorithme BSP/CGM dans lequel les blocs ont une charge de T_{max}/p. Nous désignons cette solution *MCOP_ Yao_ Tmax-p*.

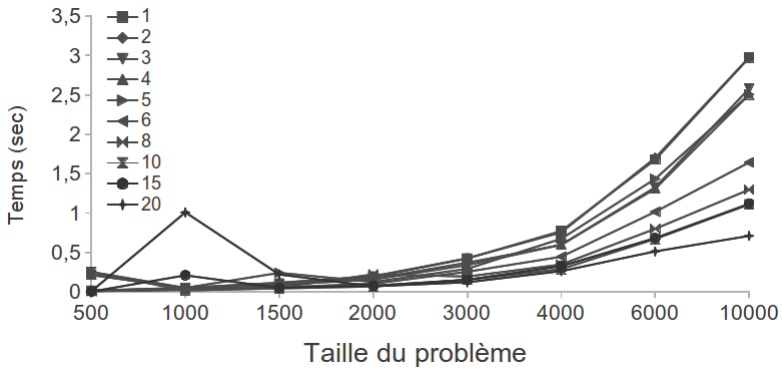


FIGURE 5.17 – Temps total d'exécution pour l'algorithme *MCOP_ Yao_ Tmax-p* avec $p \in \{1, 2, 3, 4, 5, 6, 8, 10, 15, 20\}$ et $n \in \{500,\ 1000, 1500,\ 2000, 3000, 4000,\ 6000,\ 10000\}$.

La figure 5.17 montre que le temps total d'exécution diminue avec le nombre de processeurs utilisés. De même, ce temps n'augmente pas drastiquement lorsque la taille du problème augmente (par exemple, lorsqu'on passe d'un problème d'ordre 6000 à un problème d'ordre 10000, cette augmentation est de $65,333\%$). Ainsi, on peut dire que notre algorithme est extensible à l'accroissement de la taille des données et du nombre de processeurs.

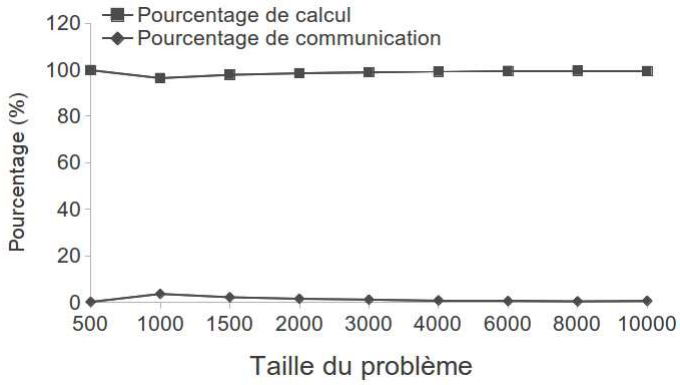


FIGURE 5.18 – Pourcentage de calcul vs pourcentage de communication pour $p = 10$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

La figure 5.18 montre comme nous l'avions annoncé dans le corollaire 7, que le temps de calcul constitue l'essentiel du temps d'exécution de cet algorithme.

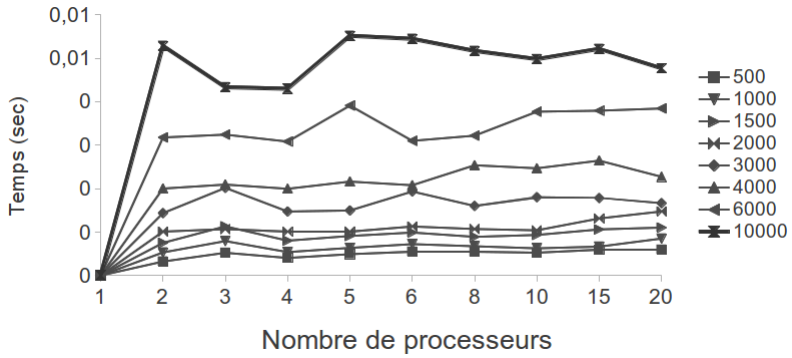


FIGURE 5.19 – Temps global de communication en fonction de p pour l'algorithme *MCOP_Yao_Tmax-p* avec $p \in \{1, 2, 3, 4, 5, 6, 8, 10, 15, 20\}$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

La figure 5.19 montre que le temps de communication augmente avec la taille du problème. Mais, pour le même problème, lorsque le nombre de processeurs augmente, ce temps évolue en dent de scie et semble décroitre globalement. Ceci est dû à la variation

imprévisible du nombre de nœuds racines des blocs du graphe lorsqu'on change leur structure (c'est-à-dire les nœuds qui constituent les blocs et leur nombre). Néanmoins, ce temps est très faible quel que soit le nombre de processeurs utilisés.

Corollaire 8 *À partir des figures 5.18 et 5.19, on déduit que le réel challenge lors de la conception d'un algorithme BSP/CGM basé sur l'accélération de Yao est l'équilibrage de charge des processeurs, car ceci permet de diminuer le temps global de calcul, et ainsi, diminuer le temps d'exécution de l'algorithme.*

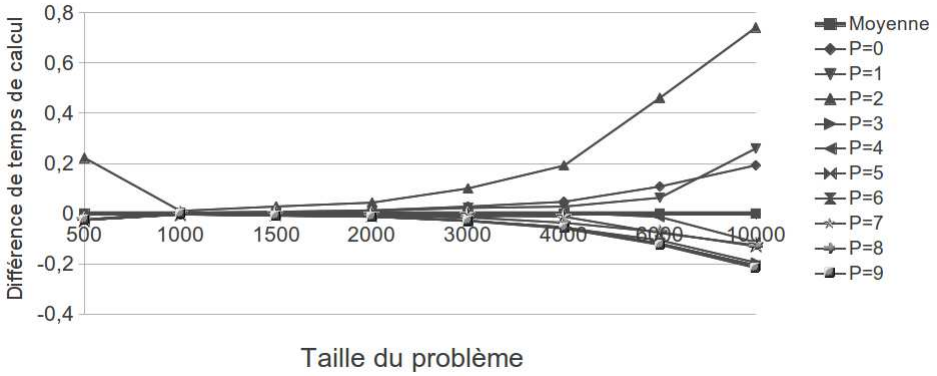


FIGURE 5.20 – Différence de charge relative à la charge moyenne pour $p = 10$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

La figure 5.20 montre un bon équilibrage de charge entre les processeurs. À l'exception du processeur P_2, tous les processeurs ont une charge très proche de la charge moyenne. P_2 est le processeur qui évalue le dernier bloc du graphe G_n, et donc effectue beaucoup plus de calculs que les autres.

Les processeurs P_0, P_1 et P_2 sont ceux qui évaluent deux blocs de G_n. Il est donc normal qu'ils effectuent plus de calculs que les autres. Pour que leurs charges soient plus proches de celles des autres processeurs, on peut partitionner le graphe G_n en des blocs de plus petite charge. Ainsi, il y aura plus de blocs dans le graphe et les processeurs seront mis d'avantage à participation. De plus, les derniers blocs du graphe (ceux des bandes supérieures) induiront un faible déséquilibre de charge entre les processeurs.

5.4.3.2 Comparaison de la solution 1 (charge des blocs : T_{max}/p) et la solution 2 (charge des blocs : T_{int}/p) de partitionnement du graphe G_n

Dans cette section, nous présentons le temps total d'exécution pour un ensemble de variantes de notre algorithme. Ces tests sont effectués sur 10 processeurs pour les algorithmes suivants :

1. *MCOP_ Yao_ Tmax-p* : l'algorithme de la section précédente (solution 1) ;
2. *MCOP_ Yao_ int* : notre algorithme avec les blocs ayant une charge T_{int}/p (solution 2) ;
3. *MCOP_ Yao_ int_ sort* : l'algorithme *MCOP_ Yao_ int* où nous avons appliqué le prétraitement supplémentaire que nous avons proposé pour l'algorithme séquentiel de Yao (*le tri des poids des sommets du polygone à trianguler*. Voir section 3.4.4 du chapitre 3) ;
4. *MCOP_ Yao_ int_ comm* : l'algorithme *MCOP_ Yao_ int* où les résultats partiels de chaque nœud racine sont communiqués juste après l'évaluation du nœud.

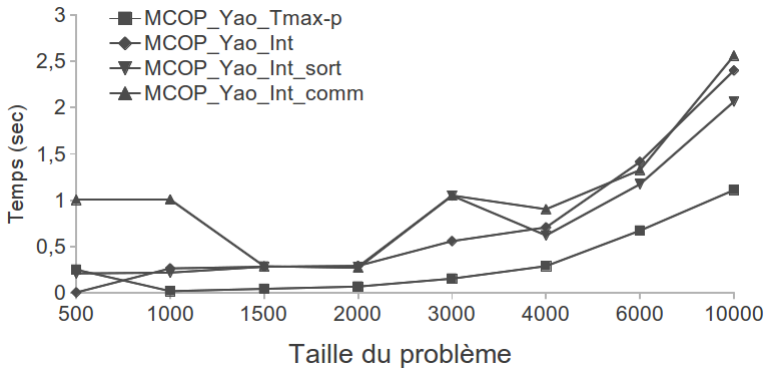


FIGURE 5.21 – Temps total d'exécution pour quelques variantes de notre algorithme avec $p = 10$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

La figure 5.21 montre que l'algorithme *MCOP_ Yao_ int_ sort* a de meilleures performances que l'algorithme *MCOP_ Yao_ int*. D'où, notre prétraitement additionnel ne nuit pas à la parallélisation. Il permet d'optimiser le temps de calcul sur chaque processeur, et donc, d'améliorer l'équilibrage des charges entre eux. Il améliore ainsi le temps d'exécution global de l'algorithme BSP/CGM.

Cette figure montre aussi que la communication des résultats partiels d'un nœud racine juste après l'évaluation du nœud ne rend pas l'algorithme plus efficace. Ceci est dû à l'accroissement de l'effet du start-up sur l'envoie des différents (petit) messages porteurs des informations des différents nœuds.

Comme nous le prévoyions, les solutions axées sur la minimisation des communications sont moins performantes. Les raisons sont les suivantes :

1. Le temps de partitionnement dynamique du graphe G_n est plus grand que celui des solutions axées sur l'équilibrage de charge ;

2. Lors des tests effectués sur les variantes de l'algorithme *MCOP_ Yao_ int*, le graphe G_n était partitionné en $p+1$ blocs. C'est-à-dire 2 bandes, constituées de p blocs dans la première et d'un seul bloc dans la seconde. Ainsi, le processeur P_1 qui évalue le seul bloc de la bande B_2, effectue énormément plus de calculs que les autres, et pendant ce temps, tous ces autres processeurs sont oisifs.

Tout ceci confirme ce que nous affirmons dans le corollaire 8. En effet, à ce stade, notre meilleur algorithme (c'est-à-dire le meilleur partitionnement du graphe G_n) est *MCOP_ Yao_ Tmax-p*.

Pour mieux équilibrer les charges entre les processeurs, nous nous proposons de diminuer la charge de calcul des blocs (c'est-à-dire leur taille) et observer le résultat obtenu.

5.4.3.3 Résultats obtenus avec des blocs de petites tailles (charge des blocs : $T_c/8p$)

Dans cette section nous comparons les résultats obtenus de notre meilleur algorithme (*MCOP_ Yao_ Tmax-p*) à ses variantes suivantes :

1. *MCOP_ Yao_ 8P* : Notre algorithme avec des blocs ayants une charge $T_c/(8p)$;
2. *MCOP_ Yao_ Tmax-p_ sort* : l'algorithme *MCOP_ Yao_ Tmax-p* où nous avons appliqués notre prétraitement supplémentaire sur l'algorithme séquentiel de Yao.

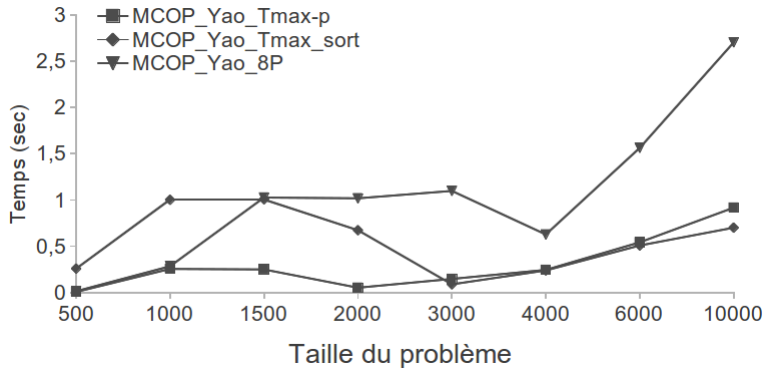


FIGURE 5.22 – Temps total d'exécution pour quelques variantes de notre algorithme avec $p = 10$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

MCOP_ Yao_ 8P est un algorithme BSP/CGM optimal dans lequel chaque processeur a au plus 8 blocs à évaluer.

La figure 5.22 nous montre que la diminution de la taille des blocs du graphe jusqu'à un certain seuil produit de mauvaises performances, mais avec un meilleur équilibrage des

charges. Ceci peut provenir de l'augmentation du nombre des nœuds racines des blocs, qui a conduit à l'augmentation du temps de communication au point de ternir l'équilibrage des charges. Nous constatons aussi qu'avec notre prétraitement supplémentaire, on obtient de meilleures performances. Finalement, notre meilleur algorithme pour la parallélisation de l'algorithme séquentiel de Yao est l'algorithme *MCOP_ Yao_ Tmax-p*. Il est plus performant avec notre prétraitement supplémentaire.

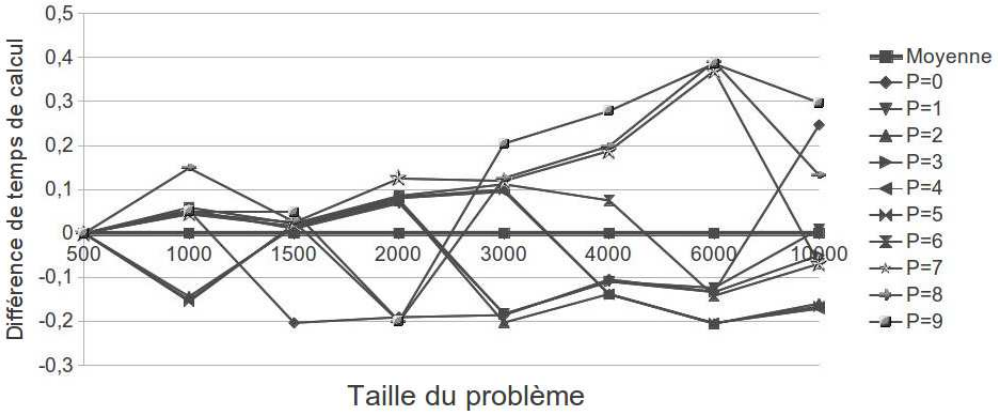


FIGURE 5.23 – Différence de charge relative à la charge moyenne pour l'algorithme *MCOP_ Yao_ 8P* avec $p = 10$ et $n \in \{500, 1000, 1500, 2000, 3000, 4000, 6000, 10000\}$.

La figure 5.23 montre que l'algorithme *MCOP_ Yao_ 8P* équilibre mieux les charges entre les processeurs que l'algorithme *MCOP_ Yao_ Tmax-p*.

5.4.4 Discussion (Résultat 6)

Les tests que nous avons effectués nous permettent de présenter les résultats suivants :

1. Le partitionnement dynamique du graphe G_n doit se faire en un minimum de temps. Il ne peut être fait avec moins de $O(n)$ calculs locaux, car il nécessite un parcours de tous les nœuds du graphe. Si ce temps excède $O(n)$, alors il se rapproche de la complexité $O(n^2)$ de l'algorithme à paralléliser, et peut être source de mauvaise performances. Néanmoins, si le partitionnement peut être effectué en parallèle, la dégradation de performances peut être réduite ;
2. Le principal challenge lors de la conception d'algorithmes BSP/CGM basés sur l'algorithme séquentiel de Yao est l'équilibrage des charges de calcul entre les processeurs ;
3. Si les blocs du graphe ont des charges élevées, alors les charges des processeurs ne seront pas déséquilibrées et l'algorithme parallèle obtenu sera peu performant ;

4. Si la charge des blocs du graphe est très faible, alors les processeurs auront des charges très équilibrés. Mais l'augmentation des communications pourra ternir cet équilibre des charges et conduire à un algorithme peu performant ;
5. L'idéal est d'utiliser d'une part un partitionnement dynamique du graphe G_n qui nécessite au plus $O(n)$ calculs séquentiels, et d'autre part, utiliser des blocs induisant une charge qui optimise l'équilibrage de charge entre les processeurs sans engendrer trop de communication. Notre solution qui remplit ces conditions est l'algorithme *MCOP_ Yao_ Tmax-p* dans lequel les blocs ont une charge de T_{max}/p ;
6. Nous avons introduit un prétraitement additionnel à l'algorithme séquentiel de Yao et l'avons combiné avec notre meilleur partitionnement dynamique des tâches pour obtenir notre meilleure solution : l'algorithme *MCOP_ Yao_ Tmax-p_ sort*.

5.5 Synthèse

Dans ce chapitre, nous avons proposés deux algorithmes BSP/CGM pour la résolution de trois des problèmes de la classe étudiée. Lors des calculs locaux, ces algorithmes BSP/CGM utilisent des algorithmes séquentiels dit accélérés.

Le premier algorithme BSP/CGM résoud un problème OBST d'ordre n en $O(n^2/p)$ temps de calcul avec $\lceil (2p)^{1/2} \rceil$ rondes de communication sur p processeurs. l'avantage de cet algorithme est qu'il est basé sur l'algorithme séquentiel de Knuth dont la complexité temporelle est en $O(n^2)$. Avec notre division des tâches et notre distribution des données, cet algorithme BSP/CGM équilibre bien les charges des processeurs.

Nous avons constaté que dans cet algorithme, les processeurs effectuent des communications inutiles qui augmentent leur temps de latence. Ainsi, nous avons proposés un protocole qui permet aux processeurs à l'aide des *messages d'informations* qu'ils s'envoient au fil des étapes de l'algorithme, d'éviter l'envoie des informations inutiles à d'autres processeurs, et de commencer et terminer l'évaluation de leur blocs aussitôt que les données dont ils ont besoins sont disponibles. Ce protocole permet d'évaluer les blocs de plusieurs diagonales en même temps, celles dont les données nécessaires à leur évaluation sont disponibles au même moment. Ces deux améliorations ont permis de diminuer de manière significative, les temps globaux de communication et de calculs locaux de notre algorithme BSP/CGM basé sur l'accélération de Knuth.

Le second algorithme BSP/CGM que nous avons présentés permet d'obtenir des résultats encore meilleurs. En effet, à cause de la structure du graphe de tâches utilisé et la

nature des calculs à effectuer, il y a peu de dépendances entre les nœuds du graphe. L'évaluation d'un nœud de ce graphe (union de deux arbres) dépend uniquement des valeurs déjà calculées correspondant à ses deux fils. Cette caractéristique fait que le temps de communication qui constitue le plus souvent l'essentiel du temps d'exécution des algorithmes parallèles BSP/CGM, est très faible avec celui-ci quelque soit le schéma de communication des processeurs. Ici, l'essentiel du temps d'exécution est constitué du temps des calculs locaux. Ainsi, les efforts lors de la conception d'algorithmes BSP/CGM basés sur l'accélération de Yao doivent être axés essentiellement sur l'équilibrage des charges entre les processeurs.

Nous avons appliqués le prétraitement supplémentaire que nous avons proposé à l'algorithme séquentiel de Yao à notre algorithme BSP/CGM. Le résultat a été satisfaisant. Il a permis d'améliorer le temps des calculs locaux, et ainsi, d'améliorer le temps global d'exécution de l'algorithme.

Conclusions et perspectives

Sommaire

Nous rappelons dans un premier temps la problématique abordée dans ce travail de thèse et les choix méthodologiques que nous avons opérés pour développer nos solutions. Nous indiquons ensuite les résultats obtenus, en mettant l'accent sur les limitations de ce travail, les critiques qu'il convient d'en faire et ce qu'ils ouvrent comme perspectives.

Problématique abordée et choix méthodologiques

Nous nous sommes intéressé dans cette thèse à la conception d'algorithmes parallèles suivant le modèle de calcul parallèle BSP/CGM (*Bulk Synchronous Parallel / Coarse-Grained Multicomputer*), pour la résolution d'une classe de problèmes de programmation dynamique de type *polyadique non-serial*. Notre travail s'est déroulé en plusieurs parties. Il a débuté (chapitres 1 et 2) par l'étude de l'évolution des machines parallèles et des modèles de calcul parallèle. Ensuite, dans la seconde partie (chapitres 3 et 4), nous avons étudiés les spécificités de la classe de problèmes abordée ainsi que quelques solutions (algorithmes) séquentielles existantes. Nous avons ainsi proposés un algorithme parallèle générique pour l'ensemble des problèmes de la classe. Dans la troisième partie (chapitre 5), nous avons proposé des solutions accélérées spécifiques à certains problèmes de la classe.

Dans la première partie, l'étude de l'évolution des machines parallèles nous a montré que plusieurs facteurs technologiques, logicielles et économiques amènent les concepteurs d'architectures parallèles à converger vers des systèmes constitués d'un ensemble d'ordinateurs complets reliés par un réseau de communication. Cette convergence suggère de modéliser le nombre de processeurs, leurs performances et les paramètres numériques des

communications, plutôt que la topologie du réseau d'interconnexion ou le routage des messages, qui sont des facteurs trop variables. Ces considérations sont celles qui ont guidés la description d'une machine abstraite capable de généraliser la majorité des machines parallèle existante : *la machine abstraite BSP*.
Le modèle CGM est une variante complètement gros-grain du modèle BSP. Dans ce modèle, les performances des algorithmes sont évaluées à travers le temps global de calcul et le nombre de rondes de communication entre les processeurs. Les algorithmes conçus via ce modèle sont portables et mappent bien sur les clusters de station de travail. Ainsi, c'est naturellement que nous l'avons adopté pour la conception de nos algorithmes parallèles.

Analyse critique des résultats obtenus

Dans la deuxième partie de notre travail, nous avons proposé une solution parallèle générique pour la classe de problèmes de programmation dynamique étudiée. Cette classe regroupe un ensemble de problèmes très utilisés (le problème d'ordonnancement de produit de chaîne de matrices, le problème de recherche de l'arbre binaire de recherche optimal, le problème de triangulation de polygone convexe, etc.). Deux propriétés du graphe des tâches correspondant aux problèmes de cette classe compliquent leur parallélisation via le modèle BSP/CGM :

- La forte dépendance entre les nœuds (sous-problèmes) des niveaux différents qui complique la division des calculs et leur affectation aux processeurs ainsi que la définition d'un schéma de communication ;
- L'inégalité des charges de calcul des différents niveaux du graphe qui complique l'équilibrage de charge entre les processeurs.

Ces deux contraintes correspondent chacune à l'un des principaux goulots d'étranglements des performances des algorithmes BSP/CGM (minimisation des communications et équilibrage de charges), et notre principale tâche était de réduire leurs effets sur l'efficacité de notre solution parallèle. Nous sommes partis d'un algorithme BSP/CGM existant, celui proposé par Kechid et Myoupo [Kec09] pour la résolution de tous les problèmes de la classe abordée. Il nécessite $O(n^3/p)$ temps de calcul sur chaque processeur avec $O(p)$ rondes de communication. Il a été conçu en donnant la priorité à la réduction des communications par rapport à la réduction du déséquilibre de charges. Nous lui avons permis de se rapprocher des deux objectifs en le dotant d'un mécanisme de rééquilibrage de charges. Ainsi nous avons proposé une méthode d'allocation des blocs aux processeurs qui s'effectue de proche en proche. Elle est guidée par l'estimation de la charge de calcul induite par

l'ensemble des blocs déjà alloués à chaque processeur, et l'emplacement de ces blocs dans le graphe. Cette méthode se décline suivant deux scénarios : *une distribution ascendante* (partant des blocs des nœuds de niveaux inférieurs vers ceux des niveaux supérieurs) facile à réaliser, mais moins efficace, et *une distribution descendante* plus complexe, mais avec de meilleures performances. Cette distribution des données réduit considérablement l'oisiveté des processeurs, car elle leur permet de rester actif plus longtemps. Elle équilibre beaucoup mieux leurs charges par rapport à la méthode initiale (la Distribution par Projection Bidirectionnelle Alternative), mais induit plus de communication.

Nous avons par la suite proposé un algorithme BSP/CGM générique qui a le même temps de calcul que le précédent (c'est-à-dire en $O(n^3/p)$), mais fait passer le nombre de rondes de communications à $\lceil (2p)^{1/2} \rceil$. Il est donc meilleur que le précédent qui en compte $O(p)$. Dans cet algorithme, le graphe des tâches est partitionné tel que chaque processeur évalue au plus 2 blocs (sous-graphes). La distribution des données sur les processeurs est proche de notre distribution ascendante. Ainsi, avec ce nouvel algorithme, les processeurs sont peu oisifs et leurs charges sont bien équilibrées. Comme le nombre de rondes de communication et le nombre de messages communiqués dans chaque ronde ne dépendent pas du nombre de données en entrée du problème, notre algorithme est extensible à l'accroissement de la taille du problème. Les résultats de l'implémentation de cet algorithme confirment son efficacité et la sauvegarde de ses performances à l'accroissement de n et de p. L'inconvénient majeur de cet algorithme est qu'il est basé sur la solution séquentielle générique dont la complexité temporelle est en $O(n^3)$, pourtant des solution plus efficaces existent pour certains problèmes de la classe abordée.

Nous nous sommes intéressés dans la troisième partie de notre travail à la conception de solutions BSP/CGM accélérées pour les problèmes MCOP, TCP et OBST. Pour ce faire, nous nous sommes basés sur deux algorithmes séquentiels plus rapides que la version générique. Ces solutions sont spécifiques à des problèmes précis de la classe étudiée. Il s'agit de l'algorithme de Knuth pour le problème OBST et l'algorithme de Yao pour les problèmes MCOP et TCP. La complexité temporelle de ces deux solutions est en $O(n^2)$, et une contrainte qu'elles ont en commun est : *l'indisponibilité, au moment de la conception, d'informations sur le graphe des tâches qui leur correspondent.*

Nous avons proposés un algorithme BSP/CGM basé sur l'algorithme séquentiel de Knuth, qui résout un problème OBST d'ordre n en $O(n^2/p)$ temps de calcul avec $\lceil (2p)^{1/2} \rceil$ rondes de communication sur p processeurs. La difficulté liée à cet algorithme vient du fait que la dépendance et la charge de calcul de chaque nœud de son graphe des tâches, ne sont connues qu'à l'étape qui précède immédiatement le traitement de ce nœud. Ainsi,

les informations sur le graphe des tâches ne peuvent être entièrement disponibles qu'à la fin de l'exécution. Néanmoins, avec notre division des tâches et notre distribution des données, ce nouvel algorithme BSP/CGM a les mêmes avantages que le premier, mais avec de meilleures performances.

Les résultats de l'implémentation de cet algorithme nous ont montré un temps global de communication beaucoup plus faible que celui obtenu avec la solution BSP/CGM générique, pourtant, deux fois plus de données sont partagées entre les processeurs. Ceci vient du fait que lorsqu'un processeur utilise trop de temps pour évaluer un ensemble de nœuds à une étape k de l'algorithme, il augmente le temps de latence des processeurs qui attendent ses résultats pour évaluer leurs nœuds lors des étapes suivantes. Cette latence des processeurs augmente le temps de communication global de l'algorithme. Dans la solution CGM générique, ce temps de latence est dû principalement à la complexité temporelle en $O(n^3)$ de l'algorithme générique séquentiel utilisé.

L'inconvénient de notre nouvel algorithme BSP/CGM basé sur l'accélération de Knuth vient du fait que le processeur d'un bloc d'une diagonale k attend que tous les blocs des diagonales inférieures à k soient évalués pour commencer à évaluer le sien, pourtant il n'utilise pas les données des blocs de toutes ces diagonales. Ceci a causé une latence inutile des processeurs, et donc, une perte de performances qu'il a fallu absolument rattraper. Pour résoudre ce problème, nous avons proposé un protocole qui permet aux processeurs, à l'aide des messages d'informations qu'ils s'envoient au fil des étapes de l'algorithme :

1. d'éviter l'envoie des informations inutiles à d'autres processeurs et ;
2. de commencer et terminer l'évaluation de leurs blocs aussitôt que les données dont ils ont besoin sont disponibles.

Ce protocole a permis que les blocs de plusieurs diagonales puissent être évalués en même temps : celles dont les données nécessaires à leur évaluation deviennent disponibles au même moment. Ces deux améliorations ont permis de diminuer les temps globaux de communication et de calculs locaux de notre algorithme BSP/CGM basé sur l'accélération de Knuth.
Il serai tout de même important d'affiner le protocole de communication proposé afin qu'il y ait moins de données inutiles communiquées entre les processeurs et moins de messages d'information qui circulent sur le réseau. De même, on pourrai envisager un traitement progressif pour cet algorithme afin d'améliorer son temps d'exécution.

Pour le problème MCOP, nous avons proposés un algorithme BSP/CGM basé sur la solution séquentielle de Yao. Il nécessite $O(n^2/p)$ temps de calcul sur chaque processeur avec moins de p rondes de communication.

Au moment de la conception, l'on ne dispose pas du graphe des tâches. Sa structure dépend de l'instance des données en entrée du problème et n'est connu qu'au moment de l'exécution. La conception statique habituellement utilisée dans les algorithmes BSP/CGM devient ainsi inexploitable, il faut donc automatiser la tâche de conception. Pour ce faire, nous nous sommes inspirés de l'*algorithme de conception dynamique* initié par Kechid et Myoupo [Kec09] que nous avons adapté à notre approche. Le partitionnement du graphe des tâches que nous avons proposé tire son efficacité du fait qu'*au moins la moitié des nœuds de ce graphe sont des feuilles*. Ainsi, nous avons proposé deux approches de partitionnement dynamique des données, une axée sur l'équilibrage des charges et l'autre axée sur la minimisation des communications. Dans les deux cas, plus de la moitié des calculs à effectuer sont réalisés par les processeurs sans aucune communication et avec un équilibrage de charges optimal.
La qualité des dépendances entre les nœuds du graphe des tâches fait qu'il y a peu de dépendances entre les différents sous-graphes détenus par les processeurs, et de ce fait, il y a peu de communication entre eux. Ainsi, quelque soit la méthode de partitionnement utilisée, le temps de communication est très faible comparé au temps de calcul. Ainsi, contrairement à tous les algorithmes BSP/CGM présentés jusqu'ici, dans celui-ci, le temps des calculs locaux constitue l'essentiel du temps global d'exécution de l'algorithme. Ainsi, les efforts lors de la conception d'algorithmes BSP/CGM basés sur l'accélération de Yao doivent être axés essentiellement sur l'équilibrage des charges entre les processeurs.

Nous avons appliqué le prétraitement supplémentaire que nous avons proposés à l'algorithme séquentiel de Yao à notre algorithme BSP/CGM et le résultat a été satisfaisant. Il a permis d'améliorer le temps des calculs locaux, et ainsi, d'améliorer le temps global d'exécution de l'algorithme.

Chacun des partitionnements des données (statiques ou dynamiques) proposés dans cette thèse permet de subdiviser le graphe des tâches en au plus $2p$ sous-graphes. Ainsi, pour tout problème dont la conception est similaire à celles décrites dans cette thèse, borner le nombre de sous-graphes (sous-problèmes) à évaluer par un processeur peut conduire à de bonnes performances.

Perspectives

Des expérimentations à grande échelle

Une des difficultés rencontrée dans notre travail a été l'accès au matériel adéquat pour effectuer les expérimentations. Les tests ont été effectués avec ceux dont nous disposions.

De ce fait, la première perspective à notre travail est la réalisation de simulations, à grande échelle, sur une machine parallèle plus adaptée, afin de consolider les tendances actuelles des performances obtenues à partir des simulations effectuées. Pour ce faire, on pourrai utiliser le cluster dénommé MAGI, situé au sein de l'Université de Paris 13, qui est constitué de 5 types de nœuds :

- *Nœuds de calcul* : dédiés au calcul parallèle via MPI ;
- *Nœud SMP* : dédié au calcul parallèle via OpenMP ;
- *Nœud Login* : nœud de connexion, les compilateurs y sont installés ;
- *Nœud Admin* : hébergeant les services courants pour le cluster (LDAP, SGBD, DNS, etc.) ;
- *Nœud E/S* : nœud d'entrées / sorties utilisant le système de fichier réparti GlusterFS.

Les nœuds sont tous reliés à l'aide du bus *infiniband* et ont les caractéristiques décrites dans la table 5.1.

Type de noeud	**Processeur**	**Mémoire**	**Quantité**	**Total**
Calcul	2 x Xeon X5670 6 coeurs (2.93GHz-6.4GT-12MB-95W)	2 Go / coeur	12	144 cœurs
SMP	4 x Xeon E7-4850 10 coeurs (2,00GHz-6.4GT-24MB-130W)	512 Go	1	
Login	2 x Xeon X5670 6 coeurs (2.93GHz-6.4GT-12MB-95W)	8 Go	1	
Admin	2 x Xeon X5670 6 coeurs (2.93GHz-6.4GT-12MB-95W)	8 Go	1	
E/S	2 x Xeon X5620 4 coeurs (2.40GHz-12MB)	24 Go	1	

TABLE 5.1 – Caractéristiques des nœuds du cluster MAGI.

Une accélération d'un algorithme CGM existant

L'algorithme CGM de Dilson et Marco [HS12] pour le problème MCOP, est est basé sur la solution séquentielle générique. Il nécessite $O(1)$ rondes de communication avec $O(n^3/p^3)$ temps de calculs locaux sur chaque processeur. On pourrai s'inspirer de cet algorithme pour produire des solutions héritant des accélérations de Knuth et de Yao.

Une automatisation de partitionnement

Puisqu'il est souvent difficile d'optimiser tous les critères de performances en même temps, on pourrai fournir aux utilisateurs finaux le choix d'optimiser un ou plusieurs d'entre eux. Et effectuer un choix en fonction des caractéristiques propres à la machine parallèle utilisé. Pour ce faire, il faudra concevoir un algorithme capable de s'adapter, par exemple, aux différents scénarios suivants :

1. diminuer la taille des blocs pour augmenter l'efficacité, mais en s'exposant à une augmentation des rondes de communication ;
2. opter pour des blocs de grande taille pour diminuer le nombre de rondes de communication mais en acceptant un déséquilibre de charge entre les processeurs ;
3. fixer le nombre de processeurs, et faire varier le grain de calcul, pour choisir celui qui offre le meilleur rapport efficacité/nombre de rondes de communication ;
4. etc.

On pourrai proposer un simulateur qui guide l'utilisateur dans son choix.

Un partitionnement de graphe non régulier

Ce type de partitionnement dynamique permettrai à l'algorithme de mieux exploiter les machines parallèles dont les processeurs n'ont pas tous à la même vitesse de calcul. Ainsi, distribuer les tâches en fonction de la puissance de calculs des processeurs.
En faisant varier la taille des blocs dans le graphe, on pourrai permettre aux processeurs de rester actif plus longtemps.

Publications liées à cette thèse

1. V. Kengne Tchendji and J. F. Myoupo. An Efficient CGM-Based Parallel Algorithm Solving the Matrix Chain Ordering Problem, *to appear in the International Journal of Grid and High Performance Computing (IJGHPC).*
2. Myoupo, J. F. and Tchendji, V. K. (2014) 'Parallel dynamic programming for solving the optimal search binary tree problem on CGM', *the International Journal of High Performance Computing and Networking*, Vol. 7, No. 4, pp. 269 - 280.
3. V. Kengne Tchendji and J. F. Myoupo. An Efficient Coarse-Grain Multicomputer Algorithm for the Minimum Cost Parenthesizing Problem, *the Journal of Supercomputing*, vol. 61 p. 463 - 480, issue 3 September 2012.
4. L. P. Fotso, V. K. Tchendji and J. F. Myoupo. Load Balancing Schemes for Parallel Processing of Dynamic Programming on BSP/CGM Model. *International Conference on Parallel Distributed Processing Technique and Applications (PDPTA'2010)*, Las Vegas July 2010.

Bibliographie

[ACD+02a] Carlos E. R. Alves, Edson Cáceres, Frank Kha Dehne, Siang W. Song, et al. A CGM/BSP parallel similarity algorithm. In *Brazilian Workshop on Bioinformatics*, pages 1–8, 2002.

[ACD02b] Carlos E. R. Alves, Edson Norberto Cáceres, and F. Dehne. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 275–281. ACM, 2002.

[ACDS03] Carlos E. R. Alves, Edson Norberto Cáceres, Frank Dehne, and Siang W. Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Computational Science and Its Applications-ICCSA 2003*, pages 249–258. Springer, 2003.

[ACS05] Carlos Eduardo Rodrigues Alves, Edson Norberto Cáceres, and Siang Wun Song. An all-substrings common subsequence algorithm. *Electronic Notes in Discrete Mathematics*, 19 :133–139, 2005.

[ACS06] Carlos E. R. Alves, Edson N. Caceres, and Siang Wun Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45(3) :301–335, 2006.

[Akl90] S. G. Akl. Parallel synergy or can a parallel computer be more efficient than the sum of its parts. Technical report, Technical report, 90-285, Queen's University, Kingston Ontario, 1990.

[Asp90] William Aspray. *John von Neumann and the origins of modern computing*, volume 191. MIT Press Cambridge, MA, 1990.

[Bis04] Rob H. Bisseling. *Parallel scientific computation*. Oxford University Press Oxford, 2004.

[BJVOR03] Olaf Bonorden, Ben Juurlink, Ingo Von Otte, and Ingo Rieping. The paderborn university BSP (pub) library. *Parallel Computing*, 29(2) :187–207, 2003.

[BML95] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Optimal parenthesization of lexical orderings for dsp block diagrams. In *VLSI Signal Processing, VIII, 1995. IEEE Signal Processing Society [Workshop on]*, pages 177–186. IEEE, 1995.

[Bra94] Phillip Gnassi Bradford. *Parallel dynamic programming.* PhD thesis, Indiana University, 1994.

[BSP13a] *BSP in the third millenium http ://www.bsp-worldwide.org/bspww3000.html.* 2013.

[BSP13b] *BSP machine parameters www.bsp-wordwide.org/implements/oxtool/params.htm.* 2013.

[CDBL08] Albert Chan, Frank Dehne, Prosenjit Bose, and Markus Latzel. Coarse grained parallel algorithms for graph matching. *Parallel Computing*, 34(1) :47–62, 2008.

[CS99] Franck Cappello and Jean-Paul Sansonnet. Introduction au parallélisme et aux architectures parallèles. *Techniques de l'ingénieur. Informatique*, (H1088) :H1088–1, 1999.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3) :251–280, 1990.

[Deh99] Frank Dehne. Guest editor's introduction : Special issue on coarse grained parallel algorithms. *Algorithmica*, 24(3/4) :173–176, 1999.

[Deh06] Frank Dehne. Guest editor's introduction : Special issue on coarse grained parallel algorithms for scientific applications. *Algorithmica*, 45(3) :263–267, 2006.

[DFC+02] Frank Dehne, Afonso Ferreira, Edson Cáceres, Siang W. Song, and Alessandro Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33(2) :183–200, 2002.

[DFRC93] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the ninth annual symposium on Computational geometry*, pages 298–307. ACM, 1993.

[Dur98] Richard Durbin. *Biological sequence analysis : probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[ECG93] Phil Edmonds, Eleanor Chu, and Alan George. Dynamic programming on a shared-memory multiprocessor. *Parallel Computing*, 19(1) :9–22, 1993.

[Fer96] A. Ferrera. *Parallel and communication algorithms for hypercube multiprocessor*. Handbook of Parallel and Distributed Computing (A. Zomaya, ed.) McGraw-Hill, 1996.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9) :948–960, 1972.

[Fos95a] E. R. Jessup C. J. Schauble Fosdick, Lloyd Dudley. *An introduction to high-performance scientific computing*. MIT Press, 1995.

[Fos95b] Ian Foster. *Designing and building parallel programs*, volume 95. Addison-Wesley Reading, 1995.

[Fru92] Mikhail Aleksandrovich Frumkin. *Systolic computations*. Kluwer Academic Publishers, 1992.

[Gen96] Marc Gengler. An introduction to parallel dynamic programming. In *Solving Combinatorial Optimization Problems in Parallel*, pages 87–114. Springer, 1996.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, Second Edition, January 16, 2003.

[Gib89] Phillip B. Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[GLRT95] Mark W. Goudreau, Kevin Lang, Satish B. Rao, and Thanasis Tsantilas. The green BSP library. *Report CS TR*, 95(11), 1995.

[GMS03] Thierry Garcia, J-F Myoupo, and David Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 349–356. IEEE, 2003.

[God73] Sadashiva S. Godbole. On efficient computation of matrix chain products. *Computers, IEEE Transactions on*, 100(9) :864–866, 1973.

[GP91] Zvi Galil and Kunsoo Park. Parallel dynamic programming. 1991.

[GP94] Zvi Galil and Kunsoo Park. Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2) :213–222, 1994.

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2) :251–267, 1994.

[H.96] Gurla H. *Time-optimal visibility algorithms on coarse-grain multiprocessors.* PhD thesis, August 1996.

[Hil98] J. Hill. BSP cost parameters, sorted by megaflop/s rate, for the oxford BSP toolset. *Computing Laboratory, Oxford University*, 1998.

[HMS97] M. Hill, W. McColl, and D. Skillicorn. Questions and answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[HMS+98] Jonathan Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib : The BSP programming library. *Parallel Computing*, 24(14) :1947–1980, 1998.

[HS84] T. C. Hu and M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2) :228–251, 1984.

[HS12] Dilson R. Higa and Marco A. Stefanes. A coarse-grained parallel algorithm for the matrix chain order problem. In *Proceedings of the 2012 Symposium on High Performance Computing*, page 1. Society for Computer Simulation International, 2012.

[IPS88] Oscar H. Ibarra, Ting-Chuen Pong, and Stephen M. Sohn. *Hypercube algorithms for some string comparison problems.* University of Minnesota, Institute of Technology, Computer Science Department, 1988.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., 1992.

[Kec09] Mounir Kechid. *La programmation dynamique non-serial dans les modèles de calcul parallèle BSP/CGM.* PhD thesis, 2009.

[KK93] George Karypis and Vipin Kumar. Efficient parallel formulations for some dynamic programming algorithms. In *Proc. of the 7th International Parallel Processing Symposium (IPPS)*. Citeseer, 1993.

[KM05] Mounir Kechid and Jean Frederic Myoupo. Towards a more realistic BSP cost model. In *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 10–pp. IEEE, 2005.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1) :14–25, 1971.

[KT06] Peter Krusche and Alexander Tiskin. Efficient longest common subsequence computation using bulk-synchronous parallelism. In *Computational Science and Its Applications-ICCSA 2006*, pages 165–174. Springer, 2006.

[Lei92] Frank Thomson Leighton. *Introduction to parallel algorithms and architectures : Arreys. Tree. Hypercube.* Morgan Kaufmann San Francisco, 1992.

[LKHL03] Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee. Processor allocation and task scheduling of matrix chain products on parallel systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(4) :394–407, 2003.

[LRSC01] Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, and Thomas H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[LT88] Basile Louka and Maurice Tchuente. Dynamic programming on two-dimensional systolic arrays. *Information Processing Letters*, 29(2) :97–104, 1988.

[LW85] Guo-Jie Li and Benjamin W. Wah. Parallel processing of serial dynamic programming problems. In *Proceedings of COMPSAC*, volume 85, pages 81–89, 1985.

[MA97] Ferreira M. Morvan A. *Parallel Computing in Optimization.* Models for Parallel Algorithm design : an Introduction, Kluwer Academic Publisher, 1997.

[McC94] W. F. McColl. BSP programming. In *Proceedings DIMACS Workshop*, volume 18, pages 21–35, 1994.

[McC95a] W. F. McColl. The BSP approach to architecture independent parallel programming. *CACM on General Purpose Practical Models of Parallel Computation*, 1995.

[McC95b] William F. McColl. Bulk synchronous parallel computing. In *Abstract machine models for highly parallel computers*, pages 41–63. Oxford University Press, 1995.

[McC96] W. F. McColl. Scalable computing. In *Computer Science Today : Recent Trends and Developments*, 1996.

[Mil93] Richard Miller. A library for bulk-synchronous parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 100–108, 1993.

[Mil94] Richard Miller. *Two-approaches to architecture-independent parallel computation.* PhD thesis, University of Oxford, 1994.

[MSDS10] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst D. Simon. Top500 supercomputer sites. *The report can be downloaded from http ://www.netlib.org/benchmark/top500.html*, 2010.

[Myo91] Jean Frédéric Myoupo. Dynamic programming on linear pipelines. *Information Processing Letters*, 39(6) :333–341, 1991.

[Myo92] Jean Frédéric Myoupo. Synthesizing linear systolic arrays for dynamic programming problems. *Parallel Processing Letters*, 2(01) :97–110, 1992.

[Myo93] Jean Frédéric Myoupo. Mapping dynamic programming onto modular linear systolic arrays. *Distributed computing*, 6(3) :165–179, 1993.

[Ola08] Stephan Olariu. *Transitional Issues : Fine-Grain to Coarse-Grain.* Handbook of parallel computing : models, algorithms and applications by Taylor & Francis Group, LLC., 2008.

[PW013] *The Paderborn University BSP Web Computing (PUB-Web) Library http ://pubweb.cs.uni-paderborn.de/index.php.* 2013.

[Rai92] Sanjay Raina. Virtual shared memory : A survey of techniques and systems. *University of Bristol, Bristol, UK*, 1992.

[Ryt88] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59(3) :297–307, 1988.

[Ski02] D. B. Skillicorn. Predictable parallel performance : The BSP model. *Applied Optimization*, 67 :85–116, 2002.

[SL06] Moshe Sniedovich and Art Lew. Dynamic programming : an overview. *Control and Cybernetics*, 35(3) :513, 2006.

[Sni89] Marc Snir. Parallel computation models-some useful questions. In *Opportunities and Constraints of Parallel Computing*, pages 139–145. Springer, 1989.

[Sny86] Lawrence Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual review of computer science*, 1(1) :289–317, 1986.

[Son01] S. W. Song. Design of efficient and scalable parallel algorithms. *Scalable Computing : Practice and Experience*, 3(3), 2001.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2) :123–169, 1998.

[TG95] D. Tang and G. Gupta. An efficient parallel dynamic programming algorithm. *Computers & Mathematics with Applications*, 30(8) :65–74, 1995.

[UD97] Ivan Uthus and Haakon Dybdahl. Simulation of the BSP model on different computer architectures. *Manuscript. Norwegian University of Science and Technology, Department of Computer Science*, 1997.

[Val89] Leslie G. Valiant. Bulk-synchronous parallel computers. *Parallel Processing and Artificial Intelligence*, pages 15–22, 1989.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, 1990.

[Was94] C. Wasel. *Parallel Computer Taxonomy.* M. Phil Aberystwyth University, 1994.

[Yao82] F. Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4) :532–540, 1982.