N° d'ordre: 2509

THÈSE

présentée

devant l'Université de Yaoundé I

pour obtenir

le grade de : DOCTEUR DES UNIVERSITÉS DE RENNES 1 ET DE YAOUNDÉ 1
Mention : INFORMATIQUE

par

Claude Martin TADONKI

Équipe d'accueil :
IRISA, projets ALADIN et COSI, Université de Rennes I
Département d'Informatique de l'Université de Yaoundé I

Titre de la thèse :

Contributions à l'Algorithmique Parallèle

Soutenue le 1er mars 2001 devant la commission d'examen

M. :	Patrice	QUINTON	Président
MM. :	Denis	TRYSTRAM	Rapporteurs
	Philippe	CLAUSS	
MM. :	Emmanuel	KAMGNIA	Examinateurs
	Bernard	PHILIPPE	
	Maurice	TCHUENTE	

La machine elle-même, plus elle se perfectionne, plus elle s'efface derrière son rôle. Il semble que tout l'effort industriel de l'homme, tous ses calculs, toutes ses nuits de veille sur les épures, n'aboutissent, comme signes visibles, qu'à la seule simplicité, comme s'il fallait l'expérience de plusieurs générations pour dégager peu à peu la courbe d'une colonne, d'une carène, ou d'un fuselage d'avion, jusqu'à leur rendre la pureté élémentaire de la courbe d'un sein ou d'une épaule.

Antoine de Saint-Exupéry, *Terre des hommes, 1931*

Remerciements

Ce travail n'aurait pas pu aboutir sans le concours actif de plusieurs personnes et institutions.

Je remercie les dirigeants et le personnel des Universités de Rennes I (France) et de Yaoundé I (Cameroun), pour leurs efforts academiques et administratifs qui ont entre autre permis de créer le cadre institutionnel de ce travail.

Je remercie, dans l'ordre alphabétique, les Professeurs Bernard PHILIPPE, Patrice QUINTON, et Maurice TCHUENTE, pour leur rôle d'encadreurs qu'ils ont joué tout au long de ce travail et même avant son commencement. Je ne saurais, en un seul paragraphe, énumérer et expliciter l'ensemble des actions qu'ils ont entreprises dans le but de valoriser mon travail et cultiver mes réflexes de jeune chercheur.

Je remercie les Professeurs Philippe CLAUSS et Denis TRYSTRAM pour avoir accepté la charge de rapporteur. Leurs multiples remarques auront permis une importante amélioration de la qualité de ce travail.

Je remercie le Dr Emmanuel KAMGNIA pour ses enseignements dans le domaine du calcul numérique.

Je remercie l'agence française *Aire développement* pour son soutien matériel et financier. Ce soutien très objectif aura été déterminant dans la mise en oeuvre d'importantes collaborations scientifiques. Dans le même ordre d'idées, j'adresse mes vifs remerciements au Ministère Français de la Recherche Scientifique.

Je remercie l'ensemble des chercheurs de l'équipe *Calcul Parallèle* et les chercheurs du laboratoire IRISA, pour leur collaboration scientifique fructueuse et l'esprit amical dans lequel nous avons évolué.

J'adresse mes sincères remerciements a Sanjay Rajopadhye pour l'ensemble des travaux réalisés ensemble et l'esprit de rigueur qu'il m'a aidé à cultiver et entretenir. Je remercie aussi Tanguy Risset pour sa collaboration permanente.

Je remercie toutes les personnes qui m'ont accompagné et soutenu tout au long de cette aventure de these. Ils s'agit entre autres de Moulaye, Gilles, Emeric, Fabrice, Lamine, Emmanuel, Winston, Patrick, Réné, Clémentin, François, Gonzalo, Ferry, Marie-Claude, Veronique, Karine, Christiane, Elisabeth, Bénédicte, et tous ceux que ne j'ai pas pu citer mais qui se reconnaisent appartenir à cette liste.

Je remercie ma famille, mes amis, et plus généralement tous les miens, pour leur soutien permanent et sans limite.

Dédicace

Je dédie ce travail à

- A Esther et Jean TADONKI

- A Ernestine, Ange Dominique, et Fred William

- A tous mes frères et soeurs

- Aux familles Akame, Kuete, Moudissa, Nkuete

Introduction

L'algorithmique, considérée comme support de base de tout processus informatique, a connu en cette fin de siècle d'importants développements. De nouveaux paradigmes ont vu le jour, donnant lieu à l'émergence des méthodes nouvelles d'analyse et de conception d'algorithmes [16, 26]. Parallèlement, les études de complexité ont été poursuivies, et ont conduit à une classification des problèmes par rapport à leur difficulté intrinsèque [58], érigeant du même coup des barrières de performances.

Lorsqu'on connaît le rôle effectif actuel de l'informatique, aussi bien dans les environnements de haute technologie que dans les mécanismes les plus usuels, on comprend qu'on soit très vite arrivé à une relative insatisfaction.

Fort heureusement, des efforts importants ont été consentis dans les différentes disciplines impliquées dans la conception des processeurs. Cette synergie a abouti à ce que les processeurs actuels ont des performances de traitement très élevées, et cette croissance évolue à un rythme assez rassurant [138]. Mais, comme on pouvait s'y attendre, cette rapide évolution a reveillé des envies de puissance beaucoup plus importantes [121]. Ceci se justifie par des besoins plus accrus en précison de calcul, le traitement des données de très grande taille, ou tout simplement le passage de la théorie à la pratique de certains concepts des mathématiques discrètes et combinatoires. Dans cette dernière gamme, on note l'utilisation des techniques heuristiques [62], qui sont une preuve que les performances nécessaires sont encore, et très souvent, au-delà du possible.

Il apparaît alors qu'il faut utiliser d'autres armes, et c'est là qu'intervient le concept de *parallélisme* [87, 34, 61, 89, 35]. Cette idée n'est certes pas récente, mais les développements technologiques nécessaires à une mise en oeuvre satisfaisante le sont presque. Ceci justifie bien le fait que le parallélisme suscite un intérêt croissant sur le plan scientifique, technique, et économique.

La tâche serait très aisée s'il s'agissait tout simplement de subdiviser un algorithme séquentiel en autant de sous-tâches que l'on désire, et de les exécuter simultanément sur plusieurs unités de traitements. Malheureusement, on en est très loin [65], et les choses sont encore beaucoup plus compliquées qu'on ne peut l'imaginer intuitivement [155, 57, 103].

En effet, d'un point de vue purement algorithmique, il s'agit de bâtir un ordre d'exécution des instructions qui soit compatible avec les dépendances éventuelles, et ensuite de mettre en place une tratégie cohérente de répartition des tâches entre les différentes unités de traitement [27, 88, 45, 114]. Etant donné que ces unités de traitement devront sans doute coopérer entre elles par des échanges d'informations [131] ou des actions de

synchronisation [108], une autre tâche moins évidente consiste à expliciter ces différents aspects [153, 129].

Tout comme en algorithmique classique, les développements de l'algorithmique parallèle reposent sur les *modèles de machines parallèles* [32, 76, 157]. Toutefois, contrairement au cas séquentiel où un modèle universel de machine, appelé modèle de Von Neumann [152], a pu être défini à partir des travaux d'Alan Turing [154] et autres chercheurs tels que Church [29], on ne dispose pas d'un modèle universel de machine parallèle. Plusieurs classifications de machines parallèles ont été proposées dans la littérature. Elles sont pour certaines d'entre elles basées sur la nature des flots de données et des flots d'instructions [51]. Ainsi, on distingue dans un premier temps deux grandes classes qui sont : celle des machines SIMD (flots d'instructions unique agissant sur des données multiples) et celle des machines MIMD (flots d'instructions et de données multiples). A côté de ces deux aspects, il faut ajouter la prise en compte des phénomènes de synchronisation, la granularité des calculs, la répartition de la mémoire, et les protocoles de communication [4, 108, 61]. On comprend alors la difficulté de définir un modèle universel de machine parallèle.

Ceci ne nuit en rien à l'activité du parallélisme. Pour une application donnée, il suffit de cibler un modèle de machine et ensuite d'adopter la démarche algorithmique appropriée. A ce niveau, deux tendances animent les activités de recherche: l'approche directe qui part d'une spécification du problème à l'algorithme parallèle [27, 114, 133, 94], et l'approche indirecte qui consiste à paralléliser un algorithme séquentiel existant [8, 7, 109, 48]. La première approche offre plus de liberté au théoricien qui peut s'inspirer des paradigmes de son choix, ou alors reformuler le problème à sa guise, le but étant bien sûr d'arriver à un résultat satisfaisant en fonction d'un critère de référence. La deuxième approche quant à elle vise surtout le public utilisateur qui, disposant déjà d'un algorithme séquentiel, ne possède pas toujours le savoir faire requis (ou le temps nécessaire) pour bâtir une version parallèle efficace. Cette voie à déjà porté ses fruits dans les langages à parallélisme de données tels que HPF (High Performace Fortran) [78].

Le parallélisme aujourd'hui est une discipline dont l'impact est réel dans les activités de l'industrie, de la recherche, de l'économie, de la prévision, de la simulation, et bien d'autres [86, 96]. Ceci est dû, en plus des acquis algorithmiques, à des efforts techniques importants qui ont conduit à ce que l'on dispose de véritables architectures parallèles [141, 96, 59, 55, 61]. Leur utilisation de plus en plus régulière est aussi due au développement des langages parallèles et des bibliothèques annexes [102, 78, 91, 60, 37]. Ajoutons à cela les idées du *ClusterComputing* [105, 5] et du *MétaComputing* [52, 3], grâce auxquelles des entités préablement vouées aux calculs séquentiels vont devoir participer à la mise en oeuvre de processus parallèles. Notons aussi l'essor des systèmes embarqués, et le développement des processeurs EPIC (Explicitly Parallel Instruction Computing). Tout ceci justifie bien le fait qu'à l'heure actuelle, les activités liées au parallélisme sont de plus en plus nombreuses et prennent une importance de premier rang.

Le travail que nous présentons dans cette thèse est dans un premier temps de nature didactique. C'est ainsi que le chapitre 1 fournit une vue synthétique et pédagogique des concepts et techniques du parallélisme, allant des idées fondamentales au techniques de

conception, d'analyse, et de mise en oeuvre. Dans les chapitres restants, nous présentons nos différentes contributions, qui sont le fruit des travaux réalisés dans le cadre des projets *Calcul Parallèle* (Université de Yaoundé I au Cameroun), ALADIN et COSI de l'IRISA (Université de Rennes I en France), et des collaborations personnelles avec des chercheurs de la communauté du parallélisme et disciplines associées. Ainsi, dans le chapitre 2, nous présentons la technique d'*ordonnancement canonique*, technique basée sur la reproduction locale d'un ordonnancement générique partiel. En effet, lorsqu'on a deux graphes isomorphes, un ordonnancement sur l'un peut être naturellement transporté sur l'autre à l'aide de l'isomorphisme considéré. En reprenant cette démarche d'un sous-ensemble à l'autre (dans une *décomposition canonique* du graphe de tâches), on obtient un ordonnancement global du graphe. Il s'agit en fait d'un ordonnancement dans lequel chaque sous-ensemble de tâches est traité exclusivement par le groupe de processeurs qui lui est associé, et, d'un sous-ensemble à l'autre, on a un *pipeline macroscopique*. Cette technique marche bien sur des schémas de type programmation dynamique, et elle n'est pas liée à la nature des dépendances (uniformes, affines, etc.), de sorte qu'on peut envisager l'appliquer à une classe assez étendue de problèmes. Le chapitre 3 est consacré à la parallélisation du produit de Kronecker et ses différentes applications. Le produit de Kronecker est une opération matricielle, bien adaptée pour la modélisation des phénomènes d'interaction, et l'expression algébrique de certains concepts importants. Les principales difficultés que l'on rencontre dans son usage sont d'une part d'ordre matériel (mémoire), et d'autre part d'ordre calculatoire (forte possibilité d'avoir des calculs redondants). Notre objectif était de construire des algorithmes parallèles efficaces pour cette opération, sans perdre de vue les deux précédents facteurs. La solution que nous proposons est basée sur une décomposition algébrique des calculs, que l'on alloue ensuite à un ensemble de processeurs organisés en une topologie de type *hypercube*. L'algorithme est de type SPMD, et par ailleurs, une analyse du coût des communications et de l'espace mémoire montre qu'il est globalement optimal. Dans le chapitre 4, nous revisitons le célèbre problème du *chemin algébrique* et apportons des solutions parallèles efficaces. Les algorithmes développés dans les chapitre 3 et 4 ont été testés sur des machines parallèles standards telles que la INTEL PARAGON, la CRAY T3E, la NEC CENJU. C'est fort des expériences acquises lors de ces différentes études, et d'une observation sur la nature des solutions, que nous sommes arrivés à une méthodologie qui se veut être *unificatrice* de nos différentes approches. Le chapitre 5 quant à lui présente un échéancier systolique ayant par ailleurs fait l'objet d'un dépot de brevet d'invention. Initialement, il s'agissait d'une architecture parallèle pour le tri d'un ensemble de valeurs, solution basée sur l'algorithme séquentiel du *tri par tas*. La transformation en un échéancier aura été faite dans le but de répondre à un besoin réel dans la technologie ATM. Le chapitre 6 conclut notre travail et présente quelques perspectives.

Chapitre 1

Les différents aspects du parallélisme

1.1 Une vision globale

Le parallélisme est une méthode de traitement de l'information qui permet, en cours d'exécution, l'exploitation d'événements concurrents [61]. Ces événements peuvent être de nature globale (entre instructions) ou locale (à l'intérieur d'une instruction).

Le parallélisme global nécessite la décomposition du traitement en tâches, la recherche des dépendances entre ces tâches et ensuite l'exécution en parallèle des tâches indépendantes. Comme nous l'avons déjà souligné, cette activité peut se faire par une analyse algorithmique d'une spécification de base du problème, ou automatiquement par des compilateurs spécialisés.

Au niveau instruction, on distingue entre autres le parallélisme de contrôle (point fort des processeurs EPIC), et la *vectorisation* [13]. Concernant la vectorisation, elle peut se faire au niveau matériel (exécution, accès mémoire, entrées/sorties vectorielles) [130, 70, 33], au niveau du système (vectoriseur) [1], au niveau du langage (compilateurs vectoriels) [90, 14], et au niveau algorithmique (paradigme du pipeline) [69, 20, 35].

Tout ceci illustre la diversité des aspects impliqués dans le développement du parallélisme. Sachant que le but principal est la réduction du temps de calcul, revenons sur les fondements et analysons l'idée sous-jacente à de tels développements.

Thèse 1 (Thèse du Calcul Parallèle) *[34] Tout problème qui peut être résolu sur un ordinateur séquentiel raisonnable en utilisant un espace de taille polynomiale peut être résolu en temps polynomial sur un ordinateur parallèle raisonnable et vice-versa.*

Dans ce postulat, la notion d'ordinateur *raisonnable* n'est pas précise. Sur cette question, un consensus a abouti à la *Thèse de l'invariance* qui s'énonce comme suit.

Thèse 2 (Thèse de l'invariance) *[34] Des machines raisonnables peuvent se simuler entre elles avec au plus un accroissement polynomial en temps et une multiplication constante de l'espace.*

Etudions maintenant en détails les modèles et outils permettant de quantifier les différents arguments évoqués.

1.2 Modèles de machine parallèle et complexité

1.2.1 Modèles de machines

D'une manière générale, un modèle de machine parallèle comprend plusieurs processeurs, chacun opérant sur une portion d'un ensemble d'instructions concourant à la résolution d'un problème donné. Les caractéristiques suivantes [34] vont nous permettre de preciser ce propos.

a) Multiplicité des flots d'instructions et de données
Selon que le flot d'instructions (resp. de données) est unique ou multiple, on distingue quatre classes de machines (SISD, SIMD, MISD, et MIMD) dont les trois dernières correspondent à des architectures parallèles, tandis que la première (SISD) correspond tout simplement au modèle séquentiel de Von Neumann.

SIMD. Dans le modèle SIMD (**S**ingle **I**nstruction stream, **M**ultiple **D**ata streams), tous les éléments de calcul sont coordonnés par un séquenceur (unité de contrôle) unique, chaque entité travaillant sur son propre espace de données. Ces espaces de données peuvent être soit resserrés, on parle alors de *Mémoire Partagée* (SM pour Share Memory), soit disjoints, ce qui correspond au modèle dit à *Mémoire Distribuée* (DM pour Distributed Memory).

MISD. La classe MISD (**M**ultiple **I**nstruction streams, **S**ingle **D**ata stream) est assez spéciale car elle applique plusieurs flots d'instructions à un unique flot de données. A première vue, son impact pratique n'est pas évident. Toutefois, on peut théoriquement lui faire correspondre le mode d'exécution en *pipeline*.

MIMD. La classe MIMD (**M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams) est la plus répandue. Ceci est tout à fait normal car, appliquant un flot multiple d'instructions à un flot multiple de données, non seulement elle regroupe théoriquement (par simulation) les autres classes, mais aussi elle modélise un schéma d'exécution beaucoup plus souple. Ici aussi, il faut distinguer les deux modes de partage de l'espace global des données.

On peut donc pratiquement retenir quatre grandes classes : SIMD-SM, SIMD-DM, MIMD-SM, et MIMD-DM. La figure Fig. 1.1 (inspirée de [61]) illustre les configurations correspondantes.

b) Synchronisation et granularité
Dans un modèle *synchrone*, il y a une synchronisation automatique entre les processeurs à chaque étape du traitement. Dans le modèle *asynchrone*, c'est au concepteur que revient la charge de spécifier toute action de synchronisation. La *granularité de*

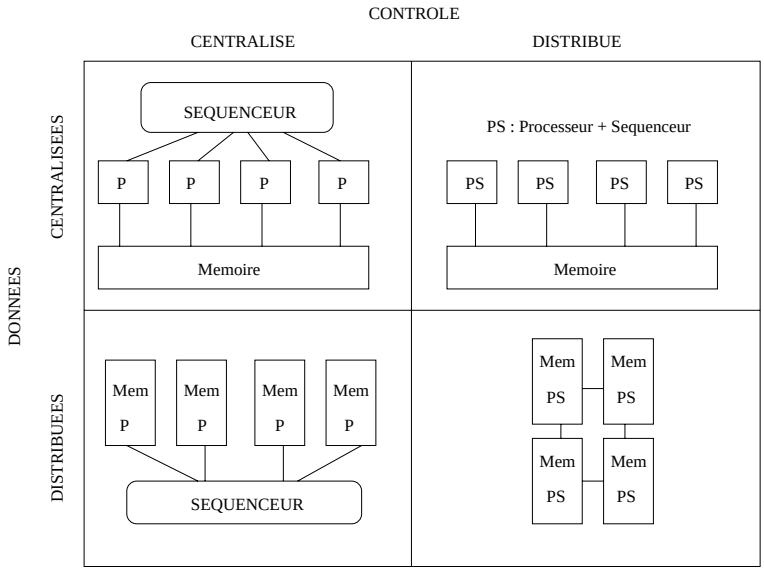


FIG. 1.1 – *Quatre grandes classes d'architectures parallèles.*

synchronisation, caractéristique des modèles *asynchrones*, correspond à la fréquence de synchronisation. De manière générique, on pourait définir des blocs de synchronisation dont la taille peut varier d'une opération (modèle systolique par exemple) à l'ensemble du programme en entier (modèle asynchrone). Cette notion est très importante en pratique car elle permet d'assurer une certaine cohésion temporelle très souvent nécessaire lorsqu'il faut faire intervenir des mécanismes de contrôle de flux ou de régulation de charge, ou tout simplement lorsqu'il faut effectuer des actions d'échanges d'informations.

c) Mécanismes de communication et Modes de gestion de la mémoire
Quel que soit le modèle de machine parallèle considéré, les processeurs doivent communiquer pour s'échanger des informations. Cette communication peut s'effectuer par *transmission de messages* (modèle à mémoire distribuée), ou à travers une *mémoire* (modèle à mémoire partagée). Il existe aussi des modèles à mémoire partagée basées sur une architecture à mémoire distribuée, on parle de DSM (Distributed Share Memory) ou NUMA (Nonuniform Memory Access). Dans ces modèles, chaque processeurs peut aussi bien accéder à sa mémoire locale qu'à celle des autres processeurs. Le modèle à mémoire distribuée requiert des connexions physiques entre les processeurs, ce qui peut parfois nuire à la *portabilité* d'un programme parallèle, surtout lorsque le concepteur a largement pris en compte une topologie bien précise. Dans certains cas, un changement de topologie peut être fatal (impossibilité d'exécuter le programme à cause par exemple des phénomènes d'interblocage), et dans d'autres cas, on peut noter une différence de performance très importante en fonction de l'adéquation entre le réseau virtuel et le réseau physique de communications.

Nous allons maintenant définir un modèle théorique de machine parallèle, le modèle PRAM (Parallel Random Access Machine) [54], modèle analogue au modèle RAM (Random Access Machine) utilisé en algorithmique séquentielle.

Le modèle PRAM
Le modèle PRAM est une abstraction des machines à mémoire partagée et constitue de ce fait un outil fondamental pour l'expression d'algorithmes parallèles et l'analyse de complexité. Globalement, une PRAM peut se décrire comme suit:

- Un ensemble non borné de processeurs notés $P_0, P_1, \cdots, P_i, \cdots$, chaque processeur connaissant son indice ou *étiquette*.
- Chaque processeur dispose d'un accumulateur, d'une mémoire locale non bornée et inaccessible aux autres processeurs, d'un compteur ordinal, et d'un registre booléen qui indique s'il est actif ou non. Le fonctionnement des processeurs est *synchrone*.
- Une mémoire globale partagée non bornée. Elle est accessible en lecture et en écriture par tous les processeurs.
- Un algorithme exécuté sur une machine PRAM consiste en une séquence finie d'instructions éventuellement étiquetées. Ces instructions peuvent être des lectures/écriture en mémoire globale, des branchements à une étiquette du programme (*saut*), ou des instructions arithmétiques et logiques d'arité bornée. Notons en plus l'existence de deux instructions spéciales qui sont *halt* et *fork*. L'instruction *halt* met un terme au fonctionnement du processeur exécutant, ou de l'ensemble de tous les processeurs si elle est évoquée par le processeur P_0. L'instruction *fork<étiquette>*, lorsqu'elle est appelée par le processeur P_i, déclenche le démarrage d'un calcul sur un processeur libre P_j après avoir vider la mémoire locale de P_j, recopie le contenu de l'accumulateur de P_i dans celui de P_j, et place *étiquette* dans le compteur ordinal de P_j. Chaque instruction a un coût unitaire (importante hypothèse pour l'analyse de complexité).

Etant donné que la mémoire globale est accessible à tous les processeurs, il peut se produire des conflits d'accès en lecture ou en écriture lors des accès simultanés (accès *concurrents*). Ceci a conduit à une classification des PRAM selon qu'on autorise ou non des lectures ou écritures concurrentes. Ainsi, on distingue :

- **Le modèle PRAM EREW.** Dans la PRAM EREW (*Exclusive Read - Exclusive Write*), les accès concurrents, aussi bien en lecture qu'en écriture, sont prohibés. C'est le modèle le plus restrictif mais suffisament réaliste pour donner lieu à une mise en oeuvre réelle.
- **Le modèle PRAM CREW.** CREW (*Concurrent Read - Exclusive Write*), autorise des lectures concurrentes mais interdit les access concurrents en écriture. Ce modèle peut aussi s'implémenter en utilisant par exemple les mécanismes de diffusions.
- **Le modèle PRAM ERCW.** Ce modèle ERCW (*Exclusive Read - Concurrent*

Write) interdit les lectures concurrentes mais admet des écritures concurrentes sur une même case mémoire. Sur cet aspect, on distingue plusieurs protocoles pour la gestion de la concurrence.

- Tous les processeurs doivent écrire la même valeur (mode *commun*). La concurrence en écriture n'est donc pas très significative.
- Une valeur est choisie au hasard parmi toutes celles qui sont proposées en écriture (mode *arbitraire*).
- La valeur choisie est celle du processeur prioritaire (la priorité étant bien précisée, par exemple un critère sur le rang). On parle de mode *prioritaire*.
- La valeur écrite est le résultat d'une fonction associative appliquée aux entrées proposées par les processeurs en concurrence (mode *fusion*).

Dans le cas particulier où une fonction associative est appliquée aux entrées, les mesures de complexité sont parfois fournies au facteur près du coût de la fonction implémentée. Une attention spéciale doit donc être accordée à de tels cas.

- **Le modèle PRAM CRCW.** Le modèle CRCW (*Exclusive Read - Concurrent Write*) autorise des lectures et écritures concurrentes.

Notons que les modèles que nous venons de décrire ne prennent pas en compte les cas où la concurrence concerne une écriture et une lecture. Ce problème est résolu en accordant un ordre de priorité soit à la lecture, soit à l'écriture.

Il convient de rappeler que les différents modèles présentés offrent des points de vue différents en matière de complexité. Toutefois, quel que soit le modèle sous-jacent à une analyse de complexité, il semble bien que l'on reste dans des classes voisinnes, même si une légère différence peut parfois avoir une importante signification en théorie.

1.2.2 Complexité parallèle

Il est naturel, tout comme en algorithmique classique, de s'intéresser à l'aspect quantitatif du comportement des algorithmes parallèles [104, 32, 79]. Nous allons présenter une vue synthétique du sujet. Le lecteur particulièrement intéressé pourra se referer à [76, 127].

L'étude de la complexité d'un algorithme parallèle s'exécutant sur une PRAM est basée sur deux facteurs :

- la *surface*, qui correspond au nombre de processeurs effectivement impliqués dans le déroulement de l'algorithme sur la PRAM.
- le *temps*, qui représente le nombre de pas de calculs (ou instructions) nécessaires à l'exécution de l'algorithme. Les hypothèses de fonctionnement synchrone et de coût unitaire de chaque instruction rendent ce décompte plus aisé et objectif.

Notons que ces mesures sont faites de manière *asymptotique*, c'est à dire qu'on ne s'intéresse qu'aux ordres de grandeur, que l'on exprime à l'aide des opérateurs O, Ω, et Θ définis ainsi qu'il suit sur une fonction quelconque f à variable entière:

$$O(f) = \{t : N \to N | g \leq cf, c \in N\} \tag{1.1}$$

$$\Omega(f) = \{g : N \to N | g \geq cf, c \in N\} \tag{1.2}$$

$$\Theta(f) = \{g : N \to N | c_1 f \leq g \leq c_2 f, c_1, c_2 \in N\} = O(f) \cap \Omega(f) \tag{1.3}$$

Les mesures de la *surface* et du *temps* d'un algorithme parallèle sont parfois appréciées de manière conjointe à travers la notion de *travail* d'un algorithme, qui se définit comme suit.

Définition 1 (Travail d'un algorithme parallèle). *On appelle travail d'un algorithme parallèle $\mathcal{A}$, la quantité notée W définie par*

$$W = P(\mathcal{A}) \times T_{//}(\mathcal{A}), \tag{1.4}$$

où $P(\mathcal{A})$ (resp. $T_{//}(\mathcal{A})$) est la surface (resp. le temps d'exécution) de l'algorithme A. □

Cette notion traduit le fait que l'on a envie de se ramener à une vision séquentielle du coût de l'algorithme. En effet, pour un algorithme séquentiel, le *travail* correspond tout simplement au *temps* de l'algorithme. Dans le cas parallèle, le *travail* pourrait donc être vu comme le coût séquentiel équivalent.

Définition 2 (Algorithme parallèle efficace). *Un algorithme parallèle est dit efficace si son temps d'exécution est poly-logarithmique et si son travail est égal au temps du meilleur algorithme séquentiel connu à un facteur poly-logarithmique près. Un tel algorithme a donc un coût en temps raisonnable et impossible à atteindre en séquentiel, avec un travail plus important mais raisonnable par rapport au meilleur algorithme séquentiel.* □

On retrouve là un concept qui est d'une certaine manière lié à la *thèse du calcul parallèle*. Il est à noter que l'hypothèse selon laquelle un algorithme séquentiel prend au moins un temps linéaire (et donc asymptotiquement plus important que du poly-logarithme) est argumentée par le fait qu'il faut au moins lire toutes les entrées du problème. Attention à ne pas confondre avec les cas où certaines entrées pouraient ne pas être consultées du fait d'une stratégie algorithmique quelconque (la recherche dichotomique par exemple). En effet, dans de tels cas, une hypothèse importante est faite sur les données du problème (dans le cas de la recherche dichotomique, on suppose que les données sont préalablement triées).

Définition 3 (Algorithme parallèle optimal). *Un algorithme parallèle sera dit optimal s'il est efficace et si son travail est du même ordre de grandeur que le temps d'exécution du meilleur algorithme séquentiel connu.* □

Cette permanente référence au schéma séquentiel n'est pas qu'un simple artifice permettant de qualifier un algorithme parallèle. Il permet aussi d'agir judicieusement sur sa performance globale. En effet, partant d'un algorithme parallèle initial, on peut opérer des aménagements dans le but d'améliorer son travail ou de payer le prix sur un paramètre pour en améliorer un autre. Cette démarche s'appuie sur un principe connu sous le nom du *principe de Brent* [17] qui s'énonce comme suit:

Le travail d'un algorithme parallèle est du même ordre que le temps d'exécution de sa simulation séquentielle ou que le travail de sa simulation sur un nombre réduit de processeurs.

On comprend alors que si on réduit suffisament le nombre initial de processeurs (augmentant ainsi la charge par processeur), on peut espérer arriver à un travail meilleur [35, 127]. Ceci suppose bien sûr que l'on dispose déjà d'un algorithme parallèle initial. Diverses techniques permettent de construire de tels algorithmes: exploitation directe du graphe des dépendances, diviser pour paralléliser, pour ne citer que ces deux là. Notons aussi l'existence d'une technique qui consiste à réduire l'effet des dépendances par l'introduction des redondances. Dans un contexte beaucoup plus pratique, cette approche permet de réduire le volume des communications entre les différents processeurs. Pour l'instant, nous faisons abstraction de cet aspect sur lequel nous reviendrons plus tard.

Les éléments quantitatifs que nous venons de présenter peuvent être combinés, ceci afin d'évaluer des paramètres qui sont beaucoup plus significatifs de la qualité des algorithmes considérés.

1.2.2.1 Paramètres d'appreciation d'un algorithme parallèle

D'une manière générale, un algorithme sera caractérisé par des grandeurs sans dimension telles que son *accélération* et son *efficacité*.

Définition 4 (Accélération). *Soit $T_p(n)$ le temps nécessaire à un algorithme parallèle pour résoudre une instance de taille n d'un problème avec p processeurs. On appelle accélération le rapport $\sigma(n,p)$ défini par*

$$\sigma(n,p) = \frac{T_1(n)}{T_p(n)}. \tag{1.5}$$

□

Il s'agit en fait d'une appréciation de l'impact du nombre de processeurs sur le temps de l'algorithme, en prenant comme référence le coût d'une exécution séquentielle. Si les algorithmes choisis sont les meilleurs possibles en temps, on obtient une accelération particulière qui caractérise le potentiel de parallélisme du problème considéré.

On peut aussi définir une grandeur permettant un jugement plus absolu par rapport à la surface requise par l'algorithme.

Définition 5 (Efficacité). *Soit $T_p(n)$ le temps nécessaire à un algorithme parallèle pour résoudre une instance de taille n d'un problème, et soit $\sigma(n,p)$ son facteur d'accélération. On définit l'efficacité e l'algorithme parallèle, notée $e(n,p)$ par*

$$e(n,p) = \frac{\sigma(n,p)}{p} = \frac{T_1(p)}{pT_p(n)}. \tag{1.6}$$

□

Une accélération est dite *linéraire* lorsque $\sigma(n,p) = p$ (i.e. $e(n,p) = 1$). Elle est dite *sous-linéaire* lorsque $\sigma(n,p) < p$ (i.e. $e(n,p) < 1$). Enfin, elle est dite *sub-linéaire* lorsque $\sigma(n,p) > p$ (i.e. $e(n,p) > 1$). En pratique, on a presque toujours des accélérations sous-linéaires, l´accéleration linéaire étant celle recherchée dans la limite du raisonnable. Pour

ce qui est des accélérations sub-linéaires, elles peuvent se produire lorsque le schéma parallèle à d'autres effets bénéfiques telles que : la réduction des défauts de cache, des défauts de pages, ou des accès disque, l'augmentation du pouvoir de vectorisation par le partitionnement des opérations. Notons que, ceci intervient dans un contexte de mesure effective de performance, car c'est bien souvent à ce stade qu'interviennent les différents facteurs mentionnés. Toutefois, et de manière générale, les qualités de l'algorithme parallèle vont fortement dépendre des caractéristiques intrinsèques du problème sous-jacent et de la surface considérée. Cet aspect est quantifié entre autre par la *loi d'Amdhal* que nous rapellons.

La loi d'Amdhal. Considérons un algorithme séquentiel qui nécessite $T(n)$ instructions, ou encore un temps $T(n)$ (chaque instruction a un coût unitaire). Posons $T(n) = T_s(n) + T_{//}(n)$, où $T_s(n)$ est le coût de la partie intrinsèquement séquentielle de l'algorithme et $T_{//}(n)$ celui de la partie parallélisable. En écrivant $T_s(n) = f(n)T(n)$ et $T_{//}(n) = (f(n) - 1)T(n)$, on obtient une expression quantitative du principe d'Amdhal [65], qui stipule que le meilleur temps d'exécution sur p processeurs est donné par

$$T_p(n) = T_s(n) + \frac{T_{//}(n)}{p} = (f(n) - \frac{1 - f(n)}{p})T(n). \tag{1.7}$$

En remarquant que

$$\lim_{p \to \infty} T_p(n) = f(n)T(n) = T_s(n), \tag{1.8}$$

on comprend que le temps d'exécution parallèle ne peut pas être réduit à volonté en augmentant le nombre de processeurs. En d'autres termes, le temps d'exécution est borné, et cette borne correspond au coût de la partie séquentielle.

Les éléments de complexité que nous avons présentés font abstraction du phénomène de communication. En pratique, le temps d'exécution parallèle prend explicitement en compte le surplus de temps dû aux communications entre les processeurs. Cet aspect, qui est d'une importance capitale dans le comportement des machines parallèles à mémoire distribuée, ainsi que dans la conception les algorithmes et programmes parallèles destinés à de telles architectures, va maintenant retenir notre attention.

1.3 Etude des communications

Les communications entre les processeurs d'une machine parallèle constituent une activité très importante dans le déroulement d'un programme parallèle. Son étude a fait l'objet de nombreux travaux dont d'excellentes synthèses peuvent être consultées dans [129, 35, 34]. Ces travaux peuvent être classés en deux grandes catégories: ceux qui se préoccupent de l'aspect géométrique et combinatoire du réseau théorique (qui peut être statique ou dynamique), et ceux qui se préoccupent des enjeux réels des communications sur des architectures physiques. Ces deux aspects se suivent et se complètent, et permettent conjointement de mener à bien une analyse de performance d'un programme parallèle, aussi bien en termes de prédiction de performances que de mesures effectives suite à des expérimentations.

1.3.1 Aspect théorique

Il s'agit principalement d'étudier la topologie du graphe des connexions, c'est à dire les mécanismes combinatoires qui sont mis en jeu lorsque les processeurs doivent communiquer entre eux. En guise d'illustrtation, nous allons présenter quelques topologies courantes et étudier leurs caractéristiques.

a) Quelques topologies

Il s'agit essentiellement de la structure du graphe des connexions physiques. Parmi les propriétés interessantes recherchées, on retrouve : la *connexité* (indispensable), le *diamètre*, l'existence d'un *cycle hamiltonien*, les *degrés* des sommets, les coûts des plus courts chemins, le *nombre* et la *disposition* relative des arêtes. Pour une étude approfondie de ces détails et bien d'autres, voir [129].

a.1) Topologies linéaires.

Ce sont les topologies les plus simples, aussi bien du point de vue théorique que du point de vue de la mise en oeuvre. On y retrouve essentiellement les configurations linéaires et les anneaux (voir la figure 1.2).

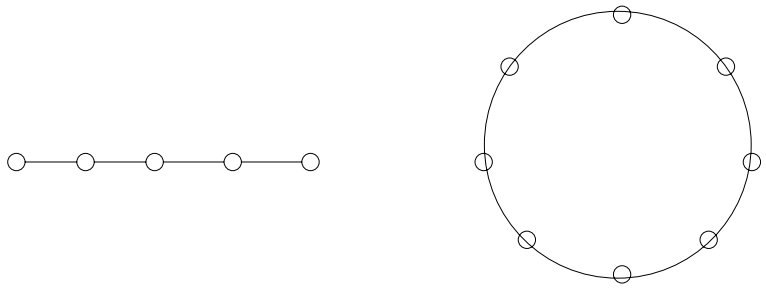


FIG. 1.2 – *Topologie linéaire et anneau.*

Considérons un réseau linéaire de p processeurs numérotés de 1 à p. Le nombre de transferts point-à-point nécessaires pour acheminer un message d'un processeur i vers un autre processeur j est au moins égal à $|i-j|$ (contre $\min(|i-j|, p-|i-j|)$ sur un anneau). Pour une diffusion optimale à partir d'un processeur, le coût moyen dans les deux cas est en $O(p)$. Dans certains cas, ce dernier coût peut être constant (le message est envoyé sur le bus et chaque processeur le lit au passage). Ceci est une des raisons du succès de cette topologie, mais le mécanisme requis est spécial par rapport au mécanisme général qui repose sur du point-à-point.

a.2) Les grilles.

Il s'agit d'une topologie rectangulaire dans laquelle chaque processeur est connecté à ses quatre ou deux voisins (voir figure 1.3). Une variante consiste à faire la boucle sur un des cotés ou sur les deux côtés (grille torique).

Le coût du transport d'un message d'un processeur à un autre est analogue au cas

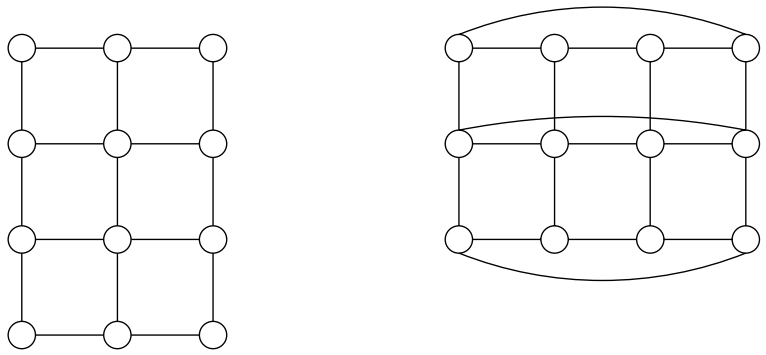


FIG. 1.3 – *Topologie en grille.*

précédent en suivant les axes. Pour la diffusion, le coût optimal est en $O(\sqrt{p})$ sur p processeurs.

Cette topologie et ses variantes sont assez utiliées sur les machines existantes du fait de leur mise en oeuvre naturelle. Par rapport aux topologies linéaires, on peut dire que le coût moyen des communications entre deux processeurs est meilleur sur une grille, et la connexité du graphe est plus stable (le graphe reste connexe même en éliminant certains liens). Cette proprété de *connexité robuste* est très importante en pratique pour optimiser le coût des communications (multiplicité des possibilités de routage et de communications simultanées), et aussi pour mieux gérer les transferts en évitant les interblocages.

a.3) L'hypercube

Définition 6 *Un graphe non orienté $G = (V, E)$ est appelé hypercube si elle vérifie la propriété recursive suivante:*
(i) $|V| = 1$ *et* $E = \emptyset$
(ii) Il existe une partition de V en deux sous-ensembles V_1 et V_2 de même cardinal, et une correspondance $\mu : V_1 \longmapsto V_2$ telle que

- *les sous-graphes induits par V_1 et V_2 sont des hypercubes.*
- *pour $x_1 \in V_1$ et $x_2 \in V_2$, $(x_1, x_2) \in E$ si et seulement si $x_2 = \mu(x_1)$.*

□

Il vient de cette définition que si $G = (V, E)$ est un *hypercube* alors il existe un entier n telle que $|V| = 2^n$; l'entier n est appelé *dimension* de l'hypercube. Ainsi, on peut appliquer un codage binaire des sommets. La correspondance mentionnée dans la définition associe alors les pairs de sommets dont les représentations ne différent que d'une composante. Analytiquement, un hypercube de dimension n, aussi appelé *n-cube*, est donc un graphe composé de 2^n sommets numérotés par les élements de $\{0, 1\}^n$, tel que deux sommets x et y sont adjacents si et seulement si $d(x, y) = 1$, où d est la distance définie comme étant égale au nombre de composantes distinctes (distance de *hamming*). La figure 1.4 fournit des configurations de *n-cubes* pour $n = 0, 1, 2, 3$. Considérant un ensemble de p processeurs organisés en hypercube, le coût de l'acheminement d'un mes-

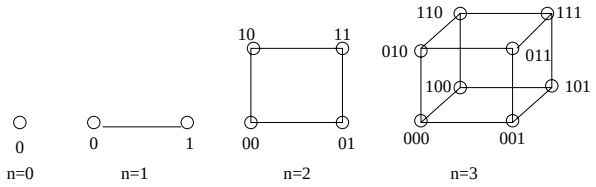


FIG. 1.4 – *Hypercubes de dimension* $0, 1, 2$ *et* 3.

sage entre deux processeurs est borné par $\theta(log(p))$, et le coût d'une diffusion otpimale est dans $\theta(log(p))$. Notons que ces coûts sont particulièrement intéressants au regard de la taille de la structure.

Les topologies en hypercube sont très utilisées, non seulement en raison de leur bonnes propriétés de communications, mais aussi à cause de leur régularité [99], et de leur capacité d'immersion dans d'autres formes de topolologies [87]. Quelques algorithmes parallèles dédiés à des architectures en hypercube [70, 136] peuvent être consultés dans [163, 144].

a.3) Graphes complets

Un graphe est dit *complet* lorsqu'il existe un lien direct entre chaque paire de sommets. Dans un tel graphe, un noeud envoie un message à n'importe quel autre noeud avec un coût unitaire, et la diffusion optimale est de l'ordre de $\theta(log(p))$. Toutefois, le maillage est très complexe (voire impossible à réaliser en pratique) d'un point de vue physique dès que le nombre de processeurs devient important. La figure Fig. 1.5 présente quelques graphes complets.

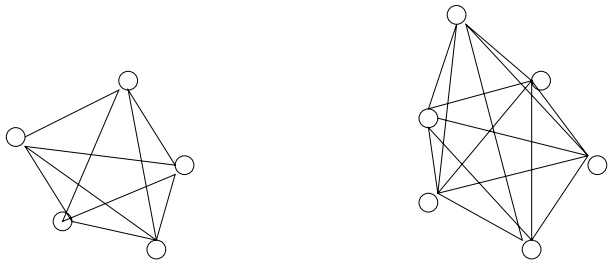


FIG. 1.5 – *Quelques graphes complets.*

a.4) Graphes de Cayley

Définition 7 *Soit* $(X, .)$ *un groupe et* S *un sous-ensemble de* X *ne contenant pas l'élément neutre, et qui est stable pour l'inversion. Le graphe de Cayley associé à* S *est le graphe* (X, E) *défini par* $E = \{(x, xs), x \in X, s \in S\}$. □

Ce graphe possède de nombreuses propriétés parmi lesquelles : *la connexité, la régularité, l'existence d'un cycle hamiltonien (conjecture) et la sommet-transitivité.* Des instances

classiques de graphes de Cayley sont : l'anneau, la grille torique, l'hypercube, et aussi le *graphe de pancake* ($S = \{(i,(i-1),\cdots,1,(i+1),(i+2),\cdots,n); i = 2,3,\cdots,n\}$ dans le groupe S_n des permutations d'ordre n), et le *graphe en étoile* (Toujours basé sur le groupe S_n des permutations d'ordre n), et où le sommet $x_1x_2\cdots x_n$ est relié au sommet $x_ix_2\cdots x_1x_{i+1}\cdots x_n$.

a.5) Graphes de De Bruijn

Définition 8 *Un graphe de De Bruijn est un graphe construit sur un ensemble D de mots de longueur d ($d \geq 2$), et dans lequel deux sommets sont reliés si et seulement si l'un est obtenu de l'autre par permutation circulaire, puis (éventuellement) remplacement de la première lettre par n'importe quelle autre lettre de D.* □

Le diamètre d'un tel graphe est évidemment égal à d, et le coût d'une diffusion optimale est dans $\Theta(log(|D|))$. On peut ainsi noter une certaine similarité avec l'hypercube du point de vue des paramètres, exception faite du degré qui est constant ici. La figure 1.6 présente un exemple de réseau de De Bruijn.

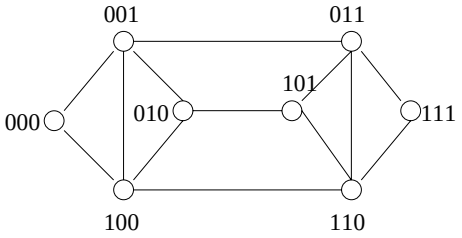


FIG. 1.6 – *Réseau de De Bruijn avec $d = 3$ sur l'alphabet $\{0,1\}$.*

a.6) Les arbres

Les topologies en arbres sont très répandues et jouent un rôle fondamental en informatique. Tout comme les hypercubes ou les graphes de De Bruijn, les topologies en arbre ont des propriétés très intéressantes telles que : le *faible voisinage*, le *faible diamètre*, et en plus la facilité de mise en oeuvre du fait de leur aspect planaire. La version la plus utilisée est celle des arbres binaires. La figure 1.7 présente quelques formes arborescentes.

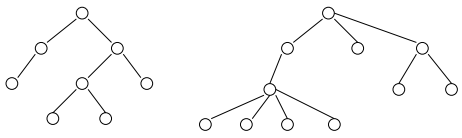


FIG. 1.7 – *Quelques formes arborescentes.*

a.7) Graphe butterfly.
Le graphe *butterfly* d'ordre n est le graphe dont l'ensemble de sommets est $\{0,1,\cdots,n-1\}\times\{0,1\}^n$, et tout sommet $(\ell, x_0x_1\cdots x_{n-1})$ est relié aux sommets $(\ell', x_0x_1\cdots x_\ell\cdots x_{n-1})$ et $(\ell', x_0x_1\cdots \bar{x_\ell}\cdots x_{n-1})$, où $\ell' = (\ell+1) \bmod n$. Le degré de chaque sommet est égal à 4 et que le diamètre du graphe est égal à $n + \lfloor \frac{n}{2} \rfloor$. La figure Fig. 1.8 présente un graphe *butterfly* d'ordre 3.

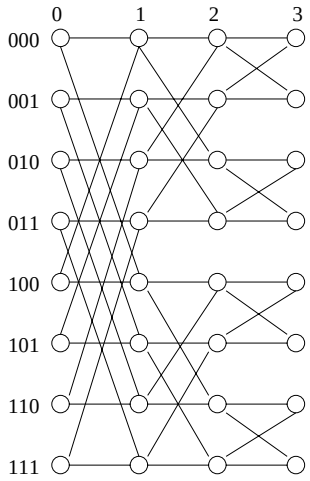


FIG. 1.8 – *Graphe butterfly d'ordre* 3.

Le lecteur pourrait consulter une étude récente des caractéritisques d'un réseau de type *Butterfly* dans [43].

Il existe plusieurs autres topologies : *araignée, caterpillar, cactus,...* [165]. Une étude de communication dans les graphes peut être consultée dans [84, 71]. Pour une vue générale sur les graphes et les résultats fondamentaux, se referer au célèbre ouvrage e Claude Berge [12].

b) Quelques résultats généraux
Nous nous intéressons ici à quelques résultats sur les graphes ayant une signification particulière sur les problèmes de communication [129].

Théorème 1 *Soit $G = (X, E)$ un graphe connexe d'ordre n. Si $\varphi(G)$ dénote le nombre minimum de sommets qu'on peut retirer pour que G perde sa connexité, alors on a :*

$$\varphi(G) \leq \min_{x\in X} d_G(x), \tag{1.9}$$

où d_G est la fonction qui donne le degré d'un sommet. □

Ce résultat montre que, bien que la propriété de *faible degré* soit matériellement intéressante du fait que les connexions sont moins complexes, elle a l'inconvénient de rendre

plus *sensible* la connectivité du graphe. Dans une architecture réelle, la "suppression" d'un processeur peut tout simplement traduire le fait qu'il soit occupé (ou défectueux en pire cas). Dans une telle situation, le message peut être mis en attente, ce qui ralentit les performances du programme, lesquelles s'éloignent par ailleurs des éventuelles prédictions théoriques. Certaines architectures se servent d'une couche système spéciale pour contourner le problème mais sans annuler complétement son effet retardateur, car il y a une intervention logicielle supplémentaire.

Théorème 2 *Si G est un graphe orienté d'ordre N, de diamètre D, et de demi-degré intérieur et extérieur au plus égal à d, alors on a :*

$$N \leq 1 + d + d^2 + \cdots + d^D. \tag{1.10}$$

□

Cette borne, appelée *borne de Moore orientée* n'est atteinte que dans le cas trivial où $d = 1$ et $D = 1$. La conjecture de J-C. Bermond stipule que cette borne reste valable si sur chaque sommet, la somme du degré extérieur et du degré intérieur est bornée par $2d$. Dans tous les cas, il semble bien qu'on ne puisse pas augmenter à volonté la taille d'un graphe connexe sans en augmenter le diamètre ou les degrés des sommets. Dans un réseau d'interconnexion, ceci traduit le fait que si le nombre de processeurs augmente, alors soit les distances deviennent plus importantes, soit les connexions locales se compliquent.

Ces résultats montrent bien les limites à prendre en compte si on veut bâtir un réseau ayant les qualités voulues. Ceci constitue la difficulté majeure dans la conception des machines parallèles à très grand nombre de processeurs.

Sur une machine parallèle, la prise en compte de la topologie dans les mesures n'est pas aussi poussée qu'en théorie, ce qui rendrait très compliquée toute analyse ou prédiction de performances. Ceci est légitimée en quelque sorte par le fait que, dans les machines parallèles à haute performance, la vitesse des transferts nous approche raisonnablement vers une configuration de graphe complet.

1.3.2 Mesure des communications

La mesure des communications sur une machine parallèle est une tâche très importante dans l'analyse des performances de programmes. Des études approfondies sur les mesures de coût des communications peuvent être consultées dans [56, 129, 131].

D'une manière générale, le coût de la communication d'un message de volume L entre deux processeurs situés à une distance d est de la forme

$$d(\alpha + L\tau), \tag{1.11}$$

dans le mode *store-and-forward* (à chaque passage par un noeud, le même mécanisme est repris). Dans le mode *wormhole*, les passages intermédiares mettent en jeu d'autres mécanismes plus légers, et le coût de l'acheminement est alors de la forme

$$\alpha + (d-1)\delta + L\tau. \tag{1.12}$$

Dans les deux cas,

- α représente le coût de la préparation du message, de l'appel de la routine de communication et des actions de synchronisation, il est communément appelé *start-up*.
- δ correspond au délai de passage par chaque noeud intermédiare.
- τ repŕesente la *vitesse* du transfert (la quantité $\frac{1}{\tau}$ est appelée *bande passante*).

Dans le cas d'un envoi direct (de voisin à voisin), on prend $d = 1$ et on obtient alors la relation $\alpha + L\tau$. A ce niveau, on distingue les cas $\alpha = 0 \wedge \tau = 0$ (hypothèse du *systolique*), $\alpha = 0 \wedge \tau \neq 0$ (pas de *start-up*), et $\alpha \neq 0 \wedge \tau = 0$ (tout se passe au niveau du *start-up*).

Une approche usuelle pour évaluer le coût des communications lorsque les paquets ont la même taille L consiste à comptabiliser le nombre de transferts (cette activité est purement combinatoire), et multiplier ensuite cette quantité par le facteur $\alpha + L\tau$.

Le modèle de coût d'un programme parallèle sera donc assez souvent de la forme

$$T_{//} = \frac{T_{seq}}{p} + [(\alpha + L\tau)\varphi(p)], \tag{1.13}$$

où $\varphi(p)$ représente le nombre d'envois de messages, obtenu en supposant que le graphe de communication sous-jacent est un graphe complet. Bien sûr ce modèle est trop simpliste mais on peut s'en contenter dans certains cas, surtout qu'il a l'avantage de ne pas être lié à une topologie spécifique (d'où une certaine robustesse de la modélisation). Toutefois, si on peut faire mieux dans une situation bien précise, il serait appréciable de s'y investir. Dans cette optique, un modèle assez courant est donné sous la forme

$$T_{//} = \frac{T_{seq}}{p} + [\psi(p)\alpha + \varphi(p)L\tau], \tag{1.14}$$

où $\psi(p)$ représente le nombre total d'envois de message à partir de la source, et $\varphi(p)$ le nombre total de communications point-à-point.

A tout ceci, il faut ajouter les phénomènes très variables tels que les routages dynamiques, l'état de la mémoire à un moment donné, et bien d'autres mécanismes mis en jeu dans le processus global de gestion des données. On comprend donc pourquoi concevoir un algorithme parallèle efficace est une tâche ardue.

1.4 Techniques de conception d'algorithmes parallèles

La conception d'un algorithme parallèle peut se faire de plusieurs manières. Soit on applique un paradigme approprié sur l'ensemble des opérations effectuées par le traitement considéré, soit on se sert d'une méthodologie existante que l'on applique sur une spécification du problème à résoudre. La première approche requiert une certaine dose d'ingéniosité, car même si les paradigmes de la parallélisation sont parfois simples dans leur description, il n'en demeure pas moins que leur application n'est pas toujours immédiate. La deuxième approche quant à elle est assez souvent "mécanique", surtout lorsque le contexte se prête bien aux hypothèses de la méthodologie considérée.

1.4.1 Les Paradigmes

Nous passons en revue quelques paradigmes permettant de concevoir des algorithmes paallèles.

a) Le pipeline.

Nous allons baser notre explication sur le *pipeline linéaire*, les autres types de pipeline n'en étant que des extensions sur des dimensions supplémentaires. Considérons une tâche S pouvant se décomposer en un ensemble de n tâches identiques $S_1, S_2, \cdots, S_n$. Supposons ensuite que chacune des tâches S_i puisse se décomposer en une séquence d'opérations $S_{i1}, S_{i2}, \cdots, S_{im}$ à exécuter dans cet ordre. Notre but est de pouvoir réaliser la tâche S sur m processeurs $p_1, p_2, \cdots, p_m$, sachant que chaque processeur p_j est spécialisé dans l'exécution de S_{*j}. La figure Fig.1.9 illustre la stratégie de l'ordonnancement pour $n = 4$ et $m = 5$. Chaque processeur exécute les sous-tâches situées sur la colonne correspondante, en communiquant au processeur suivant les informations nécessaires à l"ccomplissement de la tâche globale impliquée.

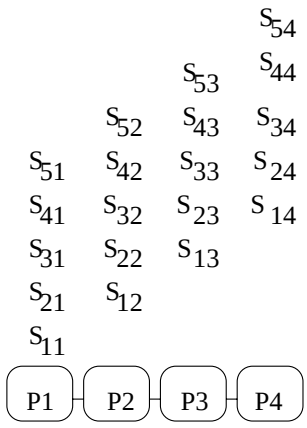


FIG. 1.9 – *Exécution en pipeline.*

Chaque tâche S_i subit un délai nécessaire pour assurer l'ordre $S_{i1}, S_{i2}, \cdots, S_{im}$. Dans la réalité, cet ordre réflète une dépendance de données, d'où le lien entre les processeurs voisins. Si on suppose que l'exécution d'une tâche élémentaire prend un temps unitaire, alors le temps total d'exécution de S sur les m processeurs est donné par

$$T_n = m + n, \tag{1.15}$$

soit une efficacité égale à

$$e_n = \frac{n}{n+m}. \tag{1.16}$$

On comprend donc que la technique du pipeline est d'autant plus efficace lorsque n est grand, c'est à dire lorsque le nombre de tâches à pipeliner est important. L'entier m est appelé *profondeur* du pipeline. La technique du pipeline est la base des techniques de

vectorisation (pipeline au niveau instruction), et d'une manière plus générale, elle est intensément utilisée dans les approches *systoliques*.

b) La technique diviser pour paralléliser.
C'est une technique analogue à l'approche *Diviser Pour Régner* qu'on retrouve dans l'agorithmique séquentielle. Elle consiste à construire progressivement la solution d'un problème à partir d'instances indépendantes (donc, pouvant être traitées en parallèle) et de taille plus petite (on parle souvent de *sous-problèmes*). Cette technique peut se décrire ainsi qu'il suit:

1. Décomposition du problème en des instances indépendantes et de taille réduite.
2. Résolution parallèle des sous-instances en appliquant récursivement la même technique.
3. Construction de la solution du problème initial à partir des solutions des sous-problèmes.

En guise d'illustration, considérons l'exemple du tri d'un vecteur de taille n. On adopte la démarche suivante:

1. Découper le vecteur en deux parties égales $u(1 \cdots \frac{n}{2})$ et $u(\frac{n}{2} + 1 \cdots n)$.
2. Trier par le même principe et en parallèle chacun de ces sous-vecteurs.
3. Fusionner les deux sous-vecteurs triés pour en faire un vecteur trié..

Si la fusion est faite en séquentiel, alors le temps d'exécution de l'algorithme parallèle sera de l'ordre de $2n$ avec n processeurs. Par ailleurs, si on considère un découpage cyclique au lieu d'un découpage en bloc, on retrouve un algorithme connu sous le nom de *odd-even sort*.

c) L'augmentation du parallélisme par la redondance.
Intuitivement, il s'agit de limiter la factorisation des opérations de manière à obtenir plus de tâches indépendantes. Il est évident que les algorithmes parallèles obtenus dans ce contexte ne sont pas optimaux (puisqu'on a ajouté des opérations en plus de celle du schéma séquentiel de base). Prenons l'exemple du schéma de Horner sur un polynôme $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$. L'évaluation séquentielle optimale est en $O(n)$ par la récurrence $y_0 = 0$; $y_{i+1} = x y_i + a_i$. Ce schéma est intrinsèquement séquentiel et donc difficile à paralléliser de manière directe. Si on dispose par exemple de deux processeurs, on peut écrire $p(x) = [a_n x^n + a_{n-1} x^{n-1} + \cdots + a_{\frac{n}{2}} x^{\frac{n}{2}}] + [a_{\frac{n}{2}+1} x^{\frac{n}{2}+1} + \cdots + a_1 x + a_0]$, qu'on peut mettre sous la forme $p(x) = x^{\frac{n}{2}} u(x) + v(x)$ où $u(x)$ et $v(x)$ sont des polynônes de degré $\frac{n}{2}$. En évaluant indépendamment $u(x)$ et $v(x)$ sur les deux processeurs, on obtient un temps d'exécution parallèle de l'ordre de $\frac{n}{2} + log(\frac{n}{2})$ ($log(\frac{n}{2})$ représente le coût de l'évaluation de $x^{\frac{n}{2}}$). Remarquons que si on reste dans cette optique et qu'on applique la technique *Diviser Pour Paralléliser*, on obtient un algorithme en $O(log(n))$ sur n processeurs. Comme nous l'avons souligné, l'agorithme n'est pas optimal car son travail est en $O(nlog(n))$ au lieu de $O(n)$, mais on peut noter l'importante réduction du temps de calcul.

d) Le systolique.
Ce concept a été motivé en grande partie par l'importante avancée technologique en matière d'intégration des circuits. Introduit en 1978 par Kung et Leiserson [81], l'idée principale du *systolique* repose sur la recherche d'une solution parallèle sous forme de circuit obtenu par adjonction de cellules identiques. Les cellules sont spécialisées et les transferts de données sont supposés avoir un coût nul, car elles sont faites à travers des signaux électriques.

Une architecture *systolique* [114] est une réseau composé de cellules élémentaires identiques et localement interconnectées. Chaque cellule reçoit des données en provenance des cellules voisines, effectue un calcul élémentaire, puis transmet les résultats à des cellules voisinnes un cycle plus tard. Seules les cellules situées sur les bords du réseau communiquent avec l'extérieur, c'est à dire une mémoire externe ou une machine hôte.

Cette voie a donné lieu à d'abondants travaux centrés sur différents aspects tels que : l'algorithmique [114, 40], les architectures[80, 95], les méthodologies de conception d'e réseaux systoliques [92, 113, 116, 132], les environnements de conception, de simulation et de vérification [156].

1.4.2 Les techniques d'ordonnancement

Il s'agit essentiellement des techniques qui permettent de construire des fonctions de temps et des stratégies d'allocation. Pour chacun de ses paramètres, il existe d'une part des résultats généraux qui donnent une idée de leur importance et de leur complexité, et d'autre part des méthodes de détermination qui se distinguent les unes des autres par le contexte d'application et la qualité des résulats.

Fonction de temps.
Considérons une représentation des tâches sous forme de graphe orienté $G = (X, A)$, où X modélise les tâches, et A les relations de dépendances.

Définition 9 *Une application $t : X \to N$ définit une fonction de temps compatible avec le graphe $G = (X, A)$ si elle vérifie la propriété*

$$\forall x, y \in X : [(x, y) \in A] \Longrightarrow [t(x) < t(y)]. \tag{1.17}$$

□

Avec une telle fonction, une tâche x est exécutée à l'instant $t(x)$. L'algorithme sera séquentiel si la fonction t est *injective*. Dans tous les cas, le temps global d'exécution, notée $\Theta(t, G)$, est donnée par

$$\mathcal{T}_{//}(t, G) = \max_{x \in X} t(x) - \min_{x \in X} t(x) + 1. \tag{1.18}$$

Etant donné qu'un graphe de précédence G n'a pas de cycle (*acyclique*), son *diamètre*, noté $\gamma(G)$, est bien défini et est égal à la longueur d'un plus long chemin dans G. On a alors le résultat suivant qui se vérifie aisément.

Théorème 3 *Si t est une fonction de temps définie sur un graphe $G = (X, A)$, on a*

$$\mathcal{T}_{//}(t, G) \leq \gamma(G). \tag{1.19}$$

□

Cette borne est générale, car elle reste valable même dans le cas des graphes valués et pondérés. Il existe des techniques bien connues qui permettent de construire des fonctions de temps qui atteignent cette borne, à l'exemple des méthodes du *chemin critique* [46].

Plusieurs méthodes ont été proposées dans la littérature pour construire des fonctions de temps. Pour la plupart, elles reposent sur des transformations analytiques directes (parfois à support géométrique), suivies éventuellement de la résolution de programmes linéaires [75, 113, 93, 145, 116] (voir [38, 135, 47] pour des notions sur la programmtion linéaire). Il existe aussi des méthodes combinatoires [145, 46, 19, 132].

Stratégie d'allocation.

Définition 10 *Soit $G = (X, E)$ un graphe orienté et t une fonction de temps compatible avec G. Une fonction $A : X \to D$, où D est un domaine discret quelconque, définit une fonction d'allocation sur G compatible avec la fonction de temps t, si elle vérifie la condition*

$$\forall x, y \in X, [t(x) = t(y)] \Longrightarrow [A(x) \neq A(y)]. \tag{1.20}$$

□

Le lecteur aura remarqué le lien entre la fonction d'allocation et la fonction de temps. En fait, ce lien traduit celui de la fonction d'allocation avec les dépendances entre les tâches dans une approche purement combinatoire. Bien que la condition (1.20) soit théoriquement suffisante, d'autres critères sont à prendre en compte de par leur implication sur la *localité*, la *régularité*, le coût global des *communications*, l'*équilibre des charges*, et la *modularité* de l'architecture induite.

Avec une fonction d'allocation A, la *surface* de l'algorithme, notée $S(A, G)$, est trivialement donnée par

$$S(A, G) = |A(X)|, \tag{1.21}$$

où $|.|$ est la fonction cardinalité. Pour une fonction de temps donnée, il existe en général plusieurs fonctions d'allocation possibles. Un des problèmes classiques consiste donc à en rechercher une allocation qui induit un nombre minimal de processeurs.

Pour ce qui est des communications, remarquons qu'elles interviennent chaque fois que l'on a deux tâches $x, y \in X$ telles que

$$[(x, y) \in V] \wedge [A(x) \neq A(y)]. \tag{1.22}$$

Dans ce cas, une communication est nécessaire pour transférer les données liées à x, de $A(x)$ vers $A(y)$. Cette communication doit intervenir à un instant situé entre $t(x)$ et $t(y) - 1$, le choix du bon moment étant lui aussi un problème pratique important (parfois superflu dans un cadre théorique). Dans tous les cas, on peut considérer que le volume des communications est donné par

$$C(t, A, G) = |(x, y) \in V : [(x, y) \in V] \wedge [A(x) \neq A(y)]|. \tag{1.23}$$

Un autre problème combinatoire qui en découle est celui de la recherche d'une allocation A qui minimise $C(t, A, G)$.

On comprend donc que la recherche d'une allocation optimale est un problème combinatoire difficile [45, 155]. Le lecteur trouvera aussi des stratégies d'allocation dans les différentes références mentionnées dans la section précédente sur la fonction de temps.

D'une manière générale, le problème de l'ordonnancement parallèle optimal peut être posé comme suit [45]:

(P1) Problème général de l'ordonnancement

Instance: Un ensemble T de n tâches, un ordre partiel $<$ sur T, des poids A_i, $1 \leq i \leq n$, m processeurs, et une borne de temps k.

Question: Peut-on trouver une fonction totale $\tau : T \rightarrow \{1, 2, \cdots, k\}$ telle que:

(i) si $i < j$ alors $\tau(i) + A_i \leq \tau(j)$

(ii) $\forall i \in T$, $\tau(i) + A_i \leq k$

(iii) $\forall t, 1 \leq t \leq k$, $|i \in T, \tau(i) \leq t \leq \tau(i) + A_i| \leq m$.

Ce problème a été prouvé NP-complet [74]. Le problème reste NP-complet même dans les cas restreints suivants:

- $A_i = 1$, $i = 1, \cdots, n$ [155]
- $m = 2$, et $A_i \in \{1, 2\}$, $i = 1, \cdots, n$ [155]

A côté de cela, il faudrait aussi noter l'existence des allocations dynamiques, techniques parfois intéressantes lorsqu'il faut faire face à des problèmes de pannes, ou lorsqu'on souhaiterait maintenir un certain équilibre des charges dans un contexte où très peu d'informations sont statiquement disponibles.

1.5 Parallélisation automatique

La parallélisation automatique est une discipline qui rassemble les techniques permettant de produire un programme parallèle à partir d'un programme séquentiel [48, 7, 85, 39]. Ces techniques reposent sur une analyse préalable des dépendances [6, 111] entre les instructions du programme (il s'agit généralement des nids de boucles), suivie éventuellement d'une série de transformations valides visant à influer sur le potentiel et le type du parallélisme apparaissant dans la structure considérée. Parmi les transformations existantes (voir [61]), on peut citer:

L'échange. C'est une technique qui consiste à interchanger l'ordre des boucles imbriquées afin d'aboutir à une configuration dans laquelle la boucle la plus externe (ou la plus interne) est parallélisable, en fonction de la granularité souhaitée.

La fusion. Elle consiste à fusionner plusieurs boucles imbriquées pour n'en obtenir qu'une seule, ceci afin de réduire la dimension de l'espace d'itération et obtenir ainsi un parallélisme plus important selon une direction.

La distribution. C'est l'opération inverse de la *fusion*. Elle est très utile lorsqu'on veut séparer la boucle séquentielle et la boucle parallélisable. Elle peut des fois per-

mettre de simplifier les dépendances, de produire des boucles régulières, ou encore de mieux gérer la mémoire.

La torsion. C'est une transformation qui affecte uniquement les indices des boucles. Elle permet de transformer les dépendances et l'espace d'itération afin d'obtenir de nouvelles boucles plus propices à une bonne parallélisation.

Une vue plus détaillée des techniques de transformation peut êtres consultées dans [167, 28, 39]. Ces différentes techniques sont actuellement très utilisées dans le but de produire des compilateurs paralléliseurs tels celui de HPF, ou des environnements de manipulation et de transformation de programmes tels que Omega de William Pugh [100], PAF de Feautrier [49], Loopo de Lengauer [66], ou encore PIPS de Irigoin [107]. Notons que même dans un cadre séquentiel, l'application de ces transformations de boucles peut influer considérablement sur le temps d'exécution, principalement par le biais de l'impact sur les problèmes de cache et de vectorisation. Il s'agit donc des techniques globalement utiles et efficaces.

1.6 Autour de l'implémentation

L'implémentation d'algorithmes parallèles sur des architectures réelles requiert la connaissance d'un langage parallèle et des bibliothèques de gestion des tâches et des communications. Toutefois, il faut noter que la portabilité des programmes parallèles est un aspect assez délicat, ce qui justifie qu'il faille dans chaque contexte recompiler le code source. Dans des cas moins fréquents, on est amené à retoucher quelque peu le programme pour lever certaines incompatibilités et effectuer ses propres optimisations. Globalement, il faut disposer à la base d'un compilateur et des bibliothèques appropriées.

1.6.1 Les langages parallèles

1.6.1.1 Langages standards

Parmi les langages standards utilisés dans la programmation parallèle, on peut citer : le FORTRAN (77 et 90), le C, JAVA, LISP. Le plus ancien de ces langages est sans doute le FORTRAN 77, dont une extension a conduit au FORTRAN 90. En général, ces langages sont couplés à des biblilothèques de communication (MPI, PVM, BLACS), et éventuellement des routines de calculs spécifiques (ScaLAPACK pour les opérations d'algèbre linéaire). Dans certains cas, le parallélisme peut être implicite pour l'utilisateur, il revient alors au compilateur la charge d'expliciter l'ordonnancement et l'exécution parallèle du code. Dans cette direction, on a l'exemple remarquable de HPF (*High Performance Fortran*).

1.6.1.2 HPF

Un programme HPF peut être vu comme étant un programme FORTRAN 90, auquel on a ajouté des primitives et directives spéciales destinées à la parallélisation du code. Grâce à cette extension, l'utilisateur peut exprimer son parallélisme à un haut niveau (les aménagements nécessaires étant à la charge du compilateur). L'utilisateur peut entre

autres spécifier: une topologie virtuelle de processeurs, une distribution des données, une distribution des boucles, l'autorisation ou non d'effectuer une parallélisation locale, des opérations de réduction, et bien d'autres primitives destinées à orienter les actions du compilateur-paralléliseur.

Il est évident qu'il ne faut envisager l'usage efficace de HPF que dans des cas de programmes ayant une structure régulière. Cette régularité peut être liée au problème sous-jacent, ou alors résulter d'un ensemble de transformations valides appliquées à une formulation directe du problème considéré. Pour ce qui est des distributions, elle peuvent être de trois principales natures: *bloc*, *cyclique*, et *bloc-cyclique*. En l'absence de ces directives, l'utilisateur peut laisser le soin au compilateur d'effectuer une distribution quelconque. Cette dernière situation serait idéale pour l'utilisateur, mais elle comporte des dangers tels que *l'incohérence* des résultats due à gestion incorrecte des dépendances, ou une sévère *inefficacité* due à un surcout excessif des communications (résultant d'une mauvaise adéquation entre le partage des données et le partage des tâches). Le lecteur intéressé pourrait consulter [78].

D'une manière générale, l'évolution de HPF dépendra des résulats obtenus dans les domaines de la parallélisation automatique et de l'optimisation de code. Toutefois, la programmation parallèle explicite, c'est à dire celle dans laquelle l'utilisateur est le principal responsable de l'organisation des opérations, reste celle qui peut plus généralement conduire aux gains de performances souhaités.

1.6.2 Les bibliothèques de communications

1.6.2.1 PVM

PVM (Parallel Virtual Machine) [60] est une bibliothèque pour la gestion des processus et communication dans le cas des réseaux de stations ou en environnement hétérogène. De ce fait, on a une forte portabilité sur les programmes écrit avec PVM, surtout au niveau des systèmes des architectures cibles. En plus des opérations classiques de communication que celles de MPI, il permet la gestion dynamique des processus. Toutefois, MPI a connu des développements plus importants, mais pour un utilisateur, le choix reste lié au contexte.

1.6.2.2 MPI

MPI (Message Passing Interface) [140] est une bibliothèque standard pour la gestion des communications dans les machines parallèles à mémoire distribuée. On y retrouve des routines de communication point-à-point, des communications globales (diffusion, collecte), et des opérations de réduction (max, min, somme). En fonction des caractéristiques de sa machine, un constructeur essayera d'optimiser autant que possible les primitives MPI. Puisque ces optimisations ne sont effectuées que lors de la compilation, cet aspect n'a pas d'influence sur la portabilité des programmes MPI.

1.6.2.3 OpenMP

OpenMP est une bibliothèque standard pour la gestion du parallélisme sur machines à mémoire partagée. Tout comme avec MPI, il est couplé à un langage de programmation (OpenMP Fortran), et fournit un ensemble de directives et de routines permettant de

specifier et de gérer des exécutions parallèles. Pour une vue complète sur OpenMP, le lecteur pourrait consulter [101].

1.7 Quelques machines parallèles

Nous donnons une vue de quelques machines parallèles existantes. Les deux premières sont relativement anciennes. Quant aux suivantes, elles sont assez récentes et actuellement très utilisées. Le lecteur particulièrement intéressé pourrait consulter le rapport TOP500 [42].

1.7.1 La Thinking Machine CM-5

La *Thinking Machine CM-5* est une machine parallèle à mémoire distribuée dont la topologie sous-jacente est celle d'un arbre ou d'un hypercube. Elle est composée de 16 à 16384 processeurs processeurs vectoriels de profondeur 4 avec une performance de 128 MFlops. La taille de la mémoire locale de chaque processeur est de 32 Mo. Les communications s'effectuent avec une vitesse de transfert de l'ordre de 20 Mo/s. Le système d'exploitation est CMost (Unix), et on retrouve les compilateurs du fortran, du C, et du Lisp.

FIG. 1.10 – *La* THINKING MACHINE CM-5.

1.7.2 La machine nCUBE2

La nCube2 est une machine à mémoire distribuée dont la topologie sous-jacente est celle de l'hypercube. Elle est composée de 8 à 8192 processeurs, cadencés à environ 3

MFlops et ayant chacun une mémoire locale de 64Mo. La mode de communication est le *wormwhole*, avec une vitesse de transfert de 2.75 Mo/s. Le systeme d'exploitation est SunOS et on y retrouve les compilateurs du fortran et du C.

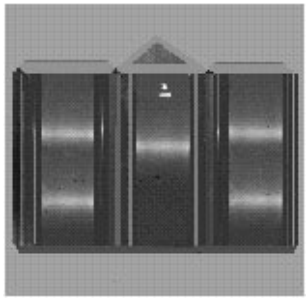


FIG. 1.11 – *La nCube2.*

1.7.3 La machine CRAY T3E

La CRAY T3E est une machine parallèle à mémoire partagée dont la topologie sous-jacente est une grille torique. Elle est composée de 256 processeurs (dont 7 pour le système d'exploitation) à 600 MFlops/s. La mémoire totale est de 128 Mo avec un mode d'adressage local et global. Les communications s'effectuent à une vitesse de transfert de 480 Mo/s. Le système d'exploitation est Unicos/mk, et on retrouve les compilateurs du fortran et du C, avec les librairies MPI et PVM.

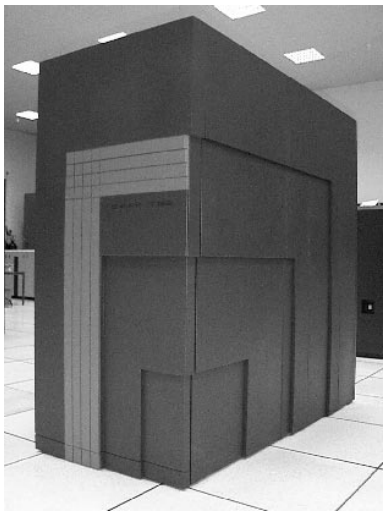


FIG. 1.12 – *La machine CRAY T3E.*

1.7.4 La machine IBM SP2

La IBM SP2 une machine à mémoire distribuée de type *cluster*, dont la topologie sous-jacente est une grille. Elle est composée de 8 à 128 processeurs, cadencés à environ 266 MFlops et ayant chacun une mémoire locale de 2 Go. Les communications s'effectuent avec une vitesse de transfert de l'ordre de 20 Mo/s . Le systeme d'exploitation est AIX (Unix) et on y retrouve les compilateurs du fortran, du C et du XL.

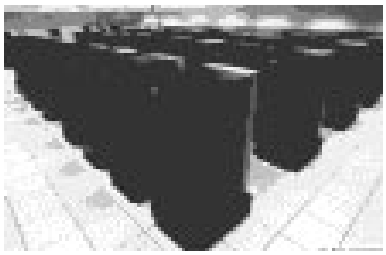


FIG. 1.13 – *La machine IBM SP2.*

1.7.5 La machine ORIGIN2000

La machine ORIGIN2000 du CINES (Centre Informatique National de l'Enseignement Supérieur) est une machine parallèle à mémoire distribuée, constituée de 256 processeurs R12000 cadensés à 300Mhz. Elle dispose d'un total de 80 Go de mémoire, et 432 Go de disque. Sa puissance théorique totale est de 154 Gflops.

FIG. 1.14 – *La machine ORIGIN2000.*

1.7.6 La machine Fujistu VPP500

La VPP500 est une machine parallèle composée de 222 processeurs *vectoriels* ayant chacun une puissance théoriquement égale à 1.6 GFlops et une mémoire locale de taille égale à 1 Go. Elle dispose en plus de 2 processeurs pour la gestion du système. La

topologie sous-jacente est celle d'une grille 2D, et les communications peuvent s'effectuer en même tant que les opérations flottantes, avec une vitesse de l'ordre de 800 Mo/s et une latence relativement faible.

FIG. 1.15 – *La machine VPP500.*

1.8 Vision sur le présent et le futur

Il semble clair que le *parallélisme* constitue actuellement un thème central dans l'ensemble des activités liées à l'informatique. Aujourd'hui, nombreux sont les sites industriels, unités de recherches, centres de ressources informatiques, services de prévision et de simulation, et autres centres dont les activités impliquent des traitements quantitativement importants, qui disposent de véritables machines parallèles dont ils se servent à profit. De ce fait, la mise au point des architectures parallèles est devenue une activité priviligiée chez bon nombre de constructeurs. Toutefois, on ne peut raisonnablement pas affirmer que les machines parallèles soient du domaine du grand public, à cause sans doute de leur coût relativement important et de la difficulté d'une standardisation du point de vue des architectures et des systèmes.

D'un point de vue philosophique, le débat sur les limites des performances des calculateurs, que ce soit de facon absolue ou par rapport aux capacités humaines, semble relancé à cause des performances théoriquement envisageables des algorithmes parallèles, et des victoires historiques telles que celle de la machine Deep Blue sur Garry Kasparov à la suite une partie d'échec ayant eu lieu le 11 mai 1997. Lorsqu'on sait que cette machine parallèle était capable d'examiner 200 millions de positions par seconde, et qu'en plus les techniques heuristiques mises en oeuvre étaient assez sophistiquées pour n'en examiner que les coups potentiellement intéressants, on a alors une idée de la puissance globale de calcul qui était mise en jeu. Des faits de même nature sont aussi à chercher dans les activités liées à la cryptographie.

Le développement spectaculaire des capacités d'intégration et techniques annexes laisse envisager un essor futur important des systèmes embarqués, et plus généralement des architectures spécialisées. A côté des approches standards de programmation parallèle, il faudra compter avec l'apport des idées du *MetaComputing*, et sur un tout autre chapitre, celui du *Calcul quantique*.

Il semble en tout cas qu'on est embarqué dans une aventure aux enjeux très importants, et dans laquelle on peut noter un sérieux investissement humain et institutionnel. On est donc en droit de maintenir nos rêves...éveillé!

Chapitre 2

Ordonnancement canonique

Ce chapitre présente une technique d'ordonnancement parallèle que nous avons publiée dans [145] et [146]. Par rapport aux textes originaux, des aménagements ont été nécessaires pour mieux le situer par rapport à l'ensemble de ce document.

2.1 Résumé

Nous proposons une méthodologie de conception d'ordonnancements parallèles réguliers. La technique s'applique aussi bien sur un système d'équations de récurrence que sur un graphe de dépendance. Le principe de base repose sur la possibilité de construire l'espace global de calcul à partir d'un sous-espace générique. Techniquement, il s'agit de partir d'un ordonnancement local de la struture générique pour dériver un ordonnancement complet par des reproductions successives. De plus, cette construction se fait systématiquement dès lors que tous les paramètres ont pu être identifiés. Les exemples du *chemin algébrique* et de la *factorisation de Cholesky* sont étudiés en guise d'illustration.

2.2 Introduction

D'une manière générale, trouver un ordre d'exécution des tâches et une répartition de celles-ci sur plusieurs processeurs constitue un des problèmes fondamentaux du parallélisme. A la base, le modèle considéré est celui des graphes orientés (éventuellement valués et pondérés). Par la suite, le mode d'expression des calculs sous forme de systèmes d'équations récurrentes [75] est apparu et a conduit à l'émergence des approches analytiques, pouvant par ailleurs faire l'objet de manipulations syntaxiques *systématiques* ou *semi-automatiques* [53, 115, 117, 132]. Toutefois, ces méthodologies fournissent en général une solution dont le partitionnement (en cas de ressources matérielles insuffisantes) est une opération supplémentaire, pouvant nécessiter une importante modification de la logique des processeurs, et même parfois conduire à une perte d'efficacité [119]. Ceci est sans doute dû au fait que dans la majorité des cas, l'on ne s'intéresse qu'à la nature des dépendances, et le problème de ressources n'est quant à lui analysé qu'à posteriori.

Nous proposons une méthode qui s'appuie le moins possible sur la nature des dé-

pendances, et dérive systématiquement un ordonnancement dès lors que les hypothèses sur la structure globale du graphe sont satisfaites. De plus, la quantité de resources disponible peût être prise en compte plus tôt, et à défaut, l'algorithme obtenu peût être trivialement adaptaté pour fonctionner un nombre réduit de processeurs.

En effet, certains schémas de calcul peuvent se décomposer en une cascade de sous-structures similaires et localement dépendantes. Un paradigme naturel pour paralléliser ce type d'algorithme est le pipeline. Dans cette optique, le pipeline peut se faire soit à l'intérieur de chaque subdivision, soit entre les subdivisions elles-mêmes. Dans cette deuxième approche qui correspond à celle qui nous considérons dans ce travail, l'ordonnancement parallèle global est obtenu en appliquant une composition de transformations locales partant d'une solution partielle. Il apparait alors que la qualité du résultat obtenu dépendra de l'adéquation entre l'ordonnancement générique et les paramètres de sa reproduction sur le graphe tout entier. Dans le cadre séquentiel, cette idée est bien développée dans le paradigme de la programmation dynamique. Par contre, dans le cas parallèle, à l'exception des schémas intrinsèquement réguliers (schémas uniformes ou affines), la synthèse d'algorithme se fait au cas par cas selon le problème considéré et par rapport au type d'architecture recherché ou encore en fonction de la complexité à atteindre.

Notre approche est donc générale et systématique. Par ailleurs, la nature de la solution obtenue met en évidence deux principaux avantages qui sont la tolérence aux pannes (par simple décalage), et le partitionnement direct (par des passes multiples). Ajoutons à cela le fait que, la nature des dépendances n'étant pas un des facteurs de nos hypothèses, le champ d'action de notre méthode est à priori beaucoup moins restrictif que d'habitude.

2.3 Equations récurrentes et reproduction canonique

2.3.1 Equations récurrentes

Définition 11 *Une* ***équation récurrente*** *définissant une fonction (ou une variable) X en tous les points z d'un domaine D, est une équation de la forme*

$$X(z) = D^X \quad : \quad g(\ldots X(f(z)) \ldots) \tag{2.1}$$

où

- *z est une* ***variable d'index*** *de dimension n.*
- *X est une* ***variable de donnée*** *correspondant à une fonction de n arguments entiers; on parle de variable de dimension n.*
- *$f : \mathcal{Z}^n \to \mathcal{Z}^n$ est une* ***fonction de dépendance****;*
- *g est une fonction scalaire; elle apparait souvent de manière implicite sous la forme d'une expression impliquant des opérandes de la forme $X(f(z))$, combinée avec des opérateurs de base et des parenthèses.*
- *D^X est un ensemble de points de $\mathcal{Z}^n$ et est appelé le* ***domaine*** *de l'équation. Les domaines sont très souvent des espaces d'index polyhédriques avec un ou plusieurs (notons ℓ) paramètres, $p \in \mathcal{Z}^\ell$.*

□

Une variable pourait être définie par plusieurs équations. Dans ce cas, on utilise la syntaxe ci-dessous:

$$X(z) = \left\{ \begin{array}{rcl} & \vdots & \\ D_i^X & : & g_i(\ldots X(f(z))\ldots) \\ & \vdots & \end{array} \right. \tag{2.2}$$

Chaque ligne correspond à un **cas**, et le domaine de X est une union des domaines disjoints de tous les cas, $D^X = \bigcup_i D_i^X$. On dit par ailleurs que la fonction de dépendance f est valable dans le (sous) domaine D_i^X.

Définition 12 *Une équation récurrente telle que définie par (2.1) est dite* ***affine*** *si chaque fonction de dépendance est de la forme $f(z) = Az + Bp + a$, où A (respectivement B) est une matrice constante $n \times n$ (respectivement $n \times l$) et a est un vecteur constant de dimension n. L'équation récurrente est dite* ***uniforme*** *si chaque fonction de dépendance est de la forme $f(z) = z + a$, où a est un vecteur constant de dimension n appelé vecteur de dépendance.* □

Définition 13 *Un* ***système*** *d'équations récurrentes* (SER) *est un ensemble de m équations récurrentes, définissant des variables de données $X_1, \cdots, X_m$, où chaque variable X_i est de dimension n_i. Puisque les équations du système sont mutuellement récursives, la fonction de dépendance f sera de type approprié.* □

Domaine.

Un important aspect du formalisme des équations récurrentes est la notion de *domaine*, ensemble d'indices sur lesquels des calculs particuliers sont définis. Ce domaine est d'habitude bien spécifié dans l'écriture d'un SER. Le type de domaine le plus souvent utilisé est le *polyhèdre* (ensemble de points entiers satisfaisant un nombre fini de contraintes linéaires d'(in)égalités), ou une réunion finie de polyhèdres.

Transformations de SER.

Une des plus importantes manipulations que l'on peut effectuer sur un SER est l'opération de *réindexation* (aussi appelée *changement de base* ou *réarrangement spatio-temporel*) de ses variables. La tranformation, notée $\mathcal{T}$, doit admettre une *réciproque à gauche* pour tous les points du domaine de la variable. Lorsqu'elle est appliquée à la variable X d'un SER défini comme suit:

$$X(z) \quad = \quad \left\{ \begin{array}{rcl} & \vdots & \\ D_i^X & : & g_i(\ldots Y(f(z))\ldots), \\ & \vdots & \end{array} \right.$$

le SER obtenu en appliquant les règles suivantes est équivalent au SER original:

– Remplacer chaque D_i^X par $\mathcal{T}(D_i^X)$, c'est à dire son image par $\mathcal{T}$.

- Dans la partie droite de l'équation définissant X, remplacer chaque dépendance f par $f \circ \mathcal{T}^{-1}$.
- Dans toutes les occurrences $X(g(z))$ de la partie droite de *chaque* équation, replacer la dépendance g par $\mathcal{T} \circ g$.

Pour le cas spécial des occurrences de X apparaissant dans la partie droite de l'équation définissant X, remplacer la fonction de dépendance f par $\mathcal{T} \circ f \circ \mathcal{T}^{-1}$.

2.3.2 Reproduction canonique

Définition 14 *Un système d'équations récurrentes sur $\mathcal{Z}^n$ est dit* **canoniquement reproductible** *s'il existe un sous-ensemble strict I de $\{1, ..., n\}$, tel que pour toute fonction de dépendance $f = (f_1, \cdots, f_n)$, chacune des projections $f_i, i \in I$, ne dépend éventuellement que des composantes z_i, $i \in I$. Le sous-ensemble I est appelé* ***direction*** *de reproduction, et son complémentaire $\bar{I}$ est appelé* ***base*** *de reproduction .* □

En d'autres termes, restreint au sous-espace de $\mathcal{Z}^n$ engendré par la famille $\{e_i, i \in I\}$ de la base canonique de $\mathcal{Z}^n$, le SER est *auto-dependant*.

Définition 15 *Une reproduction de direction I sera dite* **régulière**, *si on a aussi une reproduction de direction $\bar{I}$. En d'autres termes, pour toute fonction de dépendance $f = (f_1, \cdots, f_n)$, chacune des projections $f_i, i \in I$ (resp. $i \in \bar{I}$), ne dépend éventuellement que des composantes z_i, $i \in I$ (resp. $i \in \bar{I}$). Globalement, cela signifie que le* SER *considéré est auto-dependant dans chacune de ses restrictions aux sous-espaces supplémentaires engendrés respectivement par $\{e_i, i \in I\}$ et par $\{e_i, i \in \bar{I}\}$.* □

Dans ces définitions, il est à noter l'importance du fait que le sous-ensemble I doit être strict (c'est à dire $I \neq \emptyset \wedge I \neq \{1, \cdots, n\}$), autrement il répondrait trivialement au critère de reproductibilité sans être d'un quelconque intérêt.

Pour un SER *reproductible*, il peut exister plusieurs directions de reproduction, et il est facile de vérifier que l'ensemble des directions possibles est d'une certaine manière stable pour les opérations ensemblistes de base.

Propriétés 3.1.
(*i*) Toute intersection non vide de directions de reproduction est une direction de reproduction.
(*ii*) Toute réunion non pleine de directions de reproduction est une direction de reproduction.

Exemple 2.3.1 *Considérons le cas des récurrences fictives suivantes :*

$$X(i,j,k) = g(\cdots X(i-j,k,k)\cdots) \quad (2.3)$$
$$X(i,j,k) = g(\cdots X(i-k,j-i,k-j)\cdots) \quad (2.4)$$

La relation (2.3) implique une reproduction de direction $I = \{2, 3\}$, car en posant $\varphi(z) = z - f(z)$, on a $\varphi(i, j, k) = (j, j - k, 0)$, ce qui montre une auto-dependance du plan (j, k). Par contre la relation (2.4), qui correspond à $\varphi(i, j, k) = (k, i, j)$, n'admet pas de possibilité de reproduction canonique.

Il serait naturel, à ce niveau, de se poser la question de savoir s'il existe des systèmes *calculables* qui sont non *canoniquement reproductibles*. Le reflexe ayant conduit à cette question découle de l'entrelassement cyclique qu'on peut observer dans les dépendances *bloquantes*. Nous n'étudierons pas cet aspect du problème qui peut être aussi bien évident qu'extrêmement complexe.

Exemple 2.3.2 *Considérons maintenant l'exemple du problème du chemin algébrique. On a le système d'équations récurrentes suivant [119] (on a omis les domaines de calculs puisqu'on ne s'intéresse ici qu'à la forme syntaxique des dépendances).*

$$F(i,j,k) = \begin{cases} \{i,j,k \mid k = 0\} & : a_{i,j} \\ \{i,j,k \mid i = j = k\} & : F(i,j,k-1)* \\ \{i,j,k \mid i = k \neq j\} & : F(k,k,k) \otimes F(i,j,k-1) \\ \{i,j,k \mid j = k \neq i\} & : F(i,j,k-1) \otimes F(k,k,k) \\ \{i,j,k \mid i \neq k; j \neq k\} & : F(i,j,k-1) \oplus \\ & \qquad (F(i,k,k) \otimes F(k,j,k-1)) \end{cases} \tag{2.5}$$

On voit qu'on a une reproduction unique de direction $I = \{3\}$. Une réindexation appropriée [82] permet d'obtenir une version dans laquelle toutes les occurrences de k ne se trouvant pas en troisième position sont remplacées par une constante. Ainsi, tout sous-ensemble strict de $\{1, 2, 3\}$ devient éligible pour une direction de reproduction. Ceci illustre le fait que la réindexation est une opération qui peut augmenter le pouvoir de reproduction.

Définition 16 *Considérant une reproduction de direction I d'un* SER *de dimension n, l'entier $p = |I|$ (resp. $\frac{p}{n}$) sera appelé* ***indice absolu*** *(resp.* ***relatif****) de reproduction. Pour $p = 1$, $p = 2$, $p = 3$, et $p > 3$, on parlera respectivement de reproduction* ***linéaire****,* ***planaire****,* ***spatiale****, et* ***étendue****.* □

Dans l'exemple 2.3.2, on a une reproduction linéaire. Les systèmes d'équations récurrentes affines ont une forte capacité de reproduction, car tout sous-ensemble strict de $\{1, \cdots, n\}$ est une potentielle direction de reproduction.

L'intuition qui soutend le concept de *reproduction canonique* est la suivante. Partant d'un système d'équations récurrentes sur $\mathcal{Z}^n$ *canoniquement reproductible* et considérant une direction de reproduction I, si on a un ordonnancement du sous-système obtenu par projection du système entier sur le sous-espace de $\mathcal{Z}^n$ engendré par $\{e_i, i \in \bar{I}\}$, appelé *espace de base* (les composantes de rang dans I étant considérés ici comme des paramètres), alors on peut dériver un ordonnancement complet en reproduisant cet ordonnancement partiel le long du sous-espace de $\mathcal{Z}^n$ engendré par $\{e_i, i \in I\}$ (*espace de direction*). Selon que la reproductibilité est *régulière* ou non, la réplication sera *exacte* ou *biaisée*.

2.4 Synthèse à partir des équations récurrentes

2.4.1 Méthodologie

2.4.1.1 Fonction de temps

Considérons un SER canoniquement reproductible et I une direction de reproduction. Soit $\mathcal{F}$ l'ensemble de ses fonctions de dépendances. On procède comme suit:

Etape 1. Partitionner $\mathcal{F}$ en deux classes $\mathcal{F}_b$ et $\mathcal{F}_d$ définies par

$$\mathcal{F}_b = \{f \in \mathcal{F} : f_{/<e_i, i\in I>} = id_{/<e_i, i\in I>}\} \quad (2.6)$$

$$\mathcal{F}_d = \mathcal{F} - \mathcal{F}_b \quad (2.7)$$

En d'autres termes, $\mathcal{F}_b$ représente le sous-ensemble des fonctions de dépendance dont la projection sur le sous-espace $< e_i, i \in I >$ se réduit à l'identité. Ceci revient à dire que les dépendances de $\mathcal{F}_b$ n'ont d'influence que sur l'*espace de base*, tandis que celles de $\mathcal{F}_d$ régissent la précédence dans l'*espace de direction*.

Etape 2. Déterminer une fonction de temps valide, notée u, du SER projeté sur l'*espace de base*, avec pour seules contraintes celles induites par les dépendances de $\mathcal{F}_b$. Notons qu'à ce niveau, les occurences des composantes $z_i, i \in I$, peuvent intervenir mais plutôt comme des paramètres du sous-système. Ensuite, on en fait de même avec la projection du SER sur l'*espace de direction*, ce qui donne une fonction de temps que nous notons v (cette dernière fonction peut être constante dans le cas où $\mathcal{F}_d = \emptyset$).

Etape 3. Considérer une fonction de temps sur SER global sous la forme $t = u + \alpha v + \beta$, où β est une constante de réajustement, et α un coefficient strictement positif (pouvant dépendre des paramètres statiques du système global) qui permet d'introduire un délai entre une instance et sa reproduction. Ce délai a pour rôle de ne faire débuter une reproduction que lorsque les termes impliquées de l'instance précédente sont déjà mis à jour. Dans le cas d'un ordonnancement parallèle, on a $\alpha = 1$ au mieux, et $\alpha = \max u$ au pire.

Illustrons la technique sur l'exemple 2.3.2, avec la direction $I = \{3\}$. On a $\mathcal{F}_b = \{(i,j) \leftarrow (i,k), (i,j) \leftarrow (k,k)\}$ et $\mathcal{F}_d = \{k \leftarrow k-1\}$. On peut prendre $u(i,j) = |i-k| + |j-k| + (n-k)\delta(i<k)$, où $\delta(i<k) = 1$ si $i < k$ et 0 sinon, et $v(k) = k$. La fonction globale sera alors de la forme $t(i,j,k) = |i-k|+|j-k|+(n-k)\delta(i<k)+\alpha k+\beta$. Lorsque $i \geq k$ et $j \geq k$, la condition $t(i,j,k) \geq t(i,j,k-1)+1$, provenant de la dépendence $(i,j,k) \leftarrow (i,j,k-1)$, impose $\alpha \geq 3$. On vérifie aisément que la fonction $t(i,j,k) = |i-k| + |j-k| + (n-k)\delta(i<k) + 3k - 2$ convient. On obtient donc un ordonnancement qui s'exécute en $5n-4$ cycles de calcul sur n^2 processeurs, complexité caractérisant la plupart des solutions rencontrées dans la littérature (voir l'introduction de [119]).

2.4.1.2 Fonction d'allocation

Etant donné p processeurs, on procède comme suit. On considère une factorisation non triviale $p = ab$. Ensuite, on alloue les points de la base à b processeurs, et ceux de la direction à a processeurs. Ces allocations partielles A_b et A_d sont supposées valides par rapport aux fonctions de temps u et v, et de plus, l'allocation de direction A_d doit être **injective**. L'allocation globale est alors donnée par

$$A = (A_b, A_d). \tag{2.8}$$

L'ordonnancement partiel défini par (u, A_b) est appelé *ordonnancement générique*, et celui défini par (v, A_d) est appelé *ordonnancement de progression*. Dans l'exemple 2.1, on peut prendre $A_b(i,j) = j$ et $A_d(k) = k$, soit $A(i,j,k) = (j,k)$. Le résultat qui suit établit la validité de l'ordonnancement obtenu par notre approche.

Théorème 4 *Soit (u, A_b) un ordonnancement générique, et (v, A_d) un ordonnancement de progression. Soient ensuite α et β, deux scalaires tels que $t = u + \alpha v + \beta$ définit une fonction de temps globale valide. Alors $(t, A = (A_b, A_a))$ est un ordonnancement complet valide pour le* SER *considéré.* □

Preuve. Soient x et y, deux points distincts de $\mathcal{Z}^n$ tels que $t(x) = t(y)$. Montrons que $A(x) \neq A(y)$. Deux cas sont à envisager.

- x et y ont la même projection sur l'*espace de direction* ($x_i = y_i$, $\forall i \in I$). Ceci implique qu'on a $v(x) = v(y)$, et par suite $u(x) = u(y)$. Etant donné que x et y ont des projections distinctes sur l'*espace de base* (autrement on aurait $x = y$), on a l'égalité $u(x) = u(y)$ implique $A_b(x) \neq A_b(y)$, et par suite $A(x) \neq A(y)$.
- Les projections de x et y sur l'*espace de direction* sont distinctes. Du fait que A_d est injective, on a $A_d(x) \neq A_d(y)$ et par suite $A(x) \neq A(y)$. □

A propos du partitionnement

Puisqu'on a imposé que A_d soit injective, il faudrait donc que a (nombre de processeurs alloués sur la direction) soit égal au volume w de l'espace de direction. Si tel n'est pas le cas et si a est un diviseur de w, partant de l'allocation A_d définie en supposant un nombre non borné de processeurs, on dérive l'adaptation $\widetilde{A}_d$ ainsi qu'il suit. Considérons sans nuire à la généralité que la direction est $I = \{1, \cdots, q\}$. On factorise a en $a = a_1 \cdots a_q$ tel que a_i divise w_i,$i = 1, \cdots, q$, où w_i est le volume de la projection sur $\mathcal{Z} < e_i >$. L'allocation $\widetilde{A}_d$ est donnée par

$$\widetilde{A}_d(x_1, \cdots, x_q) = A_d(x_1 \bmod a_1, \cdots, x_q \bmod a_q) \tag{2.9}$$

Cette adaptation triviale constitue un très grand avantage des ordonnancements par reproduction (surtout lorsque la reproduction est régulière), car on a une bonne souplesse sur l'exigence en ressources matérielles. Toutefois, elle nécessite un aménagement de la fonction de temps afin d'éviter des conflits dûs à la réutilisation de l'architecture. Une

façon directe de résoudre ce problème consiste à achever un passage (mise à jour de a espaces de base consécutifs suivant la direction) avant d'entamer le passage suivant. Mais, dans certains cas, un enchaînement immédiat ou légèrement retardé est possible, avec pour conséquence une importante amélioration du temps total d'exécution. Dans l'exemple 2.3.2, on aurait besoin de $n \times n = n^2$ processeurs, soit n dans le sens de la direction. Si on avait $n \times m$ processeurs, où m est un diviseur de n, on aurait considéré l'allocation de direction $A_d(k) = k \bmod m$ $(0 \leq k \leq n-1)$, et la fonction de temps globale deviendrait $\widetilde{t} = t + 2n(k \; div \; m)$ ($2n$ est le le nombre de cycles nécessaires à la mise à jour d'un espace de base, et $k \; div \; m$ représente le rang du passage courant). Dans ce cas précis, un enchaînement retardé d'une constante est possible, cette étude est laissée au soin du lecteur motivé. Précisons tout de même qu'avec $m = n/3$ (soit $n^2/3$ processeurs), la fonction de temps n'a pas besoin d'être modifiée car l'exécution de $n/3$ reproductions consécutives nécessite $2n + 3 \times n/3 = 3n$ cycles, délai suffisant pour entamer les prochains calculs.

2.4.2 Complexité

Après avoir présenté le concept de *reproduction canonique*, et décrit une méthodologie de synthèse. Nous allons maintenant analyser la complexité des ordonnancements qu'on obtient, et faire ressortir les facteurs importants. Notons premièrement que l'obtention des ordonnancements de base et de direction peut se faire de manière ad hoc ou par l'usage des méthodologies appropriées de synthèse automatique. Toutefois, il faut faire des choix adéquats si l'on veut avoir une bonne complexité car, comme nous allons le préciser, l'efficacité globale dépend d'une bonne "coopération" entre les fonctions de temps partielles u et v. Si $\mathcal{D}$ désigne le domaine de calcul, $v(\mathcal{D})$ est un ensemble dénombrable, et par conséquent bien ordonné. On définit alors une relation de succession, qu'on dénote par $a \rightarrow b$ pour dire "b suit a" dans $v(\mathcal{D})$.

Définition 17 *Considérons un* SER *canoniquement reproductible, I une direction de reproduction, et v une fonction de temps de progression; pour $x \in \mathcal{Z}^n$, on considère $\mathcal{P}(x) = \{z \in \mathcal{Z}^n : v(z) = v(x)\}$. Pour u une fonction de temps générique vérifiant $\min u = 1$, et α un coefficient tel que $t = u + \alpha v$ est une fonction de temps valide, on définit la charge d'un point $x \in \mathcal{Z}^n$, notée $c(x)$, par*

$$c(x) = \max_{z \in \mathcal{P}(x)} u(z), \tag{2.10}$$

et pour deux points x et y tels que $v(x) \rightarrow v(y)$, on définit la latence de reproduction $\lambda(x, y)$ par

$$\lambda(x, y) = (\alpha + c(y) - c(x))\delta(\alpha + c(y) - c(x) > 0). \tag{2.11}$$

Enfin, on définit le rapport de reproduction $\rho(x, y)$ par

$$\rho(x, y) = \frac{\lambda(x, y)}{c(x)} \tag{2.12}$$

□

Ce ratio permet d'apprécier l'efficacité du pipeline. Plus il est petit, plus le pipeline est quantitativement bénéfique. Ceci vient du fait que, la fonction d'allocation de progression étant initialement injective, il vaut mieux qu'un volume considérable de calculs sur une base considérée restent à être effectués lorsque ceux de la base suivante sont entamés. De plus, les valeurs d'une base doivent être calculées de manière à ravitailler le plus immédiatement possible les calculs de la base suivante (à défaut, il y aura soit des cycles d'inactivité pour synchroniser, soit un besoin de mémoire pour les stockages intermédiaires). Notons par ailleurs que c'est cet aspect qui implique éventuellement un certain délai entre deux passes consécutives dans la version partitionnée, mais dans ce cas, un gain d'efficacité est envisageable à cause d'une réduction des cycles d'inactivité intervenant lors du premier et du dernier pipeline. Lorsque $\rho(x,y)=0$, le coût des calculs dans $\mathcal{P}(y)$ est recouvert par le coût de ceux dans $\mathcal{P}(x)$. Dans l'exemple 2.3.2, notre ordonnancement fournit un ratio égal à $\rho=\frac{3}{2n}$, ratio qui est évidemment satisfaisant. Dans le cas d'une reproduction non régulière, ce facteur peut varier en fonction des points choisis. On définit dans ce cas le *ratio minimal* ρ_{min}, le *ratio maximal* ρ_{max} et le *ratio moyen* ρ_{moy}. En supposant que v est réajusté de sorte que $\min v = 1$, on définit la *longueur de reproduction* γ par $\gamma = |v(\mathcal{D})|$ ($|.|$ est l'opérateur de cardinalité). En numérotant les élément de $v(\mathcal{D})$ dans l'ordre croissant, $v(\mathcal{D})=\{v_1,\cdots,v_\gamma\}$, on définit pour $k\in\{1,\cdots,\gamma\}$, $\mathcal{P}_k=\{x : v(x)=v_k\}$ et $c_k=\max\limits_{x\in\mathcal{P}_k} u(x)$. Le résultat suivant donne un encadrement du temps total d'exécution de l'ordonnancement.

Théorème 5 *Si $c_1\le c_2\le\cdots\le c_\gamma$, alors le temps total d'exécution $T_{//}$ de l'ordonnancement canonique vérifie,*

$$c_1(1+\rho_{min}\gamma)\le T_{//}\le c_\gamma(1+\rho_{max}\gamma) \tag{2.13}$$

□

Preuve. On a

$$T_{//}=c_\gamma+\sum_{k=1}^{\gamma-1}\lambda(x_k,x_{k+1}),\quad x_k\in\mathcal{P}_k \tag{2.14}$$

Du fait que $\lambda(x_k,x_{k+1})=c(x_k)\rho(x_k,x_{k+1})$, on a

$$T_{//}=c_\gamma+\sum_{k=1}^{\gamma-1}c_k\rho(x_k,x_{k+1}),\quad x_k\in\mathcal{P}_k \tag{2.15}$$

On obtient le résultat en appliquant à (2.15) les encadrements $c_1\le c_k\le c_\gamma, k=1,\cdots,\gamma$ et $\rho_{min}\le\rho(x_k,x_{k+1})\le\rho_{max}$. □

Remarquons que si on consière la condition duale $c_1\ge c_2\ge\cdots\ge c_\gamma$, on aura l'encadrement

$$c_\gamma(1+\rho_{min}\gamma)\le T_{//}\le c_1(1+\rho_{max}\gamma)$$

Dans notre exemple d'accompagnement, on a les ordres de grandeur suivants: $\gamma=n$, $c_1=c_n=2n$, $\rho_{min}=\frac{2}{2n}$, $\rho_{max}=\frac{4}{2n}$, ce qui donne $4n\le T_{//}\le 6n$, soit $5n$ en moyenne.

Notons que $4n$ est la meilleure complexité actuelle des ordonnancements pipelinés pour le problème du *chemin algébrique*.

Notons que la condition $c_1 \leq c_2 \leq \cdots \leq c_\gamma$, ou $c_1 \geq c_2 \geq \cdots \geq c_\gamma$, pourrait correspondre au mieux à la situation $c_1 = c_2 = \cdots = c_\gamma$ à une constante additive près. Un tel équilibre de charge assurerait une bonne régularité de l'ordonnancement. Dans tous les cas, il est clair que minimiser ρ_{max} conduirait à un temps total d'exécution meilleur. Ceci revient à rechercher parmi les fonctions temps u impliquant des charges équivalentes, celles pour lesquelles le coefficient α est minimal, ou celles qui réalisent un entrelassement dont l'effet de retardement est moins important.

2.4.3 Construction d'un ordonnancement générique efficace

2.4.3.1 Explication intuitive

Rappelons tout d'abord que l'ordonnancement de base est reproduit d'un pas à l'autre dans le sens de la direction. Deux cas peuvent se présenter:

• on a une reproduction régulière. Dans ce cas, l'efficacité de l'ordonnancement global *ne dépendra que* de l'efficacité relative de l'ordonnancement générique. En effet, le SER étant complètement *auto-dépendant* dans la base, un ordonnancement générique considéré n'aura pas d'influence qualitative sur la progression. Ce cas ne pose donc pas de problème particulier.

• on a une reproduction non régulière. Alors, soit on peut revenir au cas précédent à l'aide d'une réindexation appropriée, soit on est obligé (ça peut être aussi un choix délibéré) de poursuivre avec une configuration non régulière. C'est donc ici que ce pose le principal problème, car certaines composantes de $\mathcal{D}$ se retrouvent dans les dépendances dans $\mathcal{B}$. Ceci justifie le fait que l'ordonnancement générique doit tenir compte de l'ordonnancement de progression, à cause de l'influence des composantes de $\mathcal{D}$ sur celles de $\mathcal{B}$. Notre idée consiste à reproduire le mécanisme de progression sur la base, de façon à ravitailler au mieux le pipeline. Ce mécanisme peut être totalement ou partiellement pris en compte en fonction du type d'architecture recherchée.

2.4.3.2 Transformation formelle

Soit $\mathcal{B}$ (resp. $\mathcal{D}$) l'*espace de base* (resp. *de direction*) d'un SER. Etant donné une fonction de temps de progression v supposée injective, on procède comme suit pour lui associer une fonction de temps u de manière à obtenir un enchainement efficace.

• Définir une fonction $\phi : \mathcal{D} \to \mathcal{D}$ telle que $v(\phi(x)) \to v(x)$, $x \in \mathcal{D}$. Cette fonction existera toujours puisque v est injective. Dans l'exemple 2.3.2, on a $\phi(k) = k - 1$. En fait, la fonction ϕ fournit une modélisation *spatiale* du mécanisme de *progression*. L'idée va être de reproduire ce mécanisme sur la *base* à travers ses dépendances. Ainsi, à partir de chaque dépendance $z \leftarrow f(z)$ dans $\mathcal{B}$, on produit des dépendances *syntaxiquement interne* à $\mathcal{B}$, en remplaçant chaque occurence de composantes de $\mathcal{D}$ apparaissant dans $f(z)$ par la composante associée dans $\mathcal{B}$ à laquelle on a fait subir l'action de la fonction ϕ. Dans l'exemple 2.3.2, on obtient les dépendances $(i,j) \leftarrow (i-1,j)$ et $(i,j) \leftarrow (i,j-1)$ à partir des dépendances $(i,j) \leftarrow (i,k)$ et $(i,j) \leftarrow (k,k)$, et de $\phi(k) = k - 1$. Si tout se passe bien, cette étape se termine avec un système d'équations récurrentes complètement interne dans $\mathcal{B}$.

• Ordonnancer le système ainsi obtenu avec toutes ses dépendances, y compris celles obtenues par la technique précedente. Notons que si on a p processeurs, on doit tenir compte du fait qu'on n'en aura que $\frac{p}{\gamma}$ pour effectuer l'ordonnancement dans $\mathcal{D}$. Si on ne tient pas compte des ressources matérielles, on pourrait alors considérer les fonctions de temps au plus tôt ou au plus tard, ou toute autre fonction valide. Il convient tout de même de noter que, même si les variables de $\mathcal{D}$ n'apparaissent plus dans les dépendances, elles peuvent agir comme paramètres d'espace. Dans notre exemple, on aura:

$$\begin{cases} (i,j) \leftarrow (i-1,j) & : \quad (i \neq k) \\ (i,j) \leftarrow (i,j-1) & : \quad (j \neq k) \end{cases} \tag{2.16}$$

où les opérations sont faites modulo n (i.e $(1,j) \leftarrow (n,j)$ et $(i,1) \leftarrow (i,n)$). Pour ce cas, on pourais par exemple consiérer la fonction de temps donnée par:

$$u(i,j) = |i-k| + |j-k| + [2(i-k)+n]\delta(i<k) + [2(j-k)+n]\delta(j<k), \tag{2.17}$$

pour laquelle on a $\rho_{max} = \frac{3}{2n}$. En prenant en compte la combinaison $v(k) = k$ et $\alpha = 3$, on a un temps d'exécution global de l'ordre de $5n$, avec n^2 processeurs.

2.4.4 Dérivation de la version par bloc

Une application directe de notre méthode conduit dans un premier temps à une solution à grains fins, ce qui implique des communications portant sur des données élémentaires. Mais, dans le cadre des machines à mémoire distribuée, de tels ordonnancements conduiraient à des algorithmes inefficaces, à cause de la latence qui intervient à chaque appel d'une routine de communication. Une méthode bien connue pour venir à bout de ce problème est le regroupement en blocs des points de calculs [72, 73, 122, 120, 159, 160]. De la sorte, toutes opérations de calcul et de communication s'effectuent sur des blocs de données, limitant ainsi l'effet pénalisant de la latence. Toutefois, cette adaptation est une activité combinatoire généralement difficile car un regroupement des données créé des dépendances d'ensemble pouvant agir sur l'enchainement des tâches.

Dans le cas des ordonnancements canoniques, il suffit d'effectuer cette adaptation sur la base $\mathcal{B}$ par une approche de type LSGP (localement séquentiel et globalement parallèle). La nature de l'ordonnancement global reste ainsi conservée. Toutefois, il convient de noter le besoin d'une mémoire nécessaire au stockage et à la manipulation des groupes de données (ce qui ne consitue pas un problème particulier, puisqu'on est dans la cas des machines à mémoire locale). Le lecteur intéressé trouvera un exemple complet d'application de cette approche dans [21].

Nous venons d'étudier une méthode d'ordonnancement parallèle qui part du modèle des équations récurrentes. Il est important de noter que la condition de *reproductibilité*, qui revêt ici une forme syntaxique, pourrait être plus générale si on ajoute les transformations préalables telles que la *réindexation*.

Nous allons maintenant regarder ce qui se passe lorsqu'on considère le modèle de graphe de tâches comme point de départ de la méthode.

2.5 Ordonnancement à partir du graphe de dépendances

2.6 Décomposition canonique

L'idée de la *décomposition canonique* part du résultat suivant.

Théorème 6 *Soient $G_1 = (X_1, \Gamma_1)$ et $G_1 = (X_2, \Gamma_2)$ deux graphes isomorphes, et soit φ un isomporphisme de G_1 vers G_2. Si (t, a) est un ordonnancement valide de G_1, alors $(t \circ \varphi^{-1}, a \circ \varphi^{-1})$ est un ordonnancement valide de G_2.* □

Preuve.
• Soient $x, y \in X_2$ tels que $(x, y) \in \Gamma_2$. Montrons que $t \circ \varphi^{-1}(y) > t \circ \varphi^{-1}(x)$. En effet, $(x, y) \in \Gamma_2$ implique $(\varphi^{-1}(x), \varphi^{-1}(y)) \in \Gamma_1$, et par suite $t(\varphi^{-1}(y)) > t(\varphi^{-1}(x))$.
• Soient $x, y \in X_2$ tels que $t \circ \varphi^{-1}(x) = t \circ \varphi^{-1}(y)$. Montrons que $a \circ \varphi^{-1}(x) \neq a \circ \varphi^{-1}(y)$. On peut écrire $x = \varphi(u)$ et $y = \varphi(v)$, où $u, v \in X_1$. On obtient ainsi $t(u) = t(v)$, ce qui entraîne $a(u) \neq a(v)$, c'est à dire $a(\varphi^{-1}(x)) \neq a(\varphi^{-1}(y))$. □

On remarque donc qu'on peut se servir d'un isomorphisme de graphe pour transporter un ordonnancement d'un graphe à un autre. Pour appliquer ce principe à un graphe donné, il faudrait pouvoir le décomposer en une chaîne de sous-graphes isomorphes, et appliquer de proche en proche la technique précédente à partir d'un ordonnancement du sous-graphe initial. Ceci nous amène donc au concept de *décomposition canonique* que nous allons décrire.

Définition 18 *Un graphe de tâche $G = (T, P)$ sera dit canoniquement reproductible s'il existe une partition $T = T_1 \cup T_2 \cup \cdots \cup T_\gamma$ telle que les sous-graphes induits par T_k et T_{k+1}, $1 \leq k < \gamma$, sont isomorphes (i.e. il existe une bijection $\phi_k : T_{k+1} \to T_k$ telle que pour tout $x, y \in T_{k+1}$, $(x, y) \in P \iff (\phi_k(x), \phi_k(y)) \in P$), et de plus, toutes les dépendances directes sont internes aux T_k ou localisées entre T_k et T_{k+1} dans le sens de T_k vers T_{k+1} (i.e. $\forall (x, y) \in T^2 : (x, y) \in P \Rightarrow (x, y \in T_k) \vee (x \in T_k \wedge y \in T_{k+1})$).* □

Dans notre exemple du chemin algébrique, on peut considérer

$$T_k = \{(i, j, k) : 1 \leq i, j \leq n\}, k = 1, \cdots, n, \tag{2.18}$$

avec les isomorphismes ϕ_k définies par:

$$\phi_k(i, j, k) = \begin{cases} (1, 1, k+1) & : \quad i = n \wedge j = n \\ (1, j+1, k+1) & : \quad i = n \wedge j < n \\ (i+1, 1, k+1) & : \quad i < n \wedge j = n \\ (i+1, j+1, k+1) & : \quad i, j < n \end{cases} \tag{2.19}$$

Notons au passage que de tels isomorphismes peuvent guider la recherche d'une *réindexation* du SER, de manière à le rendre par exemple affine. Par ailleurs, il est clair que pour un graphe donné, il peut exister plusieurs décompositions possibles. De plus, le fait d'impliquer des isomorphismes découle d'un souci de régularité et d'équilibre dans les calculs. On comprend alors qu'on puisse envisager des ordonnancements des T_k ayant des complexités équivalentes à une constante additive près. Toutefois, on pourrait dans certains cas se contenter des applications injectives (surtout lorsque $|T_{k+1}| < |T_k|$).

2.7 Technique d'ordonnancement

On définit $\pi_k : T_k \to T_1$ par $\pi_k = \phi_1 o \phi_2 o \cdots o \phi_{k-2} o \phi_{k-1}$. Soit (u, A_b) un ordonnancement de T_1, où u est une fonction de temps et A_b une fonction d'allocation. On considère sur chaque $T_k, k > 1$, la fonction de temps relative $u_k = u o \pi_k$. Soit maintenant $\{v_k, k = 1, \cdots, k-1\}$, une famille de termes scalaires tels que v_k de dépend que de k et éventuellement des paramètres statiques du graphes, qui vérifie

$$\forall (x, y) \in T_k \times T_{k+1} : (x, y) \in P \Rightarrow u_{k+1}(y) - u_k(x) + v_{k+1} > 0. \tag{2.20}$$

On définit alors une fonction de temps globale $t : T \to \mathcal{N}$ comme suit:

$$T_k : t(z) = u(\pi_k(z)) + v(k), \tag{2.21}$$

où $v(k) = v_k + v_{k-1} + \cdots + v_1$.

Théorème 7 *L'expression (2.21) définit un fonction de temps valide pour le graphe* $G = (T, P)$. □

Preuve. Soient $x, y \in T$ tels que $(x, y) \in P$. Montrons que $t(y) > t(x)$. En effet, d'après les propriétés du graphe G, on a deux cas à envisager:

- $x, y \in T_k$, $1 \leq k \leq \gamma$.

D'après la définition de π_k, on a $(\pi_k(x), \pi_k(y)) \in P$ et par conséquent $u(\pi_k(y)) > u(\pi_k(x))$ car $\pi_k(x), \pi_k(y) \in T_1$ et u est un timing valide de T_1. Puisque $x, y \in T_k$, cette dernière inégalité conduit au résultat en ajoutant l'expression $v(k)$ à chacun de ses membres.

- $x \in T_k$ et $y \in T_{k+1}$. D'après (2.21), on a

$$\begin{aligned} t(y) - t(x) &= (u_{k+1}(y) + v(k+1)) - (u_k(x) + v(k)) \\ &= u_{k+1}(y) - u_k(x) + (v(k+1) - v(k)) \\ &= u_{k+1}(y) - u_k(x) + v_{k+1} > 0 \end{aligned} \tag{2.22}$$

Ceci achève la preuve du résultat. □

La fonction d'allocation globale $A : T \to T$ quant à elle donnée par

$$T_k : A(x) = (k, A_b(\pi_k(x))). \tag{2.23}$$

Théorème 8 *Les fonctions de temps et d'allocation définies respectivement par (2.21) et (2.23) forment un ordonnancement global valide du graphe* $G = (T, P)$. □

Preuve. Soient $x, y \in T$ tels que $A(x) = A(y)$. Montrons que $t(x) \neq t(y)$. En effet, $A(x) = A(y)$ implique qu'il existe $k \in \{1, \cdots, \gamma\}$ tel que $x, y \in T_k$. On a ensuite $A_b(\pi_k(x)) = A_b(\pi_k(y))$, qui implique $u(\pi_k(x)) \neq u(\pi_k(x))$ car $\pi_k(x), \pi_k(y) \in T_1$ et (u, A_b) est un ordonnancement valide de T_1. En ajoutant $v(k)$ dans les deux membres de cette dernière inégalité, on obtient $t(x) \neq t(y)$, ce qui achève la démonstration. □

Dans notre, exemple, l'isomorphisme défini par (2.19) donne

$$\pi_k^{-1}(i,j,k) = \begin{cases} (i-k+1+n, j-k+1+n, 1) & : \quad i-k<0 \wedge j-k<0 \\ (i-k+1+n, j-k+1, 1) & : \quad i-k<0 \wedge j-k\geq 0 \\ (i-k, j-k+n, 1) & : \quad i-k\geq 0 \wedge j-k<0 \\ (i-k, j-k, 1) & : \quad i-k\geq 0 \wedge j-k\geq 0 \end{cases} \tag{2.24}$$

Reste maintenant à trouver un ordonnancement générique (u, A_b) de T_1. Pour cela, on considère l'ordonnancement suivant:

$$u(i,j,1) = i+j-1 \tag{2.25}$$

$$A_b(i,j,1) = j \tag{2.26}$$

La fonction d'allocation globale sera

$$A(i,j,k) = (k,j). \tag{2.27}$$

Pour la fonction de temps, on doit d'abord déterminer u_k et v_k. D'après la définition $u_k = uo\pi_k$, on a

$$u_k(i,j,k) = \begin{cases} u(i-k+1+n, j-k+1+n, 1) & : \quad i-k<0 \wedge j-k<0 \\ u(i-k+1+n, j-k+1, 1) & : \quad i-k<0 \wedge j-k\geq 0 \\ u(i-k+1, j-k+1+n, 1) & : \quad i-k\geq 0 \wedge j-k<0 \\ u(i-k+1, j-k+1, 1) & : \quad i-k\geq 0 \wedge j-k\geq 0 \end{cases} \tag{2.28}$$

soit

$$u_k(i,j,k) = \begin{cases} (i-k+1+n)+(j-k+1+n)-1 & : \quad i-k<0 \wedge j-k<0 \\ (i-k+1+n)+(j-k+1)-1 & : \quad i-k<0 \wedge j-k\geq 0 \\ (i-k+1)+(j-k+1+n)-1 & : \quad i-k\geq 0 \wedge j-k<0 \\ (i-k+1)+(j-k+1)-1 & : \quad i-k, j-k\geq 0 \end{cases} \tag{2.29}$$

Cette expression peut se mettre sous la forme compact suivante:

$$u_k(i,j,k) = i+j-2k+1+n[\delta(i-k<0)+\delta(j-k<0)]. \tag{2.30}$$

En remarquant que $u_{k+1}(i,j,k+1) - u_k(i,j,k) \geq 2$, il vient qu'on a $(v_1 = 0) \wedge (v_k = 3 : k > 1)$ satisfait la condition (2.20). Tout ceci conduit à la fonction de temps globale donnée par

$$t(i,j,k) = i+j-2k+1+n[\delta(i-k<0)+\delta(j-k<0)]+3(k-1), \tag{2.31}$$

soit tout simplement

$$t(i,j,k) = i+j+k-2+n[\delta(i-k<0)+\delta(j-k<0)]. \tag{2.32}$$

On obtient un algorithme qui s'exécute en $t(n-1,n-1,n) = 5n-4$ étapes sur n^2 processeurs.

Dans les cas particuliers de la fermeture transitive et des plus courts chemins, du fait qu'on est affranchi du traitement des lignes $i = k$ et $j = k$, on peut prendre l'ordonnancemnt générique suivant:

$$u'(i,j,1) = \begin{cases} n+2-(i+j) : i+j \leq n+1 \\ 2n-(i+j)+2 : i+j > n+1 \end{cases} \tag{2.33}$$

$$A'_b(i,j,1) = j \tag{2.34}$$

Remarquons qu'on peut écrire

$$u'(i,j,1) = n+2-(i+j)+n\delta(i+j > n+1), \tag{2.35}$$

et par suite $u'(i,j,1) = n+3-u(i,j,1)+n\delta(u(i,j,1) > n)$. On obtient donc $u'_k(i,j,k) = n+3-u_k(i,j,k)+n\delta(u_k(i,j,k) > n)$, et l'ordonnancement recherché est donné par

$$t'(i,j,k) = n+3k-u_k(i,j,k)+n\delta(u_k(i,j,k) > n) \tag{2.36}$$

$$A'(i,j,k) = (k,j) \tag{2.37}$$

On obtient un algorithme qui s'exécute en $t(1,1,n) = 4n-3$ étapes sur n^2 processeurs. Ceci illustre l'importance de l'ordonnancement générique sur la performance globale.

Mieux encore, lorsqu'on observe que les calculs de T_1 sont achevés après n cycles (soit $3n$ cycles d'inactivités par la suite), il vient qu'on pourait avoir le même temps d'exécution en utilisant $n/3$ groupes de processeurs pour le calculs des T_k. Cette nouvelle solution, qui n'est qu'une application du partitionnement par passes multiples telle que expliqué précédemment, conduit à un ordonnancement qui s'exécute en $4n-3$ étapes sur $n \times n/3 = n^2/3$ processeurs, soit une efficacité égale à $\frac{3}{4}$. Cette dernière solution est spécifiée par les fonctions suivantes:

$$t'(i,j,k) = n+3k-u_k(i,j,k)+n\delta(u_k(i,j,k) > n) \tag{2.38}$$

$$A'(i,j,k) = ([(k-1) \bmod \frac{n}{3}]+1, j) \tag{2.39}$$

2.7.1 Etude combinatoire de la décomposition canonique du graphe

Remarquons que la méthode précédente n'a pas nécessairement besoin d'un isomorphisme entre T_{k+1} et T_k. En effet, il nous suffit d'avoir

$$\text{Pour } x, y \in T_{k+1}, (x,y) \in P \Rightarrow (\phi_k(x), \phi_k(y)) \in P. \tag{2.40}$$

Ceci allège les critères à vérifier par la décomposition du graphe sans pour autant influer sur la technique de construction de l'ordonnancemment. Toutefois, l'isomorphisme garantit que la reproduction de l'ordonnancement de T_1 sur les autres classes $T_k : k > 1$ aura une complexité relative aussi liée que possible aux dépendances dans T_k qu'elle ne l'aura été dans T_1. En effet, on pourrait avoir moins de dépendances dans T_{k+1} que dans T_k auquel cas, reproduire le timing de T_k sur T_{k+1} reviendrait en quelque sorte à prendre compte dans T_{k+1} des dépendances virtuelles (transportées de T_k vers T_{k+1} par la correspondance). Corriger une telle situation conduirait à une perte de modularité de

l'ordonnancement global. Aussi, nous allons poursuivre notre analyse avec la condition (2.40), avec en prime la liberté d'avoir $|T_k| \geq |T_{k+1}|$ au lieu de l'égalité. Enfin, notons aussi que si on a des dépendances (x, y) telle que $x \in T_k$ et $y \in T_{k+\lambda}$, $\lambda > 1$, on peut tout simplement la remplacer par la cascade de dépendances

$$(x, x_{k+1}), \{(x_t, x_{t+1}) : t = k+1, \cdots, k+\lambda-1\}, (x_{k+\lambda-1}, y), \tag{2.41}$$

où $x_{k+1}, \cdots, x_{k+\lambda-1}$ sont des sommets de $T_{k+1}, \cdots, T_{k+\lambda-1}$ respectivement, choisis de façon judicieuse.

Nous aimerions rappeler une fois de plus que les remarques précédentes visent à simplifier l'étude de l'existence et la construction d'une partition pouvant servir de point de départ de la méthode. Toutefois, le cas idéal reste celui de la définition d'origine avec ses avantages de régularité, d'équilibre et de modularité.

La méthode requiert donc au minimum une partition $T = T_1 \cup T_2 \cup \cdots \cup T_\gamma$ avec des applications injectives $\varphi_k : T_{k+1} \to T_k$ telles que:
(i) Pour $x, y \in T_{k+1}, (x, y) \in P \Rightarrow (\varphi_k(x), \varphi_k(y)) \in P$,
(ii) $(x, y) \in P \Rightarrow \exists i, j; j \geq i : (x \in T_i) \wedge (y \in T_j)$.

Le résultat suivant montre la difficulté générale de la décomposition dans sa forme originale.

Théorème 9 *Soient $G_1 = (X_1, \Gamma_1)$ et $G_2 = (X_2, \Gamma_2)$ deux graphes orientés tels que $|X_1| = |X_2|$. G_1 et G_2 sont isomorphes si et seulement si le graphe orienté défini par $G = (X_1 \cup X_2, \Gamma_1 \cup \Gamma_2 \cup (X_1 \times X_2))$ admet une décomposition canonique de longueur 2.* □

Preuve.
• G_1 et G_2 sont isomorphes. Il suffit de prendre $T_1 = X_1$ et $T_2 = X_2$ pour avoir une décomposition canonique d'ordre 2 dans $G = (X_1 \cup X_2, \Gamma_1 \cup \Gamma_2 \cup (X_1 \times X_2))$.
• $G = (X_1 \cup X_2, \Gamma_1 \cup \Gamma_2 \cup (X_1 \times X_2))$ admet une décomposition canonique T_1 et T_2. Supposons que T_1 soit différent de X_1. Ceci implique qu'on a $T_1 \cap X_2 \neq \emptyset$ et $T_2 \cap X_1 \neq \emptyset$. $X_1 \times X_2 \subset \Gamma$ implique alors qu'on a un arc de T_2 vers T_1, ce qui n'est pas conforme à la définition de la décomposition canonique. □

Bien que la décomposition canonique soit en général un problème combinatoire apparament difficile comme le montre le théorème 9, la donnée du graphe sous forme de système d'équations récurrentes sur des domaines polyhédriques simplifie le problème. En effet, la décomposition recherchée devient le résultat de transformations analytiques comme nous allons le voir dans les exemples qui vont suivre.

2.8 Applications

2.8.1 La factorisation de Cholesky

Considérons le problème de la factorisation de Cholesky qui, pour une matrice carrée A d'ordre n symétrique définie positive, consiste à construire une matrice triangulaire inférieure $L = (l_{ij}), 1 \leq j \leq i \leq n$ telle que $A = LL^T$. Les équations récurrentes

correspondantes [30] sont données par:

$$l(i,j,k) \quad = \quad \begin{cases} D \cap \{(k=-1) \wedge (j \leq i)\} & : a_{i,j} \\ D \cap \{i=j=k+1\} & : l(i,j,k-1)^{\frac{1}{2}} \\ D \cap \{(j=k+1) \wedge (i>j)\} & : l(i,j,k-1)/l(k+1,k+1,k) \\ D \cap \{k \leq j-2\} & : l(i,j,k-1) - \\ & \qquad l(i,k+1,k) \times l(j,k+1,k) \end{cases} \tag{2.42}$$

où $D = \{(i,j,k) : -1 \leq k < j \leq i \leq n\}$.

a) Synthèse directe

On a une direction de reproduction donnée par $I = \{3\}$, ce qui nous donne les classes de dépendances $\mathcal{F}_b = \{(i,j) \leftarrow (i,k+1), (i,j) \leftarrow (j,k+1), (i,j) \leftarrow (k+1,k+1)\}$ et $\mathcal{F}_d = \{k \leftarrow k-1\}$. En prenant $v(k) = k$ et en appliquant la technique décrite dans la section 2.4.3, on obtient $\phi(i,j) = \{(i,j-1), (k+1,k+1)\}$ ($\phi(k) = k-1$). Ce qui produit les dépendances suivantes:

$$\begin{cases} (i,j) \leftarrow (k+1,k+1) & : \quad j \geq k+1 \\ (i,j) \leftarrow (i,j-1) & : \quad j \geq k+2 \end{cases} \tag{2.43}$$

On peut prendre $u(i,j) = i+j-2k-1$, et on a alors une fonction de temps de la forme $t(i,j,k) = i+j+\alpha k+\beta$ (les termes $2k$ et -1 ont été immergés dans la partie $\alpha k+\beta$). La prise en compte de la dépendance $(i,j,k) \leftarrow (i,j,k-1)$ conduit à la condition $\alpha \geq 1$. En prenant donc $\alpha = 1$, on a la fonction de temps globale suivante

$$t(i,j,k) = i+j+k-1. \tag{2.44}$$

Pour les allocations, on propose $A_b(i,j) = (j-k)\delta(i+j \leq n) + (i-k)\delta(i+j > n)$ et $A_d(k) = k$, ce qui donne

$$A(i,j,k) = ((j-k)\delta(i+j \leq n) + (i-k)\delta(i+j > n), \quad k). \tag{2.45}$$

On a donc un ordonnanancement qui fonctionne en $t(n,n,n-1) = 3n-2$ étapes sur $n(n-1)/4$ processeurs, soit un travail dans l'ordre de $\frac{3}{4}n^3$.

b) Synthèse à partir du graphe de dépendance

Par rapport à la précédente description, on a

$$T_k = \{(i,j,k) \mid k < j \leq i \leq n\} : 0 \leq k \leq n-1 \tag{2.46}$$

$$\varphi_k(i,j,k) = (i-1, j-1, k-1) \tag{2.47}$$

Cherchons un ordonnancement de T_0. Ceci revient à considérer le schéma de récurrence suivant sur le domaine $D = \{(i,j) : 1 \leq j \leq i \leq n\}$:

$$l(i,j) \quad = \quad \begin{cases} D \cap \{i=j=1\} & : \sqrt{a_{ij}} \\ D \cap \{(i \geq 2) \wedge (j=1)\} & : a_{ij}/l(j,j) \\ D \cap \{j \geq 2\} & : a_{ij} - l(i,1) \times l(j,1) \end{cases} \tag{2.48}$$

Un ordonnancement possible est donné par

$$\begin{cases} u(i,j) = i + j - 1 \\ A_b(i,j) = j\delta(j \leq \lceil \frac{n}{2} \rceil) + (j - \lceil \frac{n}{2} \rceil)\delta(j > \lfloor \frac{n}{2} \rfloor) \end{cases} \tag{2.49}$$

La figure Fig. 2.1 illustre l'ordonnancement générique pour les cas $n = 7$ et $n = 8$. Les charges sont réparties en colonnes de manière cyclique sur $n/2$ processeurs. Continuons

1						
1	2					
2	3	3				
3	4	5	4			
4	5	6	7	1		
5	6	7	8	9	2	
6	7	8	9	10	11	3
7	8	9	10	11	12	13

1							
1	1						
2	3	2					
3	4	5	3				
4	5	6	7	4			
5	6	7	8	9	1		
6	7	8	9	10	11	2	
7	8	9	10	11	12	13	3
8	9	10	11	12	13	14	15

Note: La valeur au dessus de chaque colonne représente le processeur concerné.

FIG. 2.1 – *Ordonnancement de T_0 pour les cas $n = 7$ et $n = 8$.*

notre synthèse en déterminant $u_k = uo\pi_k$ et v_k. Sachant que $\pi_k = \varphi_0 o \cdots o \varphi_{k-1}$, on a

$$\pi_k(i,j,k) = (i - k, j - k, 0). \tag{2.50}$$

La relation $u_k(i,j,k) = u(i-k, j-k, 0)$ nous donne

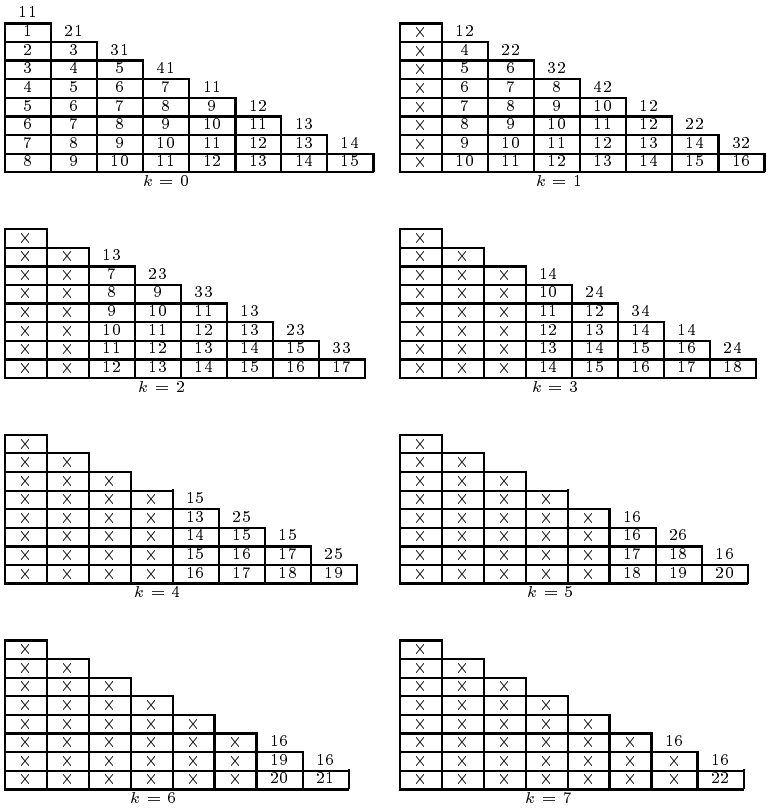
$$u_k(i,j,k) = (i-k) + (j-k) - 1 = i + j - 2k - 1 \tag{2.51}$$

En étudiant la quantité $u_k(i,j,k) - u_{k-1}(i,j,k-1)$, on obtient la condition $v_k \geq 3$. Nous prendrons donc $v_k = 3$ ($v_0 = 0$), soit $v(k) = 3k$. En posant $i_k = i - k$, $j_k = j - k$ et $n_k = n - k$, on a l'ordonnancement global donné par

$$\begin{cases} t(i,j,k) = i + j + k - 1 \\ A(i,j,k) = (\ j_k\delta(j_k \leq \lceil \frac{n_k}{2} \rceil) + (j_k - \lceil \frac{n_k}{2} \rceil)\delta(j_k > \lceil \frac{n_k}{2} \rceil),\ k+1\ ) \end{cases} \tag{2.52}$$

L'algorithme s'exécute en $t(n,n,n-1) = 3n - 2$ cycles avec $\frac{n^2}{4} + O(n)$ processeurs. Le timing est le même celui obtenu dans la version précédente, mais l'allocation est légèrement modifée. La figure Fig. 2.2 illustre l'ordonnancement complet pour le cas d'une matrice d'ordre $n = 8$. En observant cet ordonnancement, on constate que le premier processeur $(1,1)$ est libre au moment où débutent les calculs de $T_{\frac{n}{2}}$ (on peut le montrer facilement par les formules). Ceci nous amène à considérer l'allocation adaptée

$$A'(i,j,k) = (\ j_k\delta(j_k \leq \lceil \frac{n_k}{2} \rceil) + (j_k - \lceil \frac{n_k}{2} \rceil)\delta(j_k > \lceil \frac{n_k}{2} \rceil),\ k \bmod [\frac{n}{2}] + 1\ ) \tag{2.53}$$



ij=processeur de rang i du groupe j.

FIG. 2.2 – *Ordonnancement global pour le cas* $n = 8$*:* 22 *cycles avec* 20 *processeurs*

grâce à laquelle on n'aura besoin que d'environ $\frac{n^2}{8} + O(n)$ processeurs, soit un travail de l'ordre de $\frac{3}{8}n^3$ et par suite une efficaité égale à $\frac{4}{9}$ (La complexité séquentielle est de l'ordre de $\frac{n^3}{6}$). La figure 2.3 illustre l'architecture basée sur l'allocation originale pour $n = 8$.

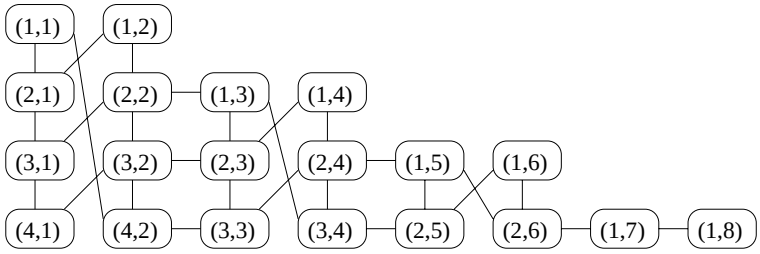


FIG. 2.3 – *Topologie pour la factorisation de Cholesky avec* $n = 8$.

Cette figure montre la modularité du réseau pour des valeurs de n ayant la même parité. Au besoin, on pourrait factoriser la matrice augmentée exprimée par

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}.$$

2.9 Conclusion

Nous avons proposé, analysé et illustré une méthodologie de conception d'ordonnancements réguliers. Cette méthodologie s'avère assez efficace pour des schémas de calcul de type programmation dynamique. De plus, la méthode n'est pas liée à la nature des dépendances, ce qui laisse entrevoir le fait qu'elle peut permettre d'ordonnancer à priori des graphes ayant des dépendances quelconques. Les solutions synthétisées ont une bonne modularité, aussi bien au niveau de l'architecture qu'au niveau du flot de données. Toutefois, il reste à approfondir l'aspect physique des réseaux dérivés, principalement l'interconnexion entre les groupes de processeurs dans les enchaînements, et la complexité de la mémoire dans la version partitionnée.

Chapitre 3

Parallélisation du produit tensoriel

Ce chapitre présente un ensemble de résultats sur le sujet de la parallélisation de la multiplication d'un vecteur par un *produit tensoriel* (ou *produit de Kronecker*) de matrices. Ces différentes études sont publiées dans [142, 144, 143, 147].

3.1 Résumé

Nous présentons ici un ensemble de résultats algorithmiques relatifs aux opérations de l'algèbre tensorielle. Principalement, nous étudions la multiplication d'un vecteur par un produit tensoriel de matrices. Nos résultats sont dérivés d'un modèle de calcul basé sur la *factorisation canonique* et une représentation multidimensionnelle des opérandes. Dans le cas séquentiel, nous présentons un schéma général, suivi d'une version minimisant l'espace mémoire nécessaire, et ensuite une version régulière et naturellement vectorisable pour le cas de matrices de mêmes tailles. Dans le cas parallèle sur machine à mémoire distribuée, noous développons un algorithme qui minimise le coût des communications entre les processeurs et la mémoire locale de chaque processeur. Une étude de complexité montre que, pour entier p qui est un diviseur de la taille du problème, le coût minimal pour la multiplication sur p processeurs implique $\Theta(log(p))$ étapes de communication. Cette borne est revue en fonction des différentes topologies standards. Quelques résultats expérimentaux sur la CRAY T3E confirment l'efficacité de nos algorithmes.

3.2 Introduction

Le *produit tensoriel* (ou *produit de Kronecker*) désigne une opération bien connue de l'algèbre matricielle [41], qui s'est révélée être un outil fondamental pour la modélisation et la conception d'algorithmes [63] dans des domaines tels que: les *Réseaux d'Automates Stochastiques* [50], la *Transformée de Fourier Rapide*, le *Solveur de Poisson Rapide* [158], et le *Calcul Quantique* [139]. Spécifiquement, le problème de la multiplication d'un vecteur par un produit tensoriel de matrices semble être d'un intérêt particulier.

Rappelons que, pour N matrices carrées d'ordre $n_i, i = 1, ..., N$, le nombre total de

multiplications flottantes est donné par

$$\rho_N = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i.$$

Cette complexité est satisfaite en faisant recours à la forme *normale* de la matrice principale. Un résumé des travaux relatifs à cette multiplication peut être consulté dans [158]. La plupart de ces travaux considèrent une distribution en blocs des vecteurs de calculs, opérant par conséquent une permutation des valeurs, laquelle permutation engendre un mécanisme de communication assez coûteux. Une distribution cyclique est considérée avec succès dans [151, 142].

Notre idée est basée sur une distribution bloc cyclique générale, conçue de telle manière que tous les déplacements de données se retrouvent dans le jeu de communication entre les processeurs. Sur p processeurs, p étant un quelconque diviseur de $L = n_1 n_2 ... n_N$ (la taille du problème), notre algorithme s'exécute en $\rho_N/p + (\alpha L/p + \beta) \times O(\Gamma(p))$, où α est la vitesse de transmission, β le coût de synchronisation, et Γ le coût d'une *diffusion* (en termes de nombre d'étapes de communication). Dans le cas d'une topologie à coût de diffusion logarithmique, on opère $\Theta(log(p))$ étapes de communication, complexité que nous établissons comme étant la meilleure possible. Toutefois, soit parce que la diffusion implique une bufferisation importante, soit parce qu'un chevauchement entre les calculs et les communications est possible, une boucle de communications pourrait être exécutée (à la place des diffusions optimales), et dans ce cas, les meilleures performances sont le prix de la résolution d'un problème combinatoire difficile, conduisant à l'usage des heuristiques fort heureusement satisfaisantes.

3.3 Préliminaires et Formalisme

Le *produit tensoriel* est une opération de l'algèbre matricielle qui se définit comme suit.

Définition 19 *(produit tensoriel). Si $A \in \mathcal{R}^{n_A \times m_A}$ et $B \in \mathcal{R}^{n_B \times m_B}$ alors le produit tensoriel $C = A \otimes B$ est la n_A-by-m_A matrice bloc*

$$C = (a_{ij}B) \in \mathcal{R}^{n_A n_B \times m_A m_B}. \tag{3.1}$$

□

Exemple 3.3.1

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \mathcal{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Le produit de Kronecker $C = A \otimes B$ est donné par

$$\mathcal{C} = \left(\begin{array}{cc|cc|cc} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} & a_{13}b_{11} & a_{13}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} & a_{13}b_{21} & a_{13}b_{22} \\ \hline a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} & a_{23}b_{11} & a_{23}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} & a_{23}b_{21} & a_{23}b_{22} \\ \hline a_{31}b_{11} & a_{31}b_{12} & a_{32}b_{11} & a_{32}b_{12} & a_{33}b_{11} & a_{33}b_{12} \\ a_{31}b_{21} & a_{31}b_{22} & a_{32}b_{21} & a_{32}b_{22} & a_{33}b_{21} & a_{33}b_{22} \end{array} \right)$$

Quelques propriétés du *produit tensoriel* peuvent être trouvées dans [158]. Parmi elles, on peut citer:

- L'associativité

$A \otimes (B \otimes C) = (A \otimes B) \otimes C$

- Compatibilité avec la multiplication matricielle ordinaire

$(A \times B) \otimes (C \times D) = (A \otimes C) \times (B \otimes D)$

- *Factorisation*

$(A \otimes B) = (A \otimes I_{n_B}) \times (I_{n_A} \otimes B) = (I_{n_A} \otimes B) \times (A \otimes I_{n_B})$

Du fait de l'associativité, le produit tensoriel $A^{(1)} \otimes A^{(2)} \otimes ... \otimes A^{(N)}$ de N matrices $A^{(i)}, i = 1, ..., N$, dénoté par $\otimes_{i=1}^{N} A^{(i)}$, est bien définie et nous avons la *factorisation canonique*

$$\otimes_{i=1}^{N} A^{(i)} = \prod_{s=1}^{N} (I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}), \tag{3.2}$$

où la multiplication matricielle intervenant est commutative avec ces facteurs spéciaux appelés *facteurs normaux*.

L'opérateur $\otimes$ n'est pas commutatif. Toutefois, on a une pseudo-commutativité par le biais des matrices de permutations [41]. Rappelons que pour une *permutation* donnée σ de l'ensemble des entiers $\{1, 2, ..., n\}$, la matrice de permutation correspondante P_σ est définie par $P_\sigma = (e_{\sigma(1)}, ..., e_{\sigma(n)})$ où e_i est le i^{th} vecteur colonne de la base canonique.

Il existe une permutation spéciale, très utilisée pour établir des résultats en Algèbre de Kronecker. Elle se définit comme suit.

Définition 20 *(Mélange parfait) Soit L un entier et p, q, deux de ses diviseurs tels que $L = pq$. On définit la permutation $\sigma^{(p,q)}$, appelée mélange parfait (ou perfect shuffle en anglais), par*

$$\sigma^{(p,q)} : \{1, ..., L\} \longrightarrow \{1, ..., L\}$$
$$a = (i-1) \times p + j \longmapsto b = (j-1) \times q + i \text{ avec } 1 \leq j \leq p.$$

□

Observons que $\sigma^{(p,q)} \circ \sigma^{(q,p)} = \varepsilon$ où ε dénote la permutation identité. En d'autres termes, on a ${\sigma^{(p,q)}}^{-1} = \sigma^{(q,p)}$. De plus, pour un *mélange parfait* $\sigma^{(p,q)}$, la matrice de

permutation correspondante est dénotée par $S^{(q,p)}$. Ainsi, pour $x \in \mathcal{R}^L$, le vecteur défini par $y = xS^{(q,p)}$ est tel que $y_i = x_{\sigma^{(p,q)}(i)}$, $1 \leq i \leq L$. Enfin, exploitant le fait que les matrices de permutations sont orthogonales, on a les lemmes suivants [41].

Lemme 1 *Si A_p est une matrice carrée d'ordre p, et I_q la matrice identité d'ordre q, alors on a*

$$S^{(p,q)}(A_p \otimes I_q)S^{(q,p)} = I_q \otimes A_p.$$

Proposition 1 *Si A_p et B_q sont deux matrices carrées d'ordre p et q respectivement, alors on a*

$$A_p \otimes B_q = S^{(q,p)}(B_q \otimes A_p)S^{(p,q)}.$$

Ce dernier résultat illustre la notion de pseudo-commutativité précédemment évoquée. Par ailleurs, manipuler les *facteurs normaux* implique l'usage de vecteurs de taille $n_1 n_2 ... n_N$ dont les composantes peuvent par conséquent être indexés avec des N-uplets de la forme $(i_1, i_2, ..., i_N)$, où $1 \leq i_s \leq n_s, s = 1, ..., N$. Le lien avec la correspondance ordinale peut être établie par la relation suivante:

$$(i_1, i_2, ..., i_N) \underset{lex}{\overset{ord}{\longrightarrow\atop\longleftarrow}} (i_1 - 1)r_1 + (i_2 - 1)r_2 + ... + (i_N - 1)r_N + 1, \tag{3.3}$$

où $r_s = n_{s+1} n_{s+2} ... n_N$.

Si p est un quelconque diviseur de $n_1 ... n_N$, alors il existe N entiers non nuls $d_i, i = 1, ..., N$ tels que $p = d_1 d_2 ... d_N$ et d_i divise n_i pour tout $i \in \{1, ..., N\}$. Etant donné p et $(n_1, ..., n_N)$, il est évident qu'il y a en général plusieurs décompositions possibles. En associant l'entier p à l'une quelconque de ses décompositions $(d_1, ..., d_N)$, nous dérivons une indexation multidimensionnelle similaire à la précédente pour p objets. Le lecteur devrait garder ce principe dans l'esprit, car il sera à la base du formalisme utilié pour l'expression de nos algorithmes et aussi le point de départ de notre ordonnancement parallèle.

Enfin, pour un entier non nul n, et $I = \{i_1, ..., i_d\}$ un sous-ensemble de $\{1, ..., n\}$ tel que $1 \leq i_k < i_{k+1} \leq d = |I|$ pour tout $k \in \{1, ..., d-1\}$, on note $e_I = [e_{i_1}, e_{i_2}, ..., e_{i_d}]$, où e_i est le i^{th} vecteur colonne de la base canonique de $\mathcal{R}^n$. Ainsi, pour une matrice A d'ordre $n \times n$, si $I = \{i_1, ..., i_d\}$, on a $Ae_I = [Ae_{i_1}, Ae_{i_2}, ..., Ae_{i_d}]$, ce qui correspond à la matrice A réduite aux colonnes sélectionnées.

3.4 Formulations explicites de la multiplication

Etant donné N matrices carrées $A^{(i)}$ d'ordre n_i, $i = 1, ..., N$, et un vecteur $x \in \mathcal{R}^L$ avec $L = n_1 ... n_N$, nous considérons la multiplication $x(A^{(1)} \otimes ... \otimes A^{(N)})$. Commençons par remarquer que si $z = x(A^{(1)} \otimes ... \otimes A^{(N)})$, alors pour un index donné $(i_1, i_2, ..., i_N)$, on a

$$z(i_1, i_2, ..., i_N) = x(A^{(1)} e_{i_1} \otimes A^{(2)} e_{i_2} \otimes ... \otimes A^{(N)} e_{i_N}). \tag{3.4}$$

Ainsi, étant donné N sous-ensembles non vides $Q_i \subseteq \{1, ..., n_i\}$, $i = 1..., N$, si on note $z(Q_1, Q_2, ..., Q_N) = \{z(i_1, i_2, ..., i_N), i_k \in Q_k, k = 1, ..., N\}$, alors

$$z(Q_1, Q_2, ..., Q_N) = x(A^{(1)} e_{Q_1} \otimes A^{(2)} e_{Q_2} \otimes ... \otimes A^{(N)} e_{Q_N}). \tag{3.5}$$

Cette relation est particulièrement utile pour exprimer les portions de calcul dans une subdivision statique de la multiplication globale. Nous présentons maintenant une approche récurrente de calcul. Du fait qu'on a la relation

$$\otimes_{i=1}^{N} A^{(i)} = \prod_{s=1}^{N} (I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}), \tag{3.6}$$

la récurrence

$$\begin{cases} V^{(N+1)} = x \\ V^{(s)} = V^{(s+1)}(I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}) & : 1 \leq s \leq N \end{cases} \tag{3.7}$$

aboutit naturellement à $V^{(1)} = x \otimes_{i=1}^{N} A^{(i)}$.

Puisque les vecteurs $V^{(s)}$ sont de taille $L = n_1 ... n_N$, on peut considérer l'indexation $(i_1, ..., i_N)$, où $1 \leq i_s \leq n_s, 1 \leq s \leq N$. Avec cet adressage, on a le résultat suivant:

Proposition 2 *Soit $i \in \{1, ..., L\}$ avec $lex(i) = (i_1, ..., i_N)$. Alors, la i^{eme} composante de $V^{(s)}$ est obtenue de $V^{(s+1)}$ par la relation*

$$V^{(s)}(i) = \sum_{t=1}^{n_s} A^{(s)}(t, i_s) V^{(s+1)}(j),$$

où $j = pos(i_1, ..., i_{s-1}, t, i_{s+1}, ..., i_N)$.

Preuve. On peut écrire $I_L = \sum_{j=1}^{L} e_j^L e_j^{L^T}$ à cause de $e_j^L = \otimes_{s=1}^{N} e_{j_s}^{n_s}$ et $e_j^{L^T} = \otimes_{s=1}^{N} e_{j_s}^{n_s T}$ où $j = pos(j_1, ..., j_N)$. Ainsi, on a :

$$\begin{aligned} V^{(s)}(i) &= (V^{(s+1)}[\sum_{j=1}^{L} e_j^L e_j^{L^T}][I_{n_1} \otimes ... \otimes I_{n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1}} \otimes ... \otimes I_{n_N}])e_i^L \\ &= \sum_{j=1}^{L} (V^{(s+1)} e_j^L)[(\otimes_{p=1}^{s-1}(e_{j_p}^{n_p T} I_{n_p} e_{i_p}^{n_p})) \otimes (e_t^{n_s T} A^{(s)} e_{i_s}^{n_s}) \otimes (\otimes_{p=s+1}^{N}(e_{j_p}^{n_p T} I_{n_p} e_{i_p}^{n_p}))] \\ &= \sum_{j=1}^{L} (V^{(s+1)}(j))[(\prod_{p=1}^{s-1} \delta_{i_p j_p})(A^{(s)}(t, i_s))(\prod_{p=s+1}^{N} \delta_{i_p j_p}) \\ &= \sum_{t=1}^{n_s} V^{(s+1)}(j) A^{(s)}(t, i_s), \end{aligned}$$

où $j = pos(i_1, ..., i_{s-1}, t, i_s, ..., i_N)$. □

Grâce à ce résultat, la récurrence (3.7) se traduit explicitement par l'algorithme suivant.

For $(i_1, ..., i_N) = (1, ..., 1)$ **to** $(n_1, ..., n_N)$ **do**
$V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) \leftarrow \sum_{t=1}^{n_s} A^{(s)}(t, i_s) V^{(s+1)}(i_1, ..., i_{s-1}, t, i_{s+1}, ...i_N)$
end do

Alg. 1 : Etape de récurrence de la multiplication

Observons que de telles étapes impliquent $n_1 n_2 ... n_N \times n_s$ multiplications flottantes, et par conséquent, l'algorithme complet effectue la multiplication souhaitée avec la complexité classique $(n_1 + ... + n_N) \times n_1 n_2 ... n_N$.

Considérons le cas particulier de matrices de même taille ($n_i = n$, $i = 1, ..., N$). Si Ψ dénote la matrice de permutation correspondant à la permutation $\sigma^{(n^{N-1}, n)}$ qui à $x = (i-1)n^{N-1} + j$ associe $y = (j-1)n + i$ (on parle de *mélange parfait*), on a

$$\Psi^{N-s}(\overbrace{I_n \otimes ... \otimes I_n}^{s-1} \otimes A^{(s)} \otimes \overbrace{I_n \otimes ... \otimes I_n}^{N-s})\Psi_t^{N-s} = \overbrace{I_n \otimes ... \otimes I_n}^{N-1} \otimes A^{(s)} \tag{3.8}$$

où $\Psi_t = \Psi^T = S^{(n, n^{N-1})} = \Psi^{-1}$. Ce résultat découle de la propriété de speudo-commutativité qu'apporte l'usage des matrices de permutations basées sur les *mélanges parfaits* [142]. On obtient ainsi la récurrence adaptée suivante.

$$\begin{cases} V^{(N+1)} = x \\ V^{(s)}\Psi = V^{(s+1)}(I_{n^{N-1}} \otimes A^{(s)}) \quad : 1 \leq s \leq N \end{cases} \tag{3.9}$$

Du fait que $\sigma^{(n^{N-1}, n)}(i_1, i_2, ..., i_{N-1}, i_N) = (i_N, i_1, i_2, ..., i_{N-1})$, on a l'algorithme suivant.

For $(i_1, ..., i_N) = (1, ..., 1)$ **to** $(n_1, ..., n_N)$ **do**
$V^{(s)}(i_N, i_1, i_2, ..., i_{N-1}) \leftarrow \sum_{t=1}^{n_s} A^{(s)}(t, i_N) V^{(s+1)}(i_1, i_2, ..., i_{N-1}, t)$
end do

Alg. 2 : Etape de la récurrence adaptée

De manière générale, nous allons considérer le schéma global suivant:

For $s \leftarrow N$ **downto** 1 **do**
  For $(i_1, ..., i_N) = (1, ..., 1)$ **to** $(n_1, ..., n_N)$ **do**
  $V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) \leftarrow \sum_{t=1}^{n_s} A^{(s)}(t, i_s) V^{(s+1)}(i_1, ..., i_{s-1}, t, i_{s+1}, ...i_N)$
  end do
end do

Alg. 3 : Schéma global de la multiplication

Nous allons maintenant analyser les diverses implémentations possibles de cet algorithme.

3.5 Implémentations séquentielles

3.5.0.1 Version directe

Observons que

$$\begin{aligned} pos(i_1,\cdots,i_s,\cdots,i_N) &= pos(i_1,\cdots,i_{s-1},\mathbf{1}) + pos(\mathbf{1},i_s,\mathbf{1})) + pos(\mathbf{1},i_{s+1},\cdots,i_N). \\ pos(i_1,\cdots,t,\cdots,i_N) &= pos(i_1,\cdots,i_s,\cdots,i_N) + (t-i_s)\times n_{s+1}\cdots n_N. \end{aligned} \tag{3.10}$$

On obtient donc une implémentation directe de Alg. 3 avec trois niveaux de boucles imbriquées.

For $s \leftarrow N$ **downto** 1 **do**
  $r \leftarrow n_{s+1}\cdots n_N$
  $I \leftarrow 1$
  For $i \leftarrow 1$ **to** $n_1 n_2 \cdots n_{s-1}$ **do**
    For $j \leftarrow 1$ **to** n_s **do**
      For $k \leftarrow 1$ **to** $n_{s+1}\cdots n_{N-1}n_N$ **do**
        $V^{(s)}(I) \leftarrow \sum_{t=1}^{n_s} A^{(s)}(t,j)V^{(s+1)}(I+(t-j)r)$
        $I \leftarrow I+1$
      End do
    End do
  End do
End do

Alg. 4 : Implémentation directe de la multiplication

Cette version présente deux inconvénients majeurs. Le premier concerne les défauts de cache qui pourraient résulter des accès de distants impliqués par les dépendances $I \leftarrow I + (t-j)r$ pour des grandes valeurs du pas r. Ce problème peut être résolu à l'aide des permutations spéciales appelées *mélanges parfaits* dont le rôle est de regrouper les composantes devant intervenir les mêmes calculs. Le deuxième inconvénient vient du fait que les entrées nulles de matrices ne sont pas explicitement prises en compte, c'est à dire que le nombre de multiplications que l'on effectue est le même quelle que soit la configuration des matrices. Sachant qu'une entrée nulle d'une matrice implique un pourcentage important de calculs inutiles (puisque le résultat est nul et n'influe donc pas sur la valeur obtenue), il serait très bénéfique de disposer d'une version qui détecte le plus tôt possible de telles situations et évite ensuite les calculs concernés.

3.5.1 Version optimale en nombre d'opérations utiles

Pour obtenir un algorithme qui optimise le nombre d'opérations utiles, il suffit d'inverser les boucles de manière à utiliser complètement chaque coefficient d'une matrice

avant de passer à la suivante. On propose ce qui suit.

$r \leftarrow 1$
$m \leftarrow n_1 n_2 ... n_N$
$U \leftarrow x$
For s$\leftarrow N$ **downto** 1 **do**
$V \leftarrow 0$
$m \leftarrow m/n_s$
For $t \leftarrow 1$ **to** n_s **do**
  For $j \leftarrow 1$ **to** n_s **do**
    If $(A^{(s)}(t,j) \neq 0)$ **then**
      For $k \leftarrow 1$ **to** m **do**
        For $\ell \leftarrow 1$ **to** r **do**
          $i = \ell + (j-1)r + (k-1)rn_s$
          $V[i] \leftarrow V[i] + A^{(s)}(t,j) \times U[i + (t-j) \times r]$
        end do
      end do
    end if
  end do
end do
$U \leftarrow V$
$r \leftarrow rn_s$
end do
$z \leftarrow V$

Alg. 5: Implémentation optimale en nombre d'opérations.

Notons que la boucle la plus interne accède de manière contigüe aux composantes du vecteur droit. Par conséquent, une bonne vectorisation est envisageable.

Toutefois, une implémentation directe de cet algorithme utiliserait deux vecteurs qui joueraient alternativement les rôles de vecteur gauche et de vecteur droit. Nous allons maintenant montrer qu'on peut éviter cet usage de deux vecteurs de tailles $n_1 n_2 \cdots n_N$.

3.5.2 Version optimale en espace mémoire

Remarquons que les index $(i_1, ..., t, ..., i_N)$ et $(i_1, ..., i_s, ..., i_N)$ adressent la même composante si et seulement si on a $t = i_s$. Par conséquent, les conflits d'accès pourraient être évités en utilisant un vecteur supplémentaire de taille n_s qui servirait à sauvegarder, pour chaque $(i_1, ..., i_{s-1}, ., i_{s+1}, ..., i_N)$, les valeurs $V^{(s)}(i_1, ..., i_{s-1}, t, i_{s+1}, ..., i_N)$, pour $t = 1, ..., n_s$. Puisque ce vecteur sera utilisé pour toutes les étapes , il devrait être de longueur $max\{n_i\}_{i=1}^{N}$, et constitue le seul vecteur supplémentaire requis par l'algorithme. Enfin, en utilisant la relation

$$pos(i_1, ..., i_s, i_{s+1}, ..., i_N) = [(k-1)n_s + (i_s - 1)]w_s + pos(1, ..., 1, i_{s+1}, ..., i_N), \quad (3.11)$$

où $k = pos(i_1, ..., i_{s-1}, 1, ..., 1)$, on obtient l'algorithme recherché qui s'exprime ainsi qu'il suit.

```
V ← x
ℓ ← n_1 n_2 ... n_N
r ← 1
For s ← N downto 1 do
  ℓ ← ℓ/n_s
  For k ← 1 to ℓ do
    For i ← 1 to r do
     For t ← 1 to n_s do U[t] ← V[((k−1)n_s+t−1)r+i]
      For j ← 1 to n_s do
        scal ← 0
       For t ← 1 to n_s do scal ← scal+A^(s)(t, j) × U[t]
        V[((k−1)n_s+j−1)r+i] ← scal
      end do
    end do
  end
  r ← r n_s
end do
z ← V
```

Alg. 6 : Implémentation optimale en espace mémoire.

Cet algorithme calcule efficacement le produit vecteur-matrice avec un espace mémoire minimal.

Tous les schémas séquentiels que nous venons de présenter sont écrits par rapport à un aspect que l'on cherche à mieux prendre en compte. Toutefois, le nombre total d'opérations semble important, surtout lorsqu'on sait que la multiplication en question intervient très souvent comme une partie des opérations effectuées dans une boucle de calcul. De plus, la mémoire impliquée dans le stockage des données est importante et les risques de défauts de cache sont importants. En conséquence, concevoir un schéma d'algorithme parallèle optimal en nombre d'opérations et espace mémoire constitue un problème important.

3.6 Ordonnancement parallèle

Rappelons que notre objectif est le calcul en parallèle de $z = x(A^{(1)} \otimes ... \otimes A^{(N)})$, où $A^{(i)}$ sont des matrices carrées d'ordre $n_i, i = 1, ..., N$ et x un vecteur de $\mathcal{R}^L$ avec $L = n_1 n_2 ... n_N$. Etant donné p processeurs (p diviseur de L), on procède comme suit. Premièrement, on calcule la décomposition de p en N entiers non nuls $d_i, i = 1, ..., N$ tels que $p = d_1 ... d_N$ et d_i divise $n_i, i = 1, ..., N$ (nous donnerons plus tard un algorithme qui réalise cette décomposition). Ensuite, pour chaque $i \in \{1, ..., N\}$, on considère une partition $Q_{ij}, j = 1, ..., d_i$ de $Q_i = \{1, ..., n_i\}$. On obtient ainsi une allocation des tâches aux processeurs définie comme suit.

$$\overbrace{(w_1, w_2, ..., w_N)}^{label\ du\ processeur} \leftarrow \overbrace{z(Q_{1w_1}, Q_{2w_2}, ..., Q_{Nw_N})}^{portion\ allouée}, \tag{3.12}$$

où $1 \leq w_i \leq d_i, i = 1, ..., N$.
Remarquons que le calcul d'une telle portion peut se réaliser de façon similaire à celle de la multiplication globale, à travers la mise à jour récursive des portions dénotées par $V^{(s)}(Q_{1w_1}, Q_{2w_2}, ..., Q_{Nw_N})$, $1 \leq s \leq N+1$. Ceci conduit à *Alg. 7* pour le processeur $(w_1, w_2, ..., w_N)$.

For $(i_1, ..., i_N) \in Q_{1w_1} \times Q_{2w_2} \times ... \times Q_{Nw_N}$ **do**
$V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) \leftarrow \sum_{t=1}^{n_s} A^{(s)}(t, i_s) V^{(s+1)}(i_1, ..., i_{s-1}, t, i_{s+1}, ...i_N)$
end do

Alg. 7: Etape de la récurrence subdivisée.

Une façon directe de partitionner les ensembles Q_i consiste à effectuer une subdivision bloc ou cyclique en d_i sous-ensembles. On obtient ainsi une partition équitable, qui, comme nous le verrons plus tard, correspond à un algorithme parallèle totalement équilibré. De plus, on a une formulation générique des indices d'itération. Nous allons maintenant présenter l'algorithme de la décomposition.

Etant donné une séquence de N entiers non nuls $(n_1, ..., n_N)$, et un entier p qui divise $n_1 n_2 ... n_N$, l'algorithme suivant réalise la décomposition souhaitée.

$(d_1, d_2, ..., d_N) \leftarrow (1, 1, ..., 1)$
$i \leftarrow 1$; $c \leftarrow p$
while $(c > 1)$ **do**
  $d_i \leftarrow pgcd(c, n_i)$
  $c \leftarrow \frac{c}{d_i}$
  $i \leftarrow i + 1$
end while

Alg. 8: Algorithme de la décomposition.

La preuve de *Alg. 4* découle de sa propriété invariante

$$(c \times \prod_{i=1}^{N} d_i = p) \wedge (\forall i \in \{1, ..., N\}\ d_i \text{ divise } n_i), \tag{3.13}$$

La Table 3.1 présente la trace l'algorithme avec $p = 60$ et $(n_1, n_2, n_3, n_4) = (10, 9, 8, 16)$.

i	c	(d_1, d_2, d_3, d_4)
1	60	(1, 1, 1, 1)
2	6	(10, 1, 1, 1)
3	2	(10, 3, 1, 1)
4	1	(10, 3, 2, 1)

TAB. 3.1 – *Trace de Alg. 8 sur* $(10, 9, 8, 16)$ *avec* $p = 60$.

Comme nous l'avons précédemment évoqué, il existe plusieurs décompositions possibles. Mais, comme nous allons le voir, elles peuvent théoriquement impliquer un coût équivalent. Toutefois, on aurait dans certains cas besoin d'une décomposition ayant une somme minimale. Ce problème combinatoire semble difficile [58], car une de ses instances assez simple correspond à la décomposition en produits de facteurs premier d'un entier [18]. Heureusement, du fait que les matrices sont en général de petites tailles, des heuristiques devraient fournir des décompositions satisfaisantes.

3.7 Analyse des communications

Lemme 2 *A une étape s, $1 \leq s \leq N$, deux processeurs $\alpha = (\alpha_1, \cdots, \alpha_N)$ et $\beta = (\beta_1, \cdots, \beta_N)$ communiquent si et seulement si $\alpha_s \neq \beta_s$ et $\alpha_i = \beta_i$ pour $i = 1, \cdots, N$, $i \neq s$.*

Preuve. De *Alg. 7*, on peut observer que la mise à jour du vecteur correspondant à $V^{(s)}(Q_{1w_1}, \cdots, Q_{s-1,w_{s-1}}, Q_{sw_s}, Q_{s+1,w_{s+1}}, \cdots Q_{Nw_N})$, $1 \leq s \leq N$ requiert uniquement les valeurs de $V^{(s+1)}(Q_{1w_1}, \cdots, Q_{s-1,w_{s-1}}, Q_s, Q_{s+1,w_{s+1}}, \cdots, Q_{Nw_N})$. Ainsi, puisqu'à la fin de l'étape $s+1$, chaque processeur $w = (w_1, \cdots, w_N)$ détient la portion $V^{(s+1)}(Q_{1w_1}, \cdots, Q_{s-1,w_{s-1}}, Q_{sw_s}, Q_{s+1,w_{s+1}}, \cdots Q_{Nw_N})$, on obtient le résultat en considérant la relation $Q_s = Q_{s1} \cup Q_{s2} \cup \cdots \cup Q_{sN}$. □

Remarque 1 *De Alg. 7, on peut remarquer quà l'étape s, les portions reçues par un processeur sont suffisantes pour mettre à jour tout le vecteur $V^{(s)}(Q_{1w_1}, \cdots, Q_s, \cdots Q_{Nw_N})$ au lieu du sous-vecteur $V^{(s)}(Q_{1w_1}, \cdots, Q_{s,w_s}, \cdots Q_{Nw_N})$. Le seul besoin supplémentaire vient du fait est qu'il faudrait avoir la totalité de la matrice $A^{(s)}$ au lieu de la sous-matrice $A^{(s)}e_{Q_{s,w_s}}$. Ce faisant, on elimine la nécessité d'une communication à la fin de l'étape $s-1$ au prix d'un calcul redondant. Une telle décision devrait être prise selon un critère donné par le résultat suivant, où $\psi(p)$ représente le nombre de communications nécessaire pour diffuser un message à p processeurs, et $\phi(\ell)$ le coût de communication d'un message de longueur ℓ.*

Proposition 3 *A une étape donnée s, $1 \leq s \leq N$, on obtient un gain en temps en effectuant un calcul redondant au lieu d'opérer une communication si on a*

$$\psi(d_s)\phi(\frac{L}{d}) \geq n_s \frac{L}{d}\tau, \tag{3.14}$$

où τ est le coût d'une opération élémentaire.

Preuve. Le résultat vient du fait que $\psi(d_s)\phi(\frac{L}{d})$ représente au temps nécessaire pour diffuser un message de longueur $\frac{L}{d}$ à d_s processeurs, et $n_s\frac{L}{d}\tau$ est le temps des calculs correspodants tels que décrits dans *Alg. 7*.
□

Corollaire 1 *Si on considère le cas $\psi(p) = p$ et $\phi(\ell) = \alpha\ell + \beta$, alors la condition (3.14) devient*

$$(\alpha - \frac{n_s}{d_s}\tau)\frac{L}{d} \leq \beta \tag{3.15}$$

Cette condition pourrait être incluse dans l'algorithme SPMD afin de permettre de faire le bon choix entre une communication ou un calcul redondant. Notons que dans ce second cas, on aurait besoin de plus de mémoire pour stocker les résultats des calculs supplémentaires. Dans tous les cas, les communications devraient être réduites le plus possible pour assurer une bonne efficacité de l'algorithme.

Pour une topologie donnée, si $\psi(p)$ désigne le nombre d'étapes de communications nécessaires pour une diffusion à p processeurs $(cste, log(p), \sqrt{p}, p)$, et $\phi(\ell)$ le coût de l'envoi d'un message de volume ℓ, alors on a le résultat suivant.

Théorème 10 *Considérant d processeurs, où d est un diviseur de $L = n_1 \cdots n_N$, l'algorithme* SPMD *Alg. 7 exécuté avec la décomposition $d = d_1 \cdots d_N$, implique un surcoût de communication $c(d)$ donné par*

$$c(d) = (\psi(d_1) + \psi(d_2) + \cdots + \psi(d_N))\phi(\frac{L}{d}). \tag{3.16}$$

□

Preuve. Considérant le processus complet incluant toutes les étapes de la récurrence, on obtient le résultat du fait que $|\Gamma(s, w)| = d_s$, et qu'avec un partitionnement équilibré de chaque ensemble $Q_s = \{1, \cdots, n_s\}$, on a

$$|V^{(s)}(Q_{1w_1}, \cdots, Q_{Nw_N})| = \prod_{s=1}^{N} |Q_{s,w_s}| = \prod_{s=1}^{N} \frac{n_s}{d_s} = \frac{\prod_{s=1}^{N} n_s}{\prod_{s=1}^{N} d_s} = \frac{L}{d}. \tag{3.17}$$

□

Le théoreme 10 montre que, quelle que soit la topologie considérée, le coût des communications dépend de la quantité $\psi(d_1) + \psi(d_2) + \cdots + \psi(d_N)$ qui elle même dépend de la décomposition $d = d_1 d_2 \cdots d_N$.

Dans [142], il est établi que si $d_i \in \{1, n_i\}$, alors l'algorithme sera très régulier et fortement vectorisable. Mais, le problème de l'existence d'une telle décomposition est NP-complet du fait son équivalence avec le problème du PRODUIT DE SOUS-ENSEMBLE (SUBSET PRODUCT) [58]. Même en considérant le cas de la FFT multidimensionnelle qui implique des matrices dont les tailles sont des puissances de deux [9], le probléme reste NP-complet du fait de son équivalence avec le problème de la SOMME DE SOUS-ENSEMBLE (SUBSET SUM) [58] ($2^d = \prod_{i \in I} 2^{n_i} \leftrightarrow d = \sum_{i \in I} n_i$). Toutefois, en fonction de l'instance du problème à résoudre, les informations sur le type des matrices pourraient arranger les choses.

D'une manière génerale (excepté le cas logarithmique où toutes les décompositions sont équivalentes), les performances seraient optimales si la somme $d_1 + d_2 + \cdots + d_N$ est minimale. Nous allons maintenant étudier ce problème qu'on désignons par DSM (Décomposition de Somme Minimale).

Nous formulons le problème de décision associé à la DSM ainsi qu'il suit.

- Etant donné une séquence $(n_1, n_2, \cdots, n_N)$, un entier d, et un entier λ, existe-t-il une décomposition $d = d_1 d_2 \cdots d_N$ avec d_i un diviseur de n_i,$i = 1 \cdots N$, telle que $d_1 + d_2 + \cdots + d_N \leq \lambda$.

Ce problème est évidemment dans NP, et est aussi difficile que le problème du test de primalité.

Proposition 4 *Un entier donné d est non premier si et seulement si il existe une décomposition par rapport à la séquence (d, d) avec une somme inférieure à d.*

Preuve. A l'exception des décompositions triviales $(1, d)$ et $(d, 1)$, toute autre décomposition (d_1, d_2) (si elle existe) est telle que $d_1 + d_2 < d_1 d_2 = d$. Par conséquent, il existe une décomposition avec une somme inférieure à d si et seulement si il existe une décomposition $(d_1, d_2) \notin \{(1, d), (d, 1)\}$, soit $d_1, d_2 \notin \{1, d\}$. En d'autres termes, d_1 et d_2 sont des facteurs non triviaux de d. □

Nous allons maintenant montrer que la DSM est aussi difficile que la factorisation des entiers (pour une vue des développements récents sur la factorisation des entiers, voir [18]).

Proposition 5 *Etant donné un entier d, si on considère la séquence $\overbrace{(d, \cdots, d)}^{d \quad termes}$ $(N = d)$, alors la décomposition $d = d_1 d_2 \cdots d_N$ est de somme minimale si et seulement si chaque d_i est un facteur premier de d.*

Preuve. Notons premièrement que d ne peut avoir plus de d facteurs premiers. Par suite, toute décomposition $d_1 d_2 \cdots d_d$ est telle que $d_i = 1$ pour un certain $i \in \{1, \cdots, d\}$. Par conséquent, si une décomposition $d_1 d_2 \cdots d_d$ contient un facteur non premier $d_j = ab$,$a, b > 1$, alors la nouvelle décomposition obtenue de la précédente en remplacant d_i par a, et d_j par b, est de somme plus petite. En effet, on a $a + b < 1 + ab = d_i + d_j$. □

En se basant sur les précédentes analyses, on est en droit d'énoncer la conjecture suivante.

Conjecture 1 *Le problème de la décomposition de somme minimale est NP-complet.*

Remarque 2 *En pratique, la résolution du problème de la décomposition optimale impliquerait des valeurs de n_i et p pas très grandes. De ce fait, une exploration exhaustive des différentes décompositions pourrait des fois être envisageable, de même que le recours à des méthodes heuristiques qui pourraient fournir des solutions assez proches de l'optimum.*

3.7.1 Algorithme par raffinement successif

L'idée ici consiste à partir d'une décomposition quelconque et ensuite appliquer des raffinements récursifs. Rappelons que pour une séquence donnée $(n_1, \cdots, n_N)$ et un entier donné d qui divise $n_1 n_2 \cdots n_N$, une décomposition directe peut être obtenue à l'aide de la récurrence suivante:

$$\begin{cases} c_0 = d & & \\ d_i = pgcd(c_{i-1}, n_i) & : & 1 \leq i \leq N \\ c_i = \frac{c_{i-1}}{d_i} & : & 1 \leq i < N \end{cases} \tag{3.18}$$

qui se justifie par le lemme suivant:

Lemme 3 *Soit (a, b) une paire d'entiers positifs et soit un entier d un entier non nul. Si d divise ab alors $\frac{a}{pgcd(d,a)}$ divise b.*

Preuve. En posant $k = pgcd(d, a)$, on a $d = kd'$ et $a = ka'$, avec $pgcd(d', a') = 1$. Puisque d' divise $a'b$, il vient que d' divise b. □

Si σ est une permutation quelconque de l'ensemble $\{1, ..., N\}$, alors la récurrence (3.18) reste valable si on remplace d_i par $d_{\sigma(i)}$, et n_i par $n_{\sigma(i)}$. Ce aspect est due à la commutativité de la multiplication dans N, et constitue une piste (par un choix approprié de σ) pour la conception des algorithmes calculant des décompositions ayant des proprétés souhaitées.

Maintenant que nous avons un algorithme qui fournit une première décomposition, nous allons passer à l'étude du problème du raffinement.

Si $c_{ij} = pgcd(d_i, \frac{n_j}{d_j})$, $1 \leq i, j \leq N$, $i \neq j$, alors en remplaçant d_i par $\frac{d_i}{c_{ij}}$, et d_j par $c_{ij}d_j$, on améliore la décomposition si $\frac{d_i}{c_{ij}} + c_{ij}d_j < d_i + d_j$, c'est à dire

$$(1 - c_{ij})(d_i - c_{ij}d_j) < 0. \tag{3.19}$$

En d'autres termes, on doit avoir $c_{ij} > 1$ et $d_i > c_{ij}d_j$. On a donc l'algorithme suivant.

```
{ Décomposition initiale}
For i ← 1 to N do
  d_i ← gcd(d, n_i)
  d ← d / d_i
enddo
{ Raffinements successifs }
For i ← 1 to N do
  For j ← 1 to N do
    c ← pgcd(d_i, n_j / d_j)
    if (c > 1) ∧ (d_i > c_ij d_j) then
      d_i ← d_i / c
      d_j ← c d_j
    endif
  enddo
enddo
```

Alg. 9: Heuristique par raffinement successif.

Pour un meilleur raffinement, on pourrait considérer un diviseur strict de $c = pgcd(d_i, \frac{n_j}{d_j})$, mais ceci augmenterait la complexité de l'algorithme. Illustrons le comportement de *Alg. 9* avec $d = 60$ et $(10, 6, 4, 2)$. La décomposition initiale est $(10, 6, 1, 1)$. Le raffinement appliqué avec $d_1 = 10$ conduit à la séquence $(5, 6, 2, 1)$ et ensuite, raffinement avec $d_2 = 6$ donne $(5, 3, 4, 1)$. L'algorithme termine avec la décomposition $(5, 3, 4, 1)$ qui a une somme égale à 13, ce qui est une amélioration de la décomposition initiale qui elle a une somme égale à 18. Notons que la meilleure décomposition est $(5, 3, 2, 2)$ avec une somme égale à 12. On aurait pu obtenir cette solution optimale si le raffinement appliqué avec $d_3 = 4$ était effectué avec le diviseur 2 de $c = pgcd(4, 2) = 4$ au lieu de c lui-même. Par ailleurs, une variante de l'algorithme consisterait à appliquer le raffinement jusqu'à ce qu'on ce qu'on n'obtienne plus d'amélioration.

3.7.1.1 Etude expérimentale de l'algorithme

La figure Fig. 3.1 présente le comportement de l'algorithme avec la séquence expérimentale $(10, 12, 8, 15, 8, 9)$. Pour les valeurs de d, nous avons considéré tous les 150 diviseurs de $L = 10 \times 12 \times 8 \times 15 \times 8 \times 9 = 1036800$. Globalement, par rapport aux sommes des séquences, la moyenne relative entre la somme des décompositions obtenues et celle la décomposition optimale correpondante (i.e $\delta = (\bar{s} - s)/s$) est de $0,0645$. Ainsi, le résultat retourné par l'algorithme est de 6% proche de l'optimum.

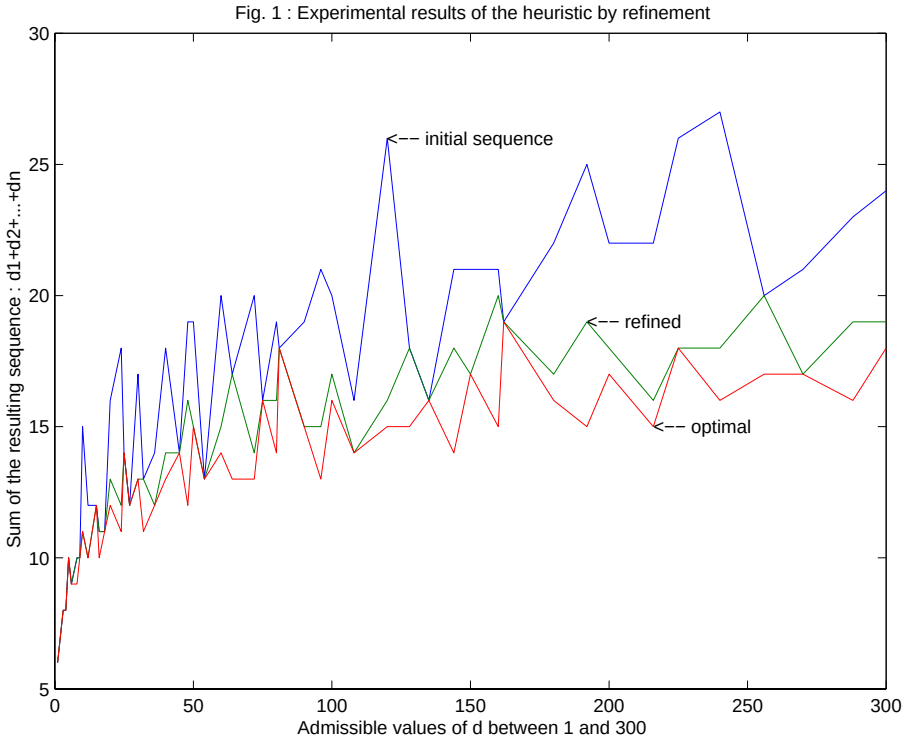


FIG. 3.1 – *Qualité expérimentale du raffinement.*

3.7.2 Algorithme glouton

Comme nous l'avons précedemment montré, le problème de la décomposition optimale est aussi difficile que celui de la factorisation d'entier. Par conséquent, il serait légitime de proposer un algorithme heuristique qui commence par la factorisation de l'entier d. Donc, étant donné une séquence $(n_1, \cdots, n_N)$ et un entier d, notre algorithme vorace commence avec l'ensemble $S = \{s_j : j = 1, \cdots, p\}$ des facteurs premiers de d tel que $d = s_1 s_2 \cdots s_p$. A chaque étape $k, 0 \leq k \leq p$, de l'algorithme, $(d_1, \cdots, d_N)$ est une décomposition de $\pi_k = d_1 \cdots d_N$, où $\pi_0 = 1$ et $\pi_p = d$ avec π_{k-1} qui est un diviseur strict de π_k, $k = 1, \cdots, p$. La sélection dans S doit être faite de manière à induire

une faible croissance de la somme $d_1 + \cdots + d_N$ de la séquence courante. A cet effet, considérons la fonction suivante:

$$\sigma(x) = \min_{j \in \{1,\cdots,N\} | x d_j \div n_j} (x d_j + \sum_{i=1\cdots N : i \neq j} d_i), \tag{3.20}$$

où $a \div b$ signifie a divise b. La fonction de sélection est alors donnée par

$$f(S) = \min_{x \in S} \sigma(x), \tag{3.21}$$

et l'indice correspondant à l'étape k sera dénoté par i_k. L'algorithme complet est le suivant:

```
{ Initialisation }
(d_1, ..., d_N) ← (1, ..., 1)
S ← primefact(d)
k ← 0
{ Processus glouton }
While S ≠ ∅ do
  k ← k + 1
  (x, i_k) ← f(S)
  S ← S − {x}
  d_{i_k} ← x d_{i_k}
enddo
```

Alg. 10 : Algorithme glouton pour la décomposition.

La table 3.2 présente la trace de l'algorithme avec $d = 60$ et $(10, 6, 4, 2)$. On commence avec $S = [3, 2, 2, 5]$ et $(d_1, d_2, d_3, d_4) = (1, 1, 1, 1)$.

k	x	i_k	(d_1, d_2, d_3, d_4)	$d_1 + d_2 + d_3 + d_4$
1	2	2	$(1, 2, 1, 1)$	5
2	2	3	$(1, 2, 2, 1)$	6
3	3	2	$(1, 6, 2, 1)$	10
4	5	1	$(5, 6, 2, 1)$	14

TAB. 3.2 – *Trace de l'algorithme glouton avec* $(10, 6, 4, 2)$ *et* $p = 60$.

La solution $(5, 6, 2, 1)$ retournée par l'algorithme est assez proche de la solution optimale $(5, 3, 2, 2)$. Le fait d'avoir manqué la solution optimale provient sans doute de l'usage précoce du facteur 2. Ceci suggère un raffinement qui consisterait à trier l'ensemble S avant d'exécuter le processus glouton.

3.7.2.1 Etude expérimentale de l'algorithme

La figure Fig. 3.2 présente le comportement de l'algorithme avec la séquence expérimentale $(10, 12, 8, 15, 8, 9)$. Pour les valeurs de d, nous avons considéré tous les 150

diviseurs de $L = 10 \times 12 \times 8 \times 15 \times 8 \times 9 = 1036800$. Dans le programme, les nombres premiers ont été pris dans l'ordre croissant. Globalement, par rapport aux sommes des séquences, la moyenne relative entre la somme des décompositions obtenues et la décomposition optimale correpondante (i.e $\delta = (\bar{s} - s)/s$) est de $0,0145$. Ainsi, le résultat retourné par l'algorithme est de 1% proche de l'optimum.

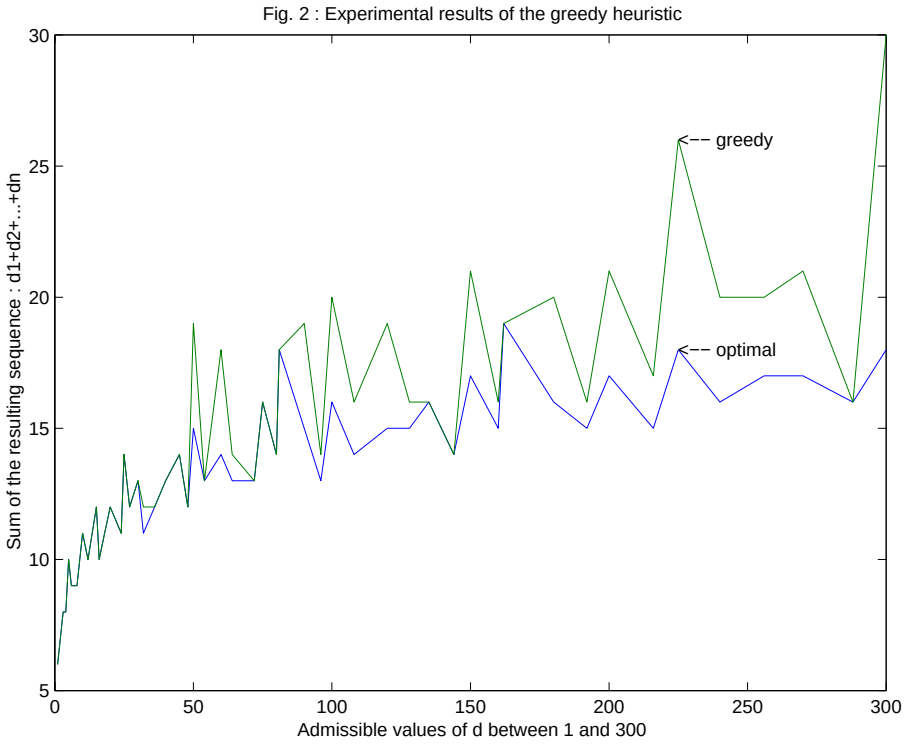


FIG. 3.2 – *Qualité expérimentale de l'algorithme glouton.*

3.7.3 Comparaison experimentale

D'après nos expérimentations, l'algorithme glouton semble globalement produire une meilleure approximation (1% contre 6%). Ceci est sans doute dû au fait que l'algorithme glouton débute avec les facteurs premiers de d pour lesquels nous avons montré qu'ils étaient liés d'une certaine manière à la solution optimale. Toutefois, lorsqu'on regarde la figure 3.3, il apparait que l'heuristique par raffinement est tout aussi bon que l'algorithme glouton, et de plus, son implémentation est beaucoup plus simple et le coût d'exécution est assez réduit. Ce dernier aspect est important, dans la mesure où nous devons réduire le surcoût de temps impliqué par la recherche d'une décomposition efficace. La figure Fig. 3.3 présente les performances relatives des deux heuristiques avec les données expérimentales précédentes.

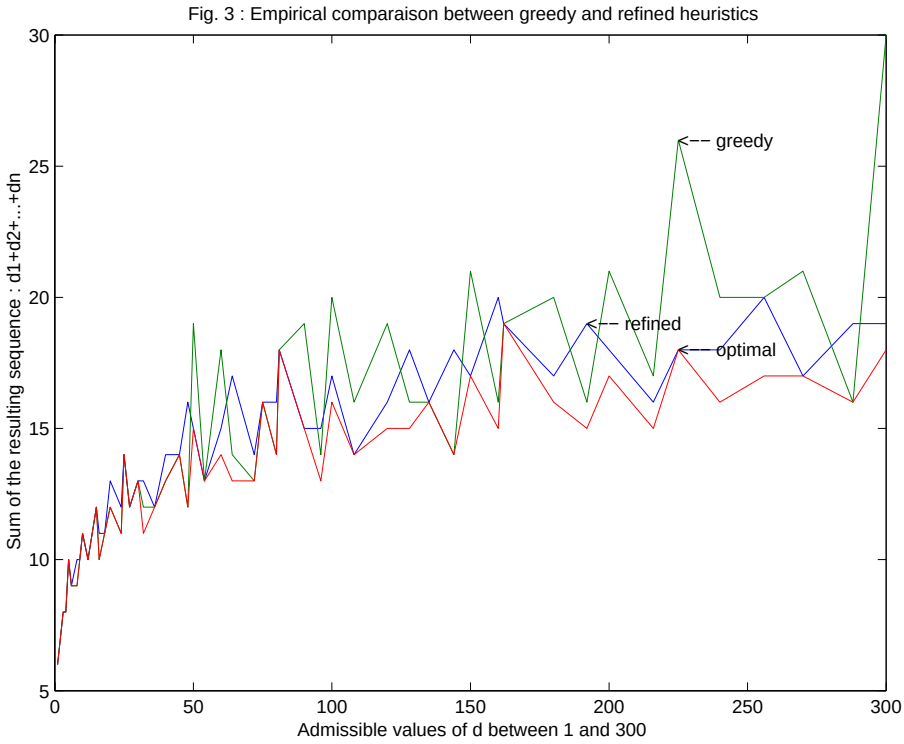


FIG. 3.3 – *Comparaison expérimentale des heuristiques.*

Nous retournons maintenant à l'algorithme de la multiplication qui est l'objet principal de cette étude.

3.8 Description de l'algorithme

Du fait que $Q_s = \bigcup_{T=1}^{p_s} Q_{sT}$, il vient de l'allocation (3.12) que chaque processeur $(w_1, \cdots, w_N)$ doit diffuser sa portion $V^{(s+1)}(Q_{1w_1}, ..., Q_{sw_s}, ..., Q_{Nw_N})$ à tous les processeurs de l'ensemble $\Gamma(w,s) = \{(w_1, \cdots, w_{s-1}, T, w_{s+1}, \cdots, w_N), T = 1, \cdots, d_s\}$. Dénotant cette opération par $diffusion(w,s)$, on obtient l'algorithme suivant pour le processeur $(w_1, ..., w_N)$.

For $s \leftarrow N$ **downto** 1 **do**
 $V^{(s)} \leftarrow 0$
 $diffusion(w,s)$
 For $(i_1, ..., i_N) \in Q_{1w_1} \times Q_{2w_2} \times ... \times Q_{Nw_N}$ **do**
  For $T = 1$ **to** d_s **do**
   $V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) \leftarrow V^{(s)}(i_1, ..., i_{s-1}, i_s, i_{s+1}, ...i_N) +$
   $+ \sum_{t \in Q_{sT}} A^{(s)}(t, i_s) V^{(s+1)}(i_1, ..., i_{s-1}, t, i_{s+1}, ...i_N)$
  end do
 end do
end do

Alg. 11: Algorithme SPMD avec diffusion locale.

Bien que cet algorithme soit assez rapide car il ne fait appel qu'à des routines de diffusion, il implique une bufferisation considérable pour stocker tous les messages reçus avant de leur utilisation. Ce problème peut être contourné en chevauchant les calculs et les communications car les messages peuvent être exploités au fur et à mesure de leur arrivée. Ainsi, non seulement on évite la bufferisation, mais aussi on peut avec des architectures adaptées, recouvrir les communications par les calculs. Toutefois, ceci nécessite un mécanisme spécial d'envoi des messages que nous allons décrire. Reprenons l'ensemble $\Gamma(w,s) = \{(w_1, ...w_{s-1}, T, w_{s+1}, ..., w_N), T = 1, ..., d_s\}$. Le mécanisme de communication globale souhaité s'exécute en d_s étapes suivant une matrice de *Hankel* pour la sélection des index des processeurs cibles. La table 3.3 illustre un exemple avec $|\Gamma(w,s)| = 5$ processeurs identifiés avec la composante T. Chaque ligne fait référence au processeur correspondant et pour chaque instant (en colonne), il fournit l'identité du processeur cible.

Si $\alpha_t(w)$ (resp. $\beta_t(w)$) représente l'identité du processeur cible (resp. source) à l'instant t, on a les relations suivantes qui se vérifient facilement.

$$\begin{cases} \alpha_1(w) &= w \\ \alpha_t(w) &= \alpha_{t-1}(w) \bmod d + 1 \quad : 1 < t \leq d \end{cases} \tag{3.22}$$

	Temps				
Id	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	2	3	4	5	1
3	3	4	5	1	2
4	4	5	1	2	3
5	5	1	2	3	4

TAB. 3.3 – *Table décrivant le mécanisme de communication globale*

$$\begin{cases} \beta_1(w) &= w \\ \beta_t(w) &= d-(d-\beta_{t-1}(w)-1) \bmod d \quad : 1<t\leq d \end{cases} \tag{3.23}$$

Ceci permet d'écrire une version révisée de *alg. 5* avec une boucle de communications (un envoi/réception pour chaque sous-ensemble Q_{iT}), au lieu d'une diffusion préalable. A cet effet, considérons les commandes suivantes. La commande **send**(T) signifie que le processeur $w = ord(w_1,\cdots,w_{s-1},w_s,w_{s+1},\cdots,w_N)$ envoie sa portion $V^{(s+1)}(Q_{1w_1},\cdots,Q_{sw_s},\cdots,Q_{Nw_N})$ au processeur $(w_1,\cdots,w_{s-1},T,w_{s+1},\cdots,w_N)$ avec l'étiquette T, tandis que la commande **recv**(T) signifie que le processeur w recoit la portion $V^{(s+1)}(Q_{1w_1},\cdots,Q_{sT},\cdots,Q_{Nw_N})$ avec le label T provenant du processeur $(w_1,\cdots,w_{s-1},T,w_{s+1},\cdots,w_N)$.

For $s \leftarrow N$ **downto** 1 **do**
  $V^{(s)} \leftarrow 0$
  $\alpha \leftarrow w$; $\beta \leftarrow w$; $d \leftarrow d_s$
  For $(i_1,\cdots,i_N) \in Q_{1w_1}\times Q_{2w_2}\times\cdots\times Q_{Nw_N}$ **do**
    $V^{(s)}(i_1,\cdots,i_{s-1},i_s,i_{s+1},\cdots i_N) \leftarrow V^{(s)}(i_1,\cdots,i_{s-1},i_s,i_{s+1},\cdots i_N)+$
      $+\sum_{t\in Q_{sw}} A^{(s)}(t,i_s)V^{(s+1)}(i_1,\cdots,i_{s-1},t,i_{s+1},\cdots i_N)$
  end do
  For $T=2$ **to** d_s **do**
    $\alpha \leftarrow \alpha \bmod d+1$
    $\beta \leftarrow d-(d-\beta-1) \bmod d$
    send(α)
    recv(β)
    For $(i_1,\cdots,i_N) \in Q_{1w_1}\times Q_{2w_2}\times\cdots\times Q_{Nw_N}$ **do**
     $V^{(s)}(i_1,\cdots,i_{s-1},i_s,i_{s+1},\cdots i_N) \leftarrow V^{(s)}(i_1,\cdots,i_{s-1},i_s,i_{s+1},\cdots i_N)+$
       $+\sum_{t\in Q_{s\beta}} A^{(s)}(t,i_s)V^{(s+1)}(i_1,\cdots,i_{s-1},t,i_{s+1},\cdots i_N)$
    end do
  end do
end do

Alg. 12 : Algorithme SPMD avec des boucles de communications.

3.9 Complexité

Rappelons que le nombre d'étapes de communication nécessaire pour effectuer une diffusion globale sur p processeurs est donné par les expressions suivantes [129]:

- $O(log(p))$ sur les arbres, hypercubes et graphes de Bruijn.
- $O(\sqrt{p})$ sur les grilles bidimensionnelles.
- $O(p)$ sur les topologies linéaires et anneaux.

Notons $\Gamma(p)$ l'expression générique de cette valeur (dans certains type d'architecture, on peut aussi imaginer $\Gamma(p) = cste$). On a alors les résultats suivants qui établissent des bornes inférieures sur les coûts de communications.

Théorème 11 *L'Algorithme Alg. 11 effectue la multiplication requise avec au plus* $O(\Gamma(p))$ *étapes de communication.* □

Preuve. A chaque étape s, chaque processeur diffuse sa portion à $|\Gamma(w,s)| = d_s$ processeurs, ce qui coûte $O(\Gamma(p))$ étapes de communications. Le nombre total d'étapes de communication est donc égal à

$$\sum_{i=1}^{N} \Gamma(d_i) \leq \Gamma(\prod_{i=1}^{N} d_i) = \Gamma(p). \tag{3.24}$$

□

Théorème 12 *Etant donné* N *matrices carrées* $A^{(i)}$ *d'ordre* $n_i, i = 1, ..., N$*, et un vecteur* $x \in \mathcal{R}^L$ *avec* $L = n_1...n_N$*, la multiplication* $x(A^{(1)} \otimes ... \otimes A^{(N)})$ *effectuée en parallèle sur* p *processeurs nécessite au moins* $\Theta(log(p))$ *étapes de communication.* □

Preuve. De la relation $z(i_1, i_2, ..., i_N) = x(A^{(1)}e_{i_1} \otimes A^{(2)}e_{i_2} \otimes ... \otimes A^{(N)}e_{i_N})$, il vient que le calcul de chaque composante du z implique tout le vecteur x. De ce fait, le graphe de communication induit est un graphe fortement connexe, ce qui implique au mieux $\Theta(log(p))$ étapes de communication. □

Notons que l'algorithme Alg. 11 (algorithme avec diffusion locale) atteint cette borne sur les topologies qui sont telles que $\Gamma(p) = log(p)$, c'est à dire les graphes complets, les hypercubes, les arbres, les topologies de Bruijn, et bien d'autres. Toutefois, du point de vue de la mémoire, la version avec diffusion requiert une mémoire de taille $(1 + Max\{d_i\}) \times L/p$, contre $3L/p$ pour la version avec chevauchement des calculs et des communications. Ce gain de mémoire est important surtout dans le cas de très grands vecteurs car le nombre d'accès disque est réduit.

3.10 Implémentation et performance

3.10.1 Implémentation

Notons premièrement que si on opère avec des partitions équilibrées, alors les composantes de $V^{(s)}(Q_{1w_1}, \cdots, Q_{sw_s}, \cdots, Q_{Nw_N})$ peuvent être stockés dans un vecteur contigu de taille

$$|Q_{1w_1} \times \cdots \times Q_{sw_s} \times \cdots \times Q_{Nw_N}| = \prod_{i=1}^{N} |Q_{iw_i}| = \prod_{i=1}^{N} \frac{n_i}{d_i} = \frac{\prod_{i=1}^{N} n_i}{\prod_{i=1}^{N} d_i} = \frac{L}{p}, \tag{3.25}$$

où $L = n_1 \cdots n_N$. Soit $c_i = \frac{n_i}{d_i}$ et considérons une partition en bloc . Alors, pour $i, 1 \leq i \leq N$ et $j, 1 \leq j \leq d_i$, on a

$$Q_{ij} = \{\alpha_i + 1, \alpha_i + 2, \cdots, \alpha_i + c_i\}, \tag{3.26}$$

où $\alpha_i = (j-1)c_i$. De plus, avec la correspondance fournie par 3.3, on a

$$ord(i_1, \cdots, t, \cdots, i_N) = ord(i_1, \cdots, i_s, \cdots, i_N) + (t - i_s) \times c_{s+1} \cdots c_N, \tag{3.27}$$

$$ord(w_1, \cdots, T, \cdots, w_N) = ord(w_1, \cdots, w_s, \cdots, w_N) + (T - w_s) \times d_{s+1} \cdots d_N. \tag{3.28}$$

Si $w = ord(w_1, \cdots, w_s, \cdots, w_N)$, alors

$$w_s = [(w-1)\mathbf{div}(d_{s+1} \cdots d_N)] \ \mathbf{mod}(d_s) + 1, \tag{3.29}$$

où **div** denote la division entière.

L'algorithme complet suivant le modèle de Alg. 12 est donc le suivant.

```
π ← 1; r ← 1; ℓ ← c_1 c_2 ··· c_N = L/p    /* c_i = n_i/d_i */
y ← x(Q_{1w_1}, Q_{2w_2}, ···, Q_{Nw_N})
For s ← N downto 1 do
  ℓ ← ℓ/c[s]
  ws ← [w div(π)] mod(d[s]) + 1; e ← (ws − 1) × c[s]
  v ← 0
  i ← 1
  For a ← 1 to ℓ do
    For j ← e + 1 to e + c[s] do
      For b ← 1 to r do
        For t ← e + 1 to e + c[s] do
          v[i] ← v[i] + A(s, t, j) y[I + (t − j)r]
        end do
        i ← i + 1
      end do
    end do
  end do
  If (ws = 1) then H ← d[s] else H ← ws − 1
  For T = ws + 1 to ws + d[s] − 1 do
    G ← mod(T − 1, d[s]) + 1
    idest ← w + (G − ws) × π
    isender ← w + (H − ws) × π
    send(y, idest, ws)
    recv(u, isender, H)
    e ← (H − 1) × c[s]
    i ← 1
    For a ← 1 to ℓ do
      For j ← 1 to c[s] do
        For b ← 1 to r do
          For t ← e + 1 to e + c[s] do
            v[i] ← v[i] + A(s, t, j) u[I + (t − j)r]
          end do
          i ← i + 1
        end do
      end do
    end do
    If (H = 1) then  H ← d[s] else H ← H − 1
  end do
  r ← r × c[s]; π ← π × d[s]
  If (s > 1) then y ← v
end do
z(Q_{1w_1}, Q_{2w_2}, ···, Q_{Nw_N}) ← v
```

Alg. 13: Implementation de l'algorithme global.

3.11 Performances

Nos tests ont été effectués sur la machine CRAY T3E à 256 processeurs cadencés à 300 MHZ. Nous avons considéré le cas de $N = 6$ matrices carrées d'ordre $n = 12$ chacune, soit $L = 2,985,984$. Les meilleures efficacités sont obtenues lorsque p est une puissance de 2 car la topologie virtuelle est celle de l'hypercube, avec une seule transmission par étape pour chaque processeur. Le tableau de performances inclut l'accélération σ (rapport entre le temps séquentiel et le temps parallèle) et l'efficacité e (l'accélération divisée par le nombre de processeurs). Globalement, l'efficacité moyenne est de 0.9.

p	$(d_1, ..., d_6)$	$T(sec)$	σ	e
1	$(1,1,1,1,1,1)$	19.2713	1	1
2	$(1,1,1,1,1,2)$	9.7235	1.98	0.99
3	$(1,1,1,1,1,3)$	6.6964	2.87	0.95
4	$(1,1,1,1,2,2)$	4.8267	3.99	0.99
6	$(1,1,1,1,2,3)$	3.4404	5.60	0.93
8	$(1,1,1,2,2,2)$	2.4755	7.78	0.97
9	$(1,1,1,1,3,3)$	2.3436	8.2	0.91
12	$(1,1,1,2,2,3)$	1.8949	10.17	0.84
16	$(1,1,2,2,2,2)$	1.3376	14.40	0.90
18	$(1,1,1,2,3,3)$	1.3247	14.54	0.81
24	$(1,1,2,2,2,3)$	0.9019	21.36	0.89
27	$(1,1,1,3,3,3)$	0.8130	23.70	0.98
32	$(1,2,2,2,2,2)$	0.6423	30.00	0.93

TAB. 3.4 – *Performances sur la CRAY T3E*

3.12 Conclusion

Dans ce chapitre, nous nous sommes penchés sur la conception des algorithmes efficaces pour la multiplication d'un vecteur par un produit tensoriel de matrices. Ce problème met en jeu une forte dépendance des données et un espace mémoire global important. Nos solutions séquentielles et parallèles ont été obtenues par un usage judicieux de l'adressage multidimensionnel appliqué à la factorisation canonique de la matrice principale. Quelques expérimentations illustrent l'efficacité de nos algorithmes.

Chapitre 4

Parallélisation du problème du chemin algébrique

Ce chapitre résulte d'une étude menée sur le sujet de la parallélisation du problème du *chemin algébrique*. Les différents résultats obtenus ont été publiés dans [119, 21].

4.1 Résumé

Nous dérivons une architecture linéaire de type SIMD pour la résolution du probléme du *chemin algébrique*. Pour un graphe ayant n sommets, notre réseau est composé de n processeurs ayant chacun une mémoire locale de taille $2n$, et calcule le résultat en $3n^2 - 2n$ étapes. Du fait de la simplicité du contrôle et de la gestion de la mémoire, notre réseau offre une issue convenable pour une implémentation du type VLSI. De plus, le systéme des Entrées/Sorties est assez souple puisque le réseau est linéaire. Dans le cas où on a $p < n$ processeurs, on a une adaptation triviale du réseau par une exécution en plusieurs passes, et de plus, cette version améliore l'efficacité globale. Pour une constante positive $\alpha < n$, le temps d'exécution de l'algorithme sur $\frac{n}{\alpha}$ processeurs est borné par $(\alpha+2)n^2$. Le travail total est donc borné par $(1+\frac{2}{\alpha})n^3$, quantité qui peut être rendue aussi proche que désirée de n^3 en augmentant α. Enfin, notons qu'avec la possibilité d'une version bloc de l'algorithme, on a une issue naturelle vers une implémentation efficace sur des machines parallèles à mémoire distribuée. Les résultats expérimentaux obtenus sur la machine INTEL PARAGON confirment nos analyses d'optimisation sur la taille des blocs d'une part, et nos prédictions sur les performances générales d'autre part. Nous proposons ensuite une autre solution de même nature qui s'exécute en $(\alpha+\frac{1}{\alpha})n^2$ cycles sur $\frac{n}{\alpha}$ processeurs ayant chacun une mémoire de taille n. Cette solution, de travail égal à $(1+\frac{1}{\alpha^2})n^3$, améliore la précédente solution et les tests sur la machine CRAY T3E confirment nos prédictions et l'efficacité de la solution.

4.2 Introduction

Le *problème du chemin algébrique*, en anglais *the algebraic path problem* (APP), unifie un nombre de problèmes classiques en un schéma formel unique. Il se définit comme suit. Etant donné un graphe valué $G = \langle V, E, w\rangle$ avec des sommets $V = \{1, 2, \ldots n\}$, des arcs $E \subseteq V \times V$ et une fonction de poids $w : E \to S$, où S est un semi anneau fermé $\langle S, \oplus, \otimes, *, \mathbf{0}, \mathbf{1}\rangle$ (fermé dans le sens que $*$ est un opérateur unaire de "fermeture", définie par la somme infinie $x* = x \oplus (x \otimes x) \oplus (x \otimes x \otimes x) \oplus \ldots$). Un chemin dans G est une séquence (finie ou non) de sommets $p = v_1 \ldots v_k$ et le poids d'un chemin est défini comme étant le produit $w(p) = w(v_1, v_2) \otimes w(v_2, v_3) \otimes \ldots \otimes w(v_{k-1}, v_k)$. Soit $P(i, j)$ l'ensemble (finie ou non) de tous les chemins de i à j. L' APP est le problème du calcul, pour toutes les paires i, j telles que $0 < i, j \leq n$, de la valeur $d(i, j)$ définie comme suit ($\bigoplus$ est la l'opérateur de "sommation" dans S)

$$d(i, j) = \bigoplus_{p \in P(i,j)} w(p)$$

l'algorithme de Warshall pour la fermeture transitive, l'algorithm de Floyd pour les plus courts chemins et l'algorithme de Gauss-Jordan pour l'inversion matricielle sont toutes des instances de l'agorithm générique pour l'APP (de ce fait appelé l'algorithme WFGJ), la seule différence étant le semi anneau impliqué. Sa complexité séquentielle (et donc le travail) est n^3 opérations de semi anneau.

Une importante activité s'est développée autour de l'implémentation de l'APP (ou d'une instance particulière) sur des réseaux systoliques (voir Table 4.1). La plupart des premières solutions étaient had hoc, tandis que les plus récentes utilisent des méthodes systématiques de conception. Ces implémentations requierent un temps en $\Theta(n)$ sur $\Theta(n^2)$ processeurs. Au début des années 90, une nouvelle localisation réduisant le temps d'exécution de $5n$ à $4n$ fut proposée indépendament par un certain nombre d'auteurs [22, 118, 124, 148]. La technique de Scheiman-Cappello et Benaini-Robert [11, 134] peut être utilisée pour réduire le nombre de processeurs dans ces réseaux rapides à $n^2/3$ [149]. Toutefois, tous ces réeseaux diminuent l'efficacité d'un facteur constant sur un nombre réduit de processeurs.

En pratique, cette limitation est importante. Une implémentation directe de l'une quelconque de ces architectures, par un partitionnement ayant pour but une exécution sur un réseau VLSI dédié, n'améliorerait pas l'efficacité du travail.

Dans ce papier, nous proposons deux architectures SIMD VLSI qui ne souffrent pas de ce problème. Nous dérivons formellement deux architectures ayant n processeurs et un temps d'exécution égal à $\Theta(n^2)$, et qui ameliorent l'efficacité dans un contexte de partitionnement. Plus présicemment, le réseau est composé de n processeurs et calcule la solution en $3n^2 - 2n$ (resp. $2n^2 + n - 2$) étapes. De plus, ils s'adaptent trivialement pour une exécution sur $p < n$ processeurs en effectuant des passes multiples, et ceci avec un gain d'efficacité. Avec α passes (sur $\frac{n}{\alpha}$ processeurs), le temps d'exécution est bornée par $(\alpha+2)n^2$ (resp. $(\alpha+\frac{1}{\alpha})n^2$), et par suite, le travail total est borné par $(1+\frac{2}{\alpha})n^3$ (resp. $(1+\frac{1}{\alpha^2})n^3$). De ce fait, en augmentant le nombre de passes, nous approchons autant que souhaité la performance optimale. Finallement, sur un nombre *fixe* de processeurs (où

Auteurs	Application	Surface	Temps
Guibas et al. [64]	TC	n^2	$6n$
Nash-Hansen [98]	MI	$3n^2/2$	$5n$
Robert-Tchuenté [125]	MI	n^2	$5n$
Kung-Lo [83]	TC	n^2	$7n$
Rote [128]	APP	n^2	$7n$
Robert-Trystram [126]	APP	n^2	$5n$
Kung-Lo-Lewis [82]	TC & SP	n^2	$5n$
Benaini et al. [10]	APP	$n^2/2$	$5n$
Benaini-Robert [11]	APP	$n^2/3$	$5n$
Scheiman-Cappello [134]	APP	$n^2/3$	$5n$
Clauss-Mongenet-Perrin [31]	APP	$n^2/3$	$5n$
Takaoka-Umehara [148]	SP	n^2	$4n$
Rajopadhye [118]	APP	n^2	$4n$
Risset-Robert [124]	APP	n^2	$4n$
Chang-Tsay [22]	APP	n^2	$4n$
Djamegni et al. [149]	APP	$n^2/3$	$4n$
Réseaux linéaires (systoliques and autres)			
Kumar-Tsai [110]	APP	n^2	$7n^2$
Myoupo-Fabret [97]	APP	n^2/α	$(3+\frac{4}{\alpha})n^2$

TAB. 4.1 – *Implémentations systoliques de l'agorithme* WFGJ *et de ses instances (*SP, TC *et* MI *dénotant respectivement les plus courts chemins, la fermeture transitive et l'inversion matricielle). La table n'est pas exhaustive mais suit l'ordre chronologique.*

$\alpha = n/p$ n'est pas *constant* mais proportionnel à n), nos implémentations sont toutes de travail optimal.

La première architecture n'est pas "purement" systolique car elle requiert deux registres de décalage de taille n sur chaque processeur. Toutefois, du fait qu'un registre de décalage est tout simplement un réseau linéaire de registres, on pourrait arguer que notre architecture est un vrai réseau systolique bidimensionel $n \times n$, dans lequel seul les processeurs extrêmes effectuent des calculs, tandis que les "processeurs" internes contiennent juste des registres. La deuxième architecture quant à elle possède un mode d'accès très régulier à la mémoire, qui de ce fait s'implémente aussi bien avec des registre de décalages. Globalement, le réseau est assez modulaire et idéal pour une implémentation de type VLSI.

Toutefois, de telles architectures sont à grains trop fins pour une implémentation sur des machines parallèles standards. En effet, le temps total d'exécution reste égal à $\Theta(n^3/p)$ sur p processeurs, mais la constante impliquée devient $\alpha+\beta$, où α est le temps d'une opération, et β le coût de synchronisation de la machine parallèle (ou les frais d'un appel à une routine de communication, généralement appelé *latence*). Dans le cas général de machines parallèles, $\beta \gg \alpha$, et les performances deviennent inacceptables.

Une méthode bien connue pour venir à bout de ce problème est le partitionnement en blocs de l'espace de calcul. Nous dérivons formellement une version bloc de notre algorithme, et par suite un réseau linéaire similaire dans lequel les opérations sont effectuées sur des blocs au lieu des opérandes élémentaires. Le temps total d'exécution reste $\Theta(n^3/p)$ sur p processeurs, mais la constante est maintenant une fonction de b pour un partitionnement en blocs de taille $b \times b$. Nous formulons et resolvons le problème d'optimisation discrète visant à déterminer la valeur de b qui minimise le temps total d'exécution. Nos predictions sont ensuites vérifiées et appréciées sur les machines INTEL PARAGON et CRAY T3E.

4.3 Préliminaires et Formalismes

Le formalisme de base largement utilisé pour étudier l'APP est celui des *équations récurrentes*. L'algorithme classique de l'APP est basée sur le système d'équations récurrentes suivant, où $D_0 = \{i, j, k \mid 0 < i, j \leq n; 0 \leq k \leq n\}$ est le domaine de F.

$$d(i,j) = \{i, j \mid 0 < i, j \leq n\} : F(i, j, n) \tag{4.1}$$

$$F(i,j,k) = \begin{cases} D_0 \cap \{i,j,k \mid k = 0\} & : a_{i,j} \\ D_0 \cap \{i,j,k \mid i = j = k\} & : F(i,j,k-1)* \\ D_0 \cap \{i,j,k \mid i = k \neq j\} & : F(k,k,k) \otimes F(i,j,k-1) \\ D_0 \cap \{i,j,k \mid j = k \neq i\} & : F(i,j,k-1) \otimes F(k,k,k) \\ D_0 \cap \{i,j,k \mid i \neq k; j \neq k\} & : F(i,j,k-1) \oplus \\ & \qquad (F(i,k,k) \otimes F(k,j,k-1)) \end{cases} \tag{4.2}$$

Mis à part le plan d'initialisation $k = 0$, le domaine de F est un $n \times n \times n$ cube. Pour une valeur donnée de k, les points $[k,k,k]$ de la diagonale $i = j = k$ sont appelés *pivots*, tandis que les lignes de chacun des plans $i = k \neq j$ et $j = k \neq i$ (i.e., les lignes et colonnes contenant les points de la forme $[k,j,k]$ et $[i,k,k]$, respectivement) sont appelées *ligne pivot* et *colonne pivot*. Les points restants sont appelés les points *intérieurs*.

Observons que pour tout plan $k = cste$, nous avons l'rodre de calcul suivant (en supposant un nombre suffisant de processeurs). Premièrement, le pivot $[k,k,k]$ est calculé puisqu'il ne dépend que de la valeur correspondante du plan "précédent". Ensuite, le reste de la colonne pivot, c'est à dire les points $[i,k,k]$ pour $i \neq k$ (respectivement les points de la ligne pivot $[k,j,k]$ pour $j \neq k$) sont calculés, puisqu'ils ne dépendent que du pivot et éventuellement des valeurs correspondantes du plan "précédent". Finalement, les points intérieurs sont calculés puisqu'ils dépendent directement de la ligne pivot et de la colonne pivot. Notons aussi que le pivot du plan $k+1$ dépend du premier point intérieur, de sorte qu'aucun calcul du plan suivant ne peut commencer avant que ce premier point intérieur ne soit mis à jour. En d'autres termes, il y a un chemin critique de longueur trois entre deux pivots successifs, qui est le suivant $[k,k,k] \to [k+1,k,k] \to [k+1,k+1,k] \to [k+1,k+1,k+1]$. Ainsi, le temps d'exécution optimal est $3n$. En effet, il est bien connu [118] que l'ordonnancement parallèle optimal

pour le systeme d'équations récurrentes (4.1-4.2) est le suivant.

$$t_f(i,j,k) = \begin{cases} D_0 \cap \{i,j,k \mid k=0\} & : 0 \\ D_0 \cap \{i,j,k \mid i=j=k\} & : 3k-2 \\ D_0 \cap \{i,j,k \mid i=k\neq j\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid j=k\neq i\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid i\neq k; j\neq k\} & : 3k \end{cases} \tag{4.3}$$

Ce ordonnancement suppose un nombre non borné de processeurs, $\Theta(n^2)$ pour être précis, avec des communications globales: en particulier la diffusion du pivot et de la colonne pivot sont nécessaires. Les précédents résultats résumés dans la table Table 4.1 illustrent le fait que la localité requise dans une implémentation systolique impose une augmentation du temps total d'exécution qui passe de $3n$ à $4n$ ou plus.

Avant de continuer, nous allons présenter une version modifiée du système d'équations (4.1-4.2) qui sera la base de notre solution. Le but est de maintenir les lignes et colonnes pivots en premières positions et obtenir une formulation dans laquelle les opérations effectuées dans chaque plan $k = cte$ sont totalement identiques. A cette fin, on définit l'opérateur $\dot{+}$ par $i \dot{+} j = (i+j) \bmod n$. En appliquant la transformation $(i,j,k \rightarrow (i-k) \bmod n, (j-k) \bmod n, k)$ à tous les points du sous-domaine $\{i,j,k \mid 0 < i,j,k \leq n\}$ de D_0, on obtient le systeme d'équations récurrentes suivant, où $D_1 = \{i,j,k \mid 0 \leq i,j < n; 0 < k \leq n\} \cup \{i,j,k \mid k = 0 < i,j \leq n\}$ est le nouveau domaine de F.

$$d(i,j) = \begin{cases} \{i,j \mid 0 < i,j < n\} & : F(i,j,n) \\ \{i,j \mid 0 < i < n; j = n\} & : F(i,0,n) \\ \{i,j \mid i = n; 0 < j < n\} & : F(0,j,n) \\ \{i,j \mid i = j = n\} & : F(0,0,n) \end{cases} \tag{4.4}$$

$$F(i,j,k) = \begin{cases} D_1 \cap \{i,j,k \mid k=-1\} & : a_{i,j} \\ D_1 \cap \{i,j,k \mid i=j=0\} & : F(i\dot{+}1, j\dot{+}1, k-1)* \\ D_1 \cap \{i,j,k \mid i=0<j\} & : F(i\dot{+}1, j\dot{+}1, k-1) \otimes F(0,0,k) \\ D_1 \cap \{i,j,k \mid i>0=j\} & : F(0,0,k) \otimes F(i\dot{+}1, j\dot{+}1, k-1) \\ D_1 \cap \{i,j,k \mid i,j>0\} & : F(i\dot{+}1, j\dot{+}1, k-1) \oplus \\ & \qquad (F(i,0,k) \otimes F(1,j,k-1)) \end{cases} \tag{4.5}$$

Nous avons juste appliqué la transformation de réindexation bien connue de Kung Lo and Lewis [82]. Cette transformation ramene les lignes et colonnes pivots sur les "bords" ($i = 0$ et $j = 0$) du domaine D_1. Ceci peut être vu comme un décalage toroidal de chaque plan k par rapport au précédent (ceci aussi explique pourquoi l'argument $F(i,j,k-1)$ est systématiquement remplacé par $F(i\dot{+}1, j\dot{+}1, k-1)$).

C'est avec le schéma de récurrence (4.5) que nous allons batir nos algorithmes. Un avantage immédiat est la forte régularité dans les dépendances, car celles-ci sont actuellement identiques sur tous les plans tels que $k = cste$. L'idée de base de notre ordonnacement va être de trouver un ordre relatif des calculs dans chaque plan $k = cste$ tel que, le flot de données en sortie des calculs d'un plan correspond au flot de données en entrée pour la mise à jour du plan suivant. D'un point de vue quantitatif, il est clair que la complexité du schéma global va fortement dépendre de la qualité des enchainements.

Nous allons maintenant passer à la présentation et l'analyse de nos réseaux linéaires. Toute notre démarche sera dans un premier temps basée sur la première solution, et ensuite nous décrirons succintement la deuxième.

sectionRéseau systolique linéaire: description intuitive Dans la solution que nous proposons, chaque plan k est exécuté par le processeur k. De ce fait, chaque processeur calcule une matrice $n \times n$ définie par le système (4.4-4.5). Toutefois, l'ordre dans lequel un processeur calcule ses valeurs est assez spéciale. En effet, les éléments sont calculés "sous-matrice principale" par "sous-matrice principale" comme suit.

- L'élément pivot $F(0,0,k)$ est calculé en premier.

- Apres le calcul de la $(i-1)^{ème}$ sous-matrice principale, les éléments des $i^{ème}$ ligne et colonne de la $i^{ème}$ sous-matrice principale sont calculés de manière alternative : $F(i,0,k)$, $F(0,i,k)$, $F(i,1,k)$, $F(1,i,k)$,...,$F(i,i-1,k)$, $F(i-1,i,k)$. A la fin, le dernier élément diagonal $F(i,i,k)$ est calculé. Ceci acheve le calcul de la $i^{ème}$ sous matrice principale.

Nous insistons sur le fait que ceci nous donne l'ordre relatif dans lequel les calculs sont effectués sur un processeur donné, et que des cycles d'inactivité pourraient intervenir. La figure 4.1 illustre cet ordonnancement pour un graphe de taille $n = 8$.

Notons aussi que ceci est l'ordre dans lequel les éléments du plan $k^{ème}$ plan sont calculés. Ils sont envoyés au processeur suivant dans cet ordre mais avec des décalages qui justifient par ailleurs l'existence des cycles d'inactivité. En fait, le processeur met à jour le plan et effectue un décalage toroidal avant d'envoyer les éléments du plan décalé au processeur suivant, dans l'ordre des sous matrices principales tel que nous l'avons décrit précédemment. La raison de ce décalage vient du fait de la dépendance $(i\dot{+}1, j\dot{+}1, k-1)$ qui signifie que la mise à jour d'un point dépend du point qui serait situé "juste en dessous" si le plan avait préalablement subi un décalage toroidal. Ainsi, le premier point intérieur est la première valeur envoyée tandis que le pivot est la dernière. Après $2n-2$ derniers cycles, les ligne et colonne pivots sont envoyées. Regardons attentivement l'implication de cet ordonnancement sur le processeur 2 (Fig. 4.1.b). Puisque le premier élément qu'il doit calculer est son propre pivot, et que celui-ci dépend de $F(1,1,1)$ qui est calculé à $t = 4$ par le processeur 1, le processeur 2 ne commence ses calculs qu'à $t = 5$. Ensuite, il est inactif durant les deux prochains cycles car d'une part les deux valeurs calculées pendant ces cycles par le processeur 1 ne sont pas envoyées, d'autre part le prochain calcul que doit effectuer ce processor 2 est celui de $F(1,0,2)$, qui dépend de $F(2,1,1)$, valeur produite à $t = 7$. A la suite de ceci, la mise à jour de la sous-matrice principale 2×2 est achevée. Au début des calculs de la troisième sous-matrice principale, on observe encore deux cycles d'inactivité pour des raisons analogues. Ceci va se repéter au début de la mise à jour de chaque ligne (comme indiqué par le 2■ dans la Fig. 4.1.b). Toutefois, il n'y a pas de cycle d'inactivité au commencement de la dernière ligne parce que celle-ci correspond aux ligne et colonne pivots du processeur précédent, lesquelles étaient déja calculées à l'avance et stockées dans les registres prévus à cet effet.

Notre comprehension du processus va actuellement s'éclaicir par l'étude de ce qui se passe au niveau du processeur 3 (Fig. 4.1.c). On voit qu'actuellement, on a deux cycles d'inactivité en plus au commencement de chaque ligne (le processeur 2 impose une latence additionnelle de 2 cycles sur chaque ligne). Toutefois, on voit aussi que dans

1	3	6	11	18	27	38	51
2	4	8	13	20	29	40	53
5	7	9	15	22	31	42	55
10	12	14	16	24	33	44	57
17	19	21	23	25	35	46	59
26	28	30	32	34	36	48	61
37	39	41	43	45	47	49	63
50	52	54	56	58	60	62	64

(a) $k = 1$

	5	9	14	21	30	41	54	67
2	8	10	16	23	32	43	56	69
2	13	15	17	25	34	45	58	71
2	20	22	24	26	36	47	60	73
2	29	31	33	35	37	49	62	75
2	40	42	44	46	48	50	64	77
2	53	55	57	59	61	63	65	79
	66	68	70	72	74	76	78	80

(b) $k = 2$

	11	17	24	33	44	57	70	83
4	16	18	26	35	46	59	72	85
4	23	25	27	37	48	61	74	87
4	32	34	36	38	50	63	76	89
4	43	45	47	49	51	65	78	91
4	56	58	60	62	64	66	80	93
2	69	71	73	75	77	79	81	95
	82	84	86	88	90	92	94	96

(c) $k = 3$

(d) $k = 4$

	71	85	98	111	124	137	150	163
12	84	86	100	113	126	139	152	165
10	97	99	101	115	128	141	154	167
8	110	112	114	116	130	143	156	169
6	123	125	127	129	131	145	158	171
4	136	138	140	142	144	146	160	173
2	149	151	153	155	157	159	161	175
	162	164	166	168	170	172	174	176

(d) $k = 8$

FIG. 4.1 – *Illustration du timing pour $n = 8$. Chaque table présente les instants de temps du calcul des éléments $F(i, j, k)$ par le processeur k (pour $k = 1 \ldots 4$, et $k = 8$ la dernière étape). Le symbole x ▮'s représente x cycle d'inactivité. La table pour $k = 4$ est laissé au lecteur à titre d'exercice.*

l'avant dernière ligne, on a 2 cycles d'inactivité au lieu de 4. Ceci est dû au fait que dans la dernière ligne du processeur précédent, il n'y avait pas de cycle d'inactivité. Comme toujours, l'ordonnancement suit l'ordre des sous-matrices principales et chaque point $F(i,j,k)$ est calculé immediatement après que le point $F(i\dot{+}1, j\dot{+}1, k-1)$ ait été envoyé par le processeur précédent.

Nous laissons au lecteur le soin de completer la table du processeur 4 (Fig. 4.1.d).

Remarque 3 *L'expression de la fonction de temps décrite dans Fig. 4.1 est donnée comme suit.*

$$t(i,j,k) = \begin{cases} \text{if } k=0 & \text{then} & 0 \\ \text{if } k=0 \wedge i \geq j & \text{then} & t_d(i,k) - 2(i-j) \\ \text{if } k=0 \wedge i < j & \text{then} & t_d(j,k) - 2(j-i) + 1 \end{cases} \tag{4.6}$$

où

$$t_d(i,k) = \begin{cases} \text{if } i+k \geq n & \text{then} & n^2 + 2n(i+k-n) + n - i - 1 \\ \text{if } i+k \leq n & \text{then} & (i+k)^2 + k - 1 \end{cases} \tag{4.7}$$

□

Observons que $t_d(i,k)$ est la valeur de $t(i,j,k)$ sur la diagonale (ligne des points (i,j,k) tels que $i=j$).

4.4 Dérivation formelle du réseau

Nous avons précédemment décrit le mécanisme d'allocation des tâches aux processeurs et donné une explication intuitive du fonctionnement du réseau. Toutefois, nous n'avons pas donné d'éléments de preuve formelle de sa validité. Nous attaquons cette question.

Théorème 13 *La fonction $t(i,j,k)$ définie par (4.6-4.7) est un ordonnancement valide pour la variable F dans le système d'équations récurrentes (4.4-4.5).* □

Preuve. Nous devons montrer que chaque fois qu'un point $(i,j,k) \in D_1$ dépend d'un autre point $(i',j',k') \in D_1$, alors $t(i,j,k) > t(i',j',k')$. Du fait que les vecteurs de dépendence (et même le nombre de dépendences) sont différents dans les différents sous domaines de D_1, la preuve doit suivre la structure du système (4.4-4.5). En effet, l'ordonnancement considéré sera valide si et seulement si on montre que le prédicat booléen défini par (4.8) peut être réduit à une tautologie. Ceci s'établit sans une difficulté de fond particulière, mais semble fastidieuse à rédiger dans les détails. On pourrait envisager l'usage d'un prouveur de théorèmes.

$$X(i,j,k) = \begin{cases} D_1 \cap \{i,j,k \mid i=j=0\} : t(i,j,k) > t(i\dot{+}1, j\dot{+}1, k-1) \\ D_1 \cap \{i,j,k \mid i=0<j\} : t(i,j,k) > \max(\\ \qquad t(i\dot{+}1, j\dot{+}1, k-1), t(0,0,k)) \\ D_1 \cap \{i,j,k \mid i>0=j\} : t(i,j,k) > \max(\\ \qquad t(0,0,k), t(i\dot{+}1, j\dot{+}1, k-1)) \\ D_1 \cap \{i,j,k \mid i,j>0\} \quad : t(i,j,k) > \max(t(i\dot{+}1, j\dot{+}1, k-1), \\ \qquad t(i,0,k), t(1,j,k-1)) \end{cases} \tag{4.8}$$

□

Rappellons que la fonction d'allocation au processeur est définie comme suit.

$$\forall (i,j,k) \in D_1, \quad p(i,j,k) = k \tag{4.9}$$

Maintenant, nous devons montrer que les fonctions de temps et d'allocation ne sont pas en conflit, c'est à dire que deux points distincts de D_1 ne sont pas alloués à un même processeur au même instant. En effet, on fait la remarque suivante dont la vérification est directe.

Remarque 4 $t(i,j,k) = t(i',j',k) \Rightarrow i = i' \wedge j = j'$. □

Remarque 5 *Le temps total d'exécution de notre algorithme sur n processeurs est donné par $t(n-1,n-1,n) = 3n^2 - 2n$, et donc une efficacité égale à $\frac{1}{3}$.* □

Dans la situation où l'on dispose de $p < n$ processeurs, on peut profiter du fait que les calculs de chaque plan $k = cste$ sont alloués à un seul processeur et que ces calculs sont complètement identiques d'un plan l'autre, pour réaliser la chaine de récurrence complète en effectuant des passes multiples dans le réseau. Rappelons que cette souplesse constitue un des point fort de notre algorithme, car elle lui confère un modularité naturelle très utile en pratique.

Nous allons maintenant faire une analyse du temps total requis par l'algorithme dans ce cas où il est exécuté sur $p < n$ processeurs avec la stratégie precedemment décrite. Dans la première passe, le k^{eme} processeur commence ses calculs à $t_s(k) = t(0,0,k) = k^2 + k - 1$ et est actif jusqu'à $t_f(k) = t(n-1,n-1,k) = n^2 + 2n(k-1)$. Il est donc actif pendant precisemment $t_a(k) = t_f(k) - t_s(k) + 1 = n^2 + 2n(k-1) - (k^2+k) + 2$ cycles, et puisque $n \geq k$, cette quantité *croit* avec k. Ceci se concoit aisément dans la mesure où chaque processeur effectue exactement n^2 calculs, mais k augmentant, le processeur a de plus en plus de cycles d'inactivité. Donc, si on commence la prochaine passe dès que le premier processeur est libre (c'est à dire à $t = n^2 + 1$), il y aura des conflits.

Dans la situation idéale, on aurait souhaité que le dernier processeur p commence sa prochaine passe exactement à $t_f(p) + 1$, de sorte qu'il n'y ait pas de temps perdu pour ce processeur. Puisqu'il y a exactement $t_s(p)$ cycles depuis le début d'une passe (sur le premier processeur) jusqu'au moment où le dernier processeur entame ses calculs correspondants à cette passe, notre but peut être atteint si le *premier* processeur commence sa seconde passe à $t_f(p) + 1 - t_s(p)$, c'est à dire $t_a(p) + 1$.

Avec un tel ordonnancement, on peut facilement établir le théorème suivant.

Théorème 14 *Le temps total d'exécution sur p processeurs est donné par*

$$T_p = \frac{n^3}{p} + 2n^2\frac{p-1}{p} - (n-p)(p+1) + \frac{2n}{p} - 2 \tag{4.10}$$

□

Corollaire 2 *Pour toute constante (entier positif) α, le temps total d'exécution sur $p = \frac{n}{\alpha}$ processeurs est donné ci-dessous, en supposant que $n \approx n + \alpha \approx n - \alpha$, et en ignorant les termes constants.*

$$T_p \approx n^2 \left(\alpha + 2 - \frac{\alpha - 1}{\alpha^2}\right) \leq n^2(\alpha + 2) \tag{4.11}$$

□

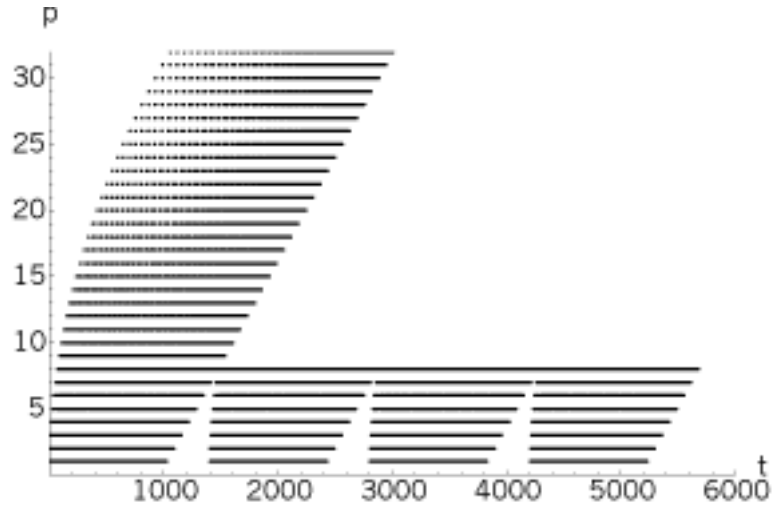


FIG. 4.2 – *Illustration du comportement spatio-temporel de deux réseaux (pour $n = 32$), l'un avec 32 processeurs qui exécute une seule passe et termine à $t = 3008$, et l'autre avec 8 processeurs qui effectue 4 passes et termine à $T = 5678$. Un processeur est actif durant les instants représntés par une ligne solide, et inactif durant ceux représentés par des lignes en pointillées. Bien observer la raison pour laquelle le premier processeur n'a pas de cycles d'inactivité.*

Le travail total de l'algorithme avec des passes multiples sur $\frac{n}{\alpha}$ processeurs est $n^3(1 + \frac{2}{\alpha} - \frac{\alpha-1}{\alpha^3}) \approx (1 + \frac{2}{\alpha})n^3$. Donc, nous pouvons améliorer l'efficacité tout simplement en augmentant α, ce qui correspond à sacrifier le temps d'exécution par un facteur constant, avec une diminution correspondante du nombre de processeurs. Il est important de noter que cette récupération relative de cycles d'inactivité est dû au fait que le premier processeurs n'est jamais inactif dans aucune passe (voir Fig. 4.2), et qu'il émule l'activité d'un processeur qui en aurait eu des cycles d'inactivité dans une version à nombre réduit de passes. Ce mécanisme est illustré par la figure Fig. 4.2.

L'analyse qui vient d'être faite suppose que α est constant, c'est à dire qu'on utilise toujours $\Theta(n)$ processeurs, en maintenant proportionnelles la taille de l'architecture et celle du problème. D'un autre côté, le théorème 4.10 montre que avec un nombre fixe

de processeurs, nous avons une implémentation optimale avec un temps d'exécution dominé par le terme $\frac{n^3}{p}$.

Nous allons maintenant nous pencher sur la version bloc de l'algorithme, ceci afin d'établir un certain équilibre entre les opérations et les communications (dont on sait qu'elles ont un coût significatif sur les machines parallèles standards).

4.5 Version Bloc de l'algorithme

4.5.1 Formulation par blocs

Nous allons essayer de montrer que la spécification de l'APP reste valable si on considère les blocs de points au lieu des entités élémentaires. Ceci semble plus naturel dans les instances de la fermeture transitive et des plus courts chemins. Aussi, nous allons utiliser l'instance de la fermeture transtive pour illustrer nos propos.
Considérons un graph $G = (S, \Gamma)$ d'ordre N. Pour un diviseur donné p de n, considérons une partition de S en $q = \frac{N}{p}$ sous-ensembles S_i tels que $\mid S_i \mid = p$, $1 \leq i \leq q$. Pour $k \in \{1, ..., q\}$, notre but est de déterminer pour chaque pair $x, y \in S$, si oui ou non il existe un chemin entre x et y qui n'emprunte que les sommets dans S_k. Supposons que le problème de la fermeture transtive est déjà résolu dans S_k (sous-ensemble pivot). Si $\ell^{(k)}$ est un prédicat définie par $\ell^{(k)}(x, y) = 1$ s'il y a une chemin entre x entre y empruntant uniquement les sommets de S_k et $\ell^{(k)}(x, y) = 0$ sinon, alors on a les relations suivantes.

$$\ell^{(k)}(x, y) = \Gamma(x, y) \vee [\bigvee_{z \in S_k} (\Gamma(x, z) \wedge \ell^{(k)}(z, y))] \text{ si } x \notin S_k \text{ et } y \in S_k \tag{4.12}$$

$$\ell^{(k)}(x, y) = \Gamma(x, y) \vee [\bigvee_{z \in S_k} (\ell^{(k)}(x, z) \wedge \ell^{(k)}(z, y))] \text{ si } x \notin S_k \text{ et } y \notin S_k \tag{4.13}$$

Considérons les matrices carrées booléennes $M_{ij}^{(k)}$, $1 \leq i, j \leq q$, d'ordre p, définies par

$$M_{ij}^{(k)}(s, t) = \begin{cases} 1 & \text{si } \ell^{(k)}(x_s, y_t) = 1 \text{ pour } x_s \in S_i \text{ et } y_t \in S_j \\ 0 & \text{sinon} \end{cases} \tag{4.14}$$

$$M_{ij}(s, t) = \begin{cases} 1 & \text{si } \Gamma(x_s, y_t) = 1 \text{ pour } x_s \in S_i \text{ et } y_t \in S_j \\ 0 & \text{sinon} \end{cases} \tag{4.15}$$

Partant de $S = \{x_1, ..., x_N\}$, on considère la partition $S_k = \{x_{(k-1)p+1}, ..., x_{kp}\}$, $1 \leq k \leq q$. En appliquant la renumérotation $\{1, ..., p\}$ des sommets de chaque sous-ensembles S_k, il vient que les matrices M_{ij} sont les blocs correspondants de la matrice d'adjacence globale du graphe entier. Ainsi, si on traduit en termes d'opérations matricielles le calcul de la fermeture transitive de S_k et les opérations (4.12) et (4.13), on obtient une formulation compacte qui équivaut à la formulation considérant des entités élémentaires. Ceci est surtout valable dans les cas où c'est la ligne pivot en cours qui doit être prise en compte au lieu de sa valeur précédente. Cette hypothèse semble nécessaire, car de manière intuitive, les traitements sont effectuées par inticipation, ce qui justifie le fait qu'il faut d'abord traiter les blocs situés sur la ligne ou colonne pivots avant leur usage pour le reste des calculs. A première vue, seuls les cas de la fermeture transitive et des plus courts chemins (instances les plus courament concernées) respectent ce prérequis.

4.5.2 Recherche de la taille optimale des blocks

On suppose maintenant que l'algorithme est appliqué à des blocs de taille $b \times b$. Le temps global d'exécution sur une machine parallèle à mémoire distribuée peût être directement modelisé en considérant la formule (4.10) multipliée par le facteur $\tau b^3 + \beta + \alpha b^2$ (representant le temps d'une opération sur un bloc suivi de son transfert). Toutefois, la recherche d'une taille optimale de bloc à partir de l'expression obtenue conduit à la résolution d'une équation n'admettant pas de solution directe. Aussi, dans le but de fournir une expression analytique donnant la meilleure taille des blocs, nous allons procéder à quelques suppositions simplificatrices. En effet, si on considère la cas de p processeurs et une matrice de taille $N \times N$ partitionnée en $(\frac{N}{b})^2$ blocks de taille $b \times b$, on a les considérations suivantes.

- Il n'y a pas de latence entre deux passes consécutives sur le premier processeur.
- Le coût de chaque communication, qui est de la forme $\tau b^2 + \beta$, peut être approximé par β du fait des petites valeurs de b et d'un éventuel recouvrement entre la transmission d'un bloc et le traitement du bloc suivant.
- Chaque processeur effectue trois actions sur chaque bloc qui sont : la *réception*, le *calcul* et l'*envoi*.
- Chaque opération sur un bloc implique b^3 opérations scalaires.

Dans ces conditions, et du fait qu'on réalise $\frac{(\frac{N}{b})}{p}$ passes dans le réseau, on obtient que le temps total d'éxécution peut être estimé par

$$T(p,b) = \frac{N}{bp}[(\frac{N}{b})^2(\tau b^3 + 2\beta)] + 2(\frac{N}{p})(\tau b^3 + 2\beta), \tag{4.16}$$

qui peut aussi s'exprimer par

$$T(p,b) = (\frac{N^3}{p}\tau)[(1 + 2\frac{\beta}{\tau b^3}))] + 2(\frac{N}{p})(\tau b^3 + 2\beta). \tag{4.17}$$

Toujours en supposant des valeurs de b suffisament petites, ceci nous donne une efficacité de l'ordre de

$$\boxed{e = \frac{1}{1+2\frac{\beta}{\tau b^3}}} \tag{4.18}$$

La détermination de la valeur de b qui fournit un temps d'exécution minimal s'obtient en résolvant l'équation

$$\frac{\partial T}{\partial b} = -6N(\frac{\beta}{b^4}N^2 - b^2\tau) = 0 \tag{4.19}$$

qui donne

$$\boxed{b_{\text{opt}} = \sqrt[6]{\frac{\beta}{\tau}N^2}} \tag{4.20}$$

Nous insistons sur le fait que ceci est une analyse approximative qui vise à situer la taille qui donnerait de bonnes performances en pratique. A la base, le problème est le suivant. Plus les blocs sont grands, plus la latence entre le calcul d'un bloc et son envoi est importante, ce qui pénalise la performance globale. De même, si les blocs sont de très petite taille, le nombre d'appels aux routines de communication sera important et par conséquent, le surcoût de temps imputé aux communications sera prohibitif. Il faut donc donc trouver une taille qui assure au mieux le compromis entre ces deux aspects. Il apparait donc que trois paramètres sont importants dans cette analyse, il s'agit de la *vitesse des processeurs*, de la *latence des communications*, et de la *taille* du problème. A ce niveau, le lecteur va sans doute être perplexe de constater que le nombre de processeurs semble ne pas intervenir. En effet, l'impact du nombre de processeurs est négligeable devant celui des autres paramètres, de sorte que la performance du programme est moins sensible à variation de la taille du réseau.

4.6 Résultats expérimentaux

4.6.1 Environnement matériel

La machine INTEL PARAGON est une machine parallèle à mémoire distribuée, composée de 64 noeuds (8 pour le système et 56 pour les programmes utilisateurs) organisés en une grille bidimensionnelle. Chaque noeuds contient deux processeurs $i860^{tm}$ dont l'un est dédié aux calculs et est cadencé à 75 MHZ avec une mémoire locale de 16 MB, tandis que l'autre est spécialisé pour les communications (Fig. 4.3).

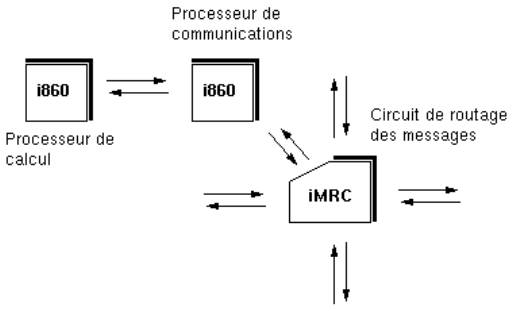


FIG. 4.3 – *Principe architectural de la* PARAGON.

4.6.2 Implémentation

Le programme a été écrit en FORTRAN 77 avec la librairie de communication NX. C'est un programme de type SPMD dans lequel chaque processeur exécute le code nécessaire à la mise à jour d'un plan. Ici, la synchronisation se fait automatiquement au niveau des communications. Etant donné que chaque bloc envoyé est immédiatement

consommé par le processeur qui le recoit, il en ressort que les seuls cas de bufferisation pouvant apparaitre proviendront uniquement du léger déséquilibre de charge entre les temps de calculs sur les blocs. Par ailleurs, pour ce qui est des passes multiples, nous avons implémenté un mécanisme permettant d'anticiper la réception des blocs d'une passe ultérieure sur le premier processeur, afin d'éviter leur stockage temporaire éventuellement sur disque, ce qui aurait sans doute entrainé une sérieuse perte d'efficacité. Toutefois, ce dernier aspect est inévitable si le graphe est de très grande taille, mais, étant donné que seuls les processeurs extrêmes auraient à faire des accès disque, un certain équilibre pourait être atteint dans la phase de plein régime.

4.6.3 Performances mesurées

Nous avons considéré le cas d'un graphe de taille $N = 512$. Commencons par évaluer la meilleure taille de bloc conformément à la formule (4.20). La mesure des paramètres τ et β nous a donné les valeurs suivantes sur la machine INTEL PARAGON : $\tau = 0.05 \times 10^{-6}$ and $\beta = 30 \times 10^{-6}$. L'évaluation par la formule (4.20) nous donne alors la valeur $b = \sqrt[6]{0.05 \times 30 \times 512} = 8.2 \approx 8$. Ceci nous a amené à considérer les valeurs de l'ensemble $\{4, 8, 16\}$ comme tailles de blocs pour nos expérimentations. Les résultats obtenus sont résumés dans la table de la figure 4.4 et la courbe correspondante en figure 4.5.

Nombre de processeurs	1	2	4	8	16	32
$b = 4$	68	81.5	41	21	11.75	7
$b = 8$	68	39	18	9	5	3
$b = 16$	68	38.5	21	12	8.75	9

FIG. 4.4 – *Tableau des performances sur la* PARAGON *avec les blocs de taille 4, 8, et 16.*

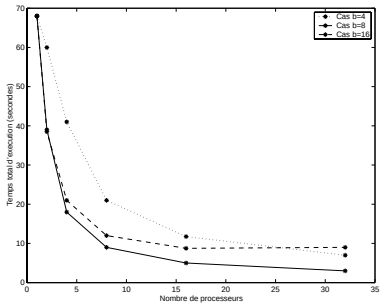


FIG. 4.5 – *Courbe de performances sur la* PARAGON *avec les blocs de taille 4,8, et 16.*

La table de la figure 4.4 montre que la valeur $b = 8$ donne les meilleurs performances

comme prévu. En considérant cette valeur de b, on obtient la table de la figure 4.6 et la courbe des accélérations optimales en figure 4.7.

Nombre de processeurs	1	2	4	8	16	32
Temps d'exécution (s)	68	39	18	9	5	3
Accélération	1.00	1.74	3.78	7.56	13.60	22.67

FIG. 4.6 – *Table des performances optimales sur la* PARAGON

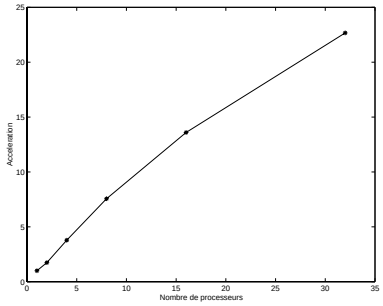


FIG. 4.7 – *Courbe des accélérations optimales sur la* PARAGON.

4.6.4 Commentaires

De nos expérimentations, il apparait (comme prévu d'ailleurs) que la taille des blocs a une importance capitale dans les performances globales de l'algorithme. Celle-ci ne doit être ni trop petite (trop d'appels aux routines de communication), ni top grande (déséquilibre important entre les calculs et les tranferts entrainant des inerties momentanées globalement coûteuses). On peut observer que les écarts de performances sont de plus en plus importantes lorsque le nombre de processeurs croît. Par ailleurs, la contrainte d'avoir un nombre de processeurs p qui est un diviseur de la taille $\frac{N}{b}$ de la matrice bloc peut être contournée, soit en exécutant l'algorithme sur un graphe augmenté, avec des valeurs correspondant à l'élément neutre pour l'opération scalaire impliquée, soit en prenant en compte le fait que la dernière passe ne devrait pas être exécuté entièrement (même si on peut le faire sans altérer le resultat final, car au delà de l'étape de la récurrence finale, les opérations deviennent sans effet sur les valeurs). Cette deuxième alternative s'implémente sans difficulté particulière sur les machines parallèles standards dans la mesure où le réseau est virtuel. Dans tous les cas, notre estimation de la meilleur taille de bloc semble concorder avec les mesures effectives et de plus, les performances globales sont appréciables.

4.7 Une solution linéaire amméliorée

Ici nous décrivons une autre solution étudiée dans le même esprit.

4.7.1 Ordonnancement

De manière intuitive, la solution consiste à parcourir chaque plan k colonne après colonne, en considérant pour chaque colonne en cours l'ordre croissant des index.

Les fonctions de temps et d'allocation correspondantes sont données par les expressions (4.21) et (4.22).

$$t(i,j,k) = \begin{cases} \text{si } k = 0 & \text{then} \quad 0 \\ \text{si } k \geq 1 & \text{then} \quad (n+2)(k-1) + jn + i + 1 \end{cases} \tag{4.21}$$

et

$$\forall (i,j,k) \in D_1, \quad p(i,j,k) = k \tag{4.22}$$

Pour établir la validité de l'ordonnancement, outre une preuve directe, on peut aussi passer par l'usage d'un prouveur de théorèmes (tel PVS). Pour cela, il serait utile de considérer la forme fournie par l'équation (4.23).

$$\begin{array}{l} D_1 \cap \{i,j,k \mid i = j = 0\} : \\ \qquad t(i,j,k) > t(i+1, j+1, k-1) \\ D_1 \cap \{i,j,k \mid i = 0; 0 < j < N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(i+1, j+1, k-1), t(0,0,k)) \\ D_1 \cap \{i,j,k \mid i = 0; j = N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(i+1, 0, k-1), t(0,0,k)) \\ D_1 \cap \{i,j,k \mid j = 0; 0 < i < N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(0,0,k), t(i+1, j+1, k-1)) \\ D_1 \cap \{i,j,k \mid i = N-1; j = 0; k > 0\} : \\ \qquad t(i,j,k) > \max(t(0,0,k), t(0, j+1, k-1)) \\ D_1 \cap \{i,j,k \mid 0 < i,j < N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(i+1, j+1, k-1), t(i,0,k), t(1,j,k-1)) \\ D_1 \cap \{i,j,k \mid 0 < i < N-1; j = N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(i+1, 0, k-1), t(i,0,k), t(1,j,k-1)) \\ D_1 \cap \{i,j,k \mid 0 < j < N-1; i = N-1; k > 0\} : \\ \qquad t(i,j,k) > \max(t(0, j+1, k-1), t(i,0,k), t(1,j,k-1)) \end{array} \tag{4.23}$$

A l'aide de PVS, nous avons montrer que les assertions de (4.23) se réduisent à des tautologies.

L'adéquation entre la fonction de temps et la fonction d'allocation peut se faire de manière directe sans aucune difficulté particulière.

En ce qui est de la mémoire, chaque processeur aura besoin de d'une fifo de taille n pour le stokage de la colonne pivot du plan k, et de deux registres pour le stockage du point intérieur du plan précédent et de l'élément de la colonne courante situé sur la ligne pivot. Etant donné que les éléments de la ligne pivot sont visités en continu et de manière cyclique, on peut imaginer la représentation fournie par la figure (Fig. 4.8).

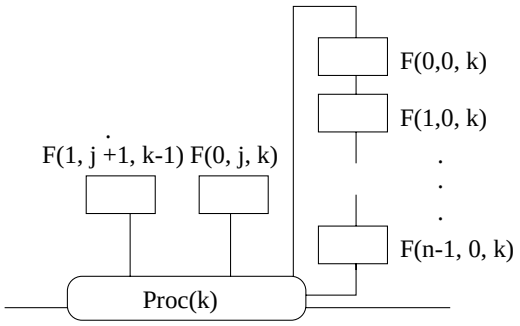


FIG. 4.8 – *Illustration de la gestion locale des données.*

A ce niveau, il est utile de montrer que l'architecture fonctionne correctement, c'est à dire que le flot de données est cohérent et s'accorde bien avec les fonctions de temps et d'allocation. A cet effet, nous avons besoin d'une modélisation de la circulation des données dans la mémoire (BUF) et dans les régistres (L0 et L1). Nous proposons la spécification suivante écrite avec le formalisme de PVS.

```
dependKLL [N : {n:nat | n>=4}]
          : THEORY
  BEGIN

domain: TYPE = {i,j,k:nat |  i<=N-1 AND j<=N-1 and k<=N}
cell: TYPE = [{k:nat | k<=N},{i:nat | i<=N+2}]
L0 : nat = N
L1 : nat = N+1
BUF : nat = N+2

time(i,j,k:nat) : int= (N+2)*(k-1) + j*N + i + 1

mem(p:domain,t:int) : cell = LET (i,j,k) = (PROJ_1(p),PROJ_2(p),PROJ_3(p)) IN
     COND
       j = 0 AND i = 0 AND time(i,j,k)<=t AND t <= time(N-1,N-1,k)+N -> (k,i),
       j = 0 AND i = 0 AND t = time(N-1,N-1,k)+N+1 -> (k+1,BUF),
       j = 0 AND i = 1 AND time(i,j,k)<=t AND t <= time(N-1,N-1,k)+2 -> (k,i),
       j = 0 AND i = 1 AND t = time(N-1,N-1,k)+3 -> (k+1,BUF),
       j = 0 AND i = 1 AND time(N-1,N-1,k)+4<=t AND
                    t <= time(N-1,N-1,k)+2*N+2 -> (k+1,L1),
       j = 0 AND i > 1 AND time(i,j,k)<=t AND
                   t <= (time(N-1,N-1,k) +i ) -> (k,i),
       j = 0 AND i > 1 AND t = time(N-1,N-1,k)+i+1  -> (k+1,BUF),
       j > 0 AND i = 0 AND time(i,j,k)<=t AND t <= (time(i,j,k)+N -1)->(k,L0),
       j > 0 AND i = 0 AND  t = (time(i,j,k)+N )->(k+1,BUF),
       j > 0 AND i = 1 AND time(i,j,k)+1=t ->(k+1,BUF),
       j > 0 AND i = 1 AND  time(i,j,k)+2<=t AND
                   t <= time(i,j,k)+N AND k<=N-1 ->(k+1,L1),
       j > 0 AND i > 1 AND time(i,j,k)+1=t ->(k+1,BUF)
     ENDCOND

  END dependKLL
```

A l'aide ce cette expression, nous avons spécifier et prouver les différentes proprétés

qui garantissent le bon fonctionnement de l'architecture (principe de la localité, accès exclusifs, durée de vie suffisante).

4.7.2 Complexité

Théorème 15 *Le temps d'exécution de l'algorithme sur p processeurs est donné par*

$$T_p = \frac{n^3}{p} + (n+2)(p-1) \tag{4.24}$$

□

Preuve. Notons premièrement qu'il n'y a aucun cycle d'inactivité sur un processeur depuis son premier calcul jusqu'au dernier. Ensuite, chaque passe requiert $t(n-1, n-1, 1)$ cycles relatifs sur le premier processeur, et la dernière passe se termine sur le dernier processeur après $t(n-1, n-1, p)$ cycles. Ainsi, on a

$$\begin{aligned} T_p &= (\tfrac{n}{p} - 1)t(n-1, n-1, 1) + t(n-1, n-1, p) \\ &= (\tfrac{n}{p} - 1)(n^2) + t(n-1, n-1, p) \\ &= (\tfrac{n}{p} - 1)(n^2) + (n+2)(p-1) + (n-1)n + (n-1) + 1 \\ &= \tfrac{n^3}{p} + (n+2)(p-1) \end{aligned}$$

ce qui achève la preuve. □

Corollaire 3 *Pour tout entier positif α, le temps total d'exécution sur $p = \frac{n}{\alpha}$ processeurs est donné par*

$$T_p = n^2(\alpha + \frac{1}{\alpha}) - [n(1 - \frac{2}{\alpha}) + 2] \leq n^2(\alpha + \frac{1}{\alpha}) \tag{4.25}$$

□

Le travail de l'algorithme à passes multiples sur $\frac{n}{\alpha}$ processors est de $(1 + \frac{1}{\alpha^2})n^3$. Par conséquent, on peut améliorer l'efficacité globale en augmentant suffisemment α, c'est à dire en sacrifiant le temps d'exécution par un facteur constant avec une diminution correspondante du nombre de processeurs.

De (4.24), il est clair qu'avec un nombre fixe de processeurs, on a une implémentation de travail optimal, avec un temps d'exécution dominé par le terme $\frac{n^3}{p}$.

Nous donnons maintenant quelques résultats généraux de complexité au sujet des ordonnancements similaires à ceux que nous avons considérés, avec en plus quelques restrictions sur le mécanisme des entrées/sorties, et nous montrons que notre solution est optimale dans ce contexte.

Lemme 4 *Le calcul du pivot $F(0, 0, n)$ du dernier plan $k = n$ requiert tout les points du premier plan $k = 1$ appartenant à l'ensemble $\{F(0, 0, 1)\} \cup \{F(i, j, 1) : 1 < i, j \leq n\}$.*

Preuve. Il suffit de développer les chaines de dépendances issues du dernier pivot jusqu'au premier plan $k = 1$. □

Ce résulat admet aussi l'interprétation intuitive suivante. L'APP implique un traitement dans lequel le résultat sur chaque point dépend de celui de tous les autres points dans un certain ordre. De plus, le traitement effectué sur chaque plan k commence par le pivot duquel dépend toutes les autres opérations (voir [21] pour des détails formels).

En prenant en compte le résultat du lemme (4) et le fait que tous les points d'un plan k sont calculés par un seul processeur, on obtient les résultats suivants.

Théorème 16 *Le temps d'exécution de tout ordonnancement du* SRE *(4.1-4.2) tel que* $a(i,j,k) = \phi(k)$ *est minoré par* $2n^2$. □

Théorème 17 *La taille mémoire totale de tout réseau calculant la solution de (4.1-4.2) et tel que* $a(i,j,k) = \phi(k)$ *est minoré par* n^2. □

Ces deux résultats peuvent être résumés de la manière suivante. Tout ordonnancement de (4.1-4.2) tel que $a(i,j,k) = \phi(k)$ requiert requiert un espace mémoire total égal à n^2, et s'exécute en au moins $2n^2 + \theta(n)$ cycles. Notre solution courante atteint ses performances et nous pouvons de ce fait affirmer qu'elle est optimale dans le contexte que nous avons précisé.

4.7.3 Version bloc et Optimisation

En considérant la version bloc de l'algorithme, on a un temps d'exécution donné par

$$T(b) = \left(\frac{(\frac{n}{b})^3}{p} + (\frac{n}{b} + 2)(p-1) \right) (\tau b^3 + \alpha b^2 + \beta), \tag{4.26}$$

qu'on peut approximer par

$$\tau \frac{n^3}{p} + \beta \frac{n^3}{pb^3} + n(p-1)(\alpha b^2 + \beta). \tag{4.27}$$

Avec cette approximation, l'équation $\frac{\partial T}{\partial b} = 0$ est èquivalente à

$$2b\alpha(p-1)n - 3\beta \frac{n^3}{pb^4} = 0, \tag{4.28}$$

qui admet la solution

$$\boxed{b_{opt} = \sqrt[5]{\frac{3n^2\beta}{2\alpha p(p-1)}}} \tag{4.29}$$

4.7.4 Performances

Nous avons considéré le cas d'un graphe de taille $n = 1024$ sur la machinee CRAY T3E pour laquelle on a mesuré (sur des courts messages, ce qui est le cas ici) $\beta = 10 \times 10^{-6}$, $\alpha = 0.01 \times 10^{-6}$ and $\tau = 0.02 \times 10^{-6}$. Le temps d'exécution de l'algorithme sur un

processeur est $T = 20s$, et le meilleur temps d'exécution parallèle sur $p = 16$ a été obtenu avec $b = 23$ (la valeur donnée par la formule est $b_{opt} = 23.08$), et est égale à $T_p = 1.42s$, soit un speedup égal à $\sigma = 14.08$. La figure Fig. 4.9 présente les prédictions et les résultats expérimentaux. La valeur de b qui conduit à la meilleure performance est attestée par les expérimentations, ainsi que l'allure globale du timing. Notons que pour les valeurs de b pour lesquelles p n'est pas un facteur de n/b, la dernière passe n'a pas besoin d'être effectuée sur le réseau tout entier, mais ceci ne pose aucun problème car dans ce cas de machine parallèle à mémoire distribuée, car la topologie du réseau est "virtuelle" et il n'y a donc aucun problème particulier à obtenir les résultats finaux sur le processeur approprié.

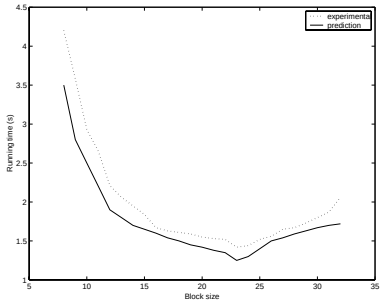


FIG. 4.9 – *Prédictions et résultats expérimentaux sur p=16 processeurs et N=1024*

4.7.5 Commentaires

Il est important de noter que dans les deux solutions que nous venons de présenter, le partitionnement est automatique, c'est à dire qu'on doit juste passer en paramètres le nombre de processeurs et l'exécution s'en suivra sans aucune autre intervention extérieure. Par ailleurs, nous pensons que nos algorithmes sont bien adaptés au calcul out-of-core car la taille des données qu'on a besoin de stocker à des instants donnés est assez faible par rapport à la totalité de l'ensemble des données à manipuler.

4.8 Conclusion

Nous avons présenté deux architectures SIMD pour le problème du chemin algébrique. Ceux-ci s'exécutent en temps $3n^2$ (resp. $2n^2$) sur n processeurs ayant chacun une mémoire FIFO de taille $2n$ (resp. n). De plus, les réseaux dérivés s'adaptent trivialement au cas o'u $p < n$ et améliorent l'efficacité globale. Des versions en blocs de ces algorithmes nous ont permis d'avoir des implementations efficaces sur machine a mémoire distribuée. Les résultats expérimentaux obtenus sur les machines INTEL PARAGON et CRAY T3E confirment nos prédictions, attestent l'efficacité de nos solutions, et prouvent d'une manière générale que notre démarche est assez prometteuse.

Chapitre 5

Un échéancier systolique

Ce chapitre présente une étude du problème de file de priorité systolique et ses diverses applications. Ces résultats sont publiés dans [112]. Toutefois, ce travail a évolué dans contexte que nous décrirons brièvement à la fin du chapitre.

5.1 Introduction

Le dispositif décrit est destiné à réaliser un échéancier synchrone en temps constant. On suppose qu'une suite de valeurs positives entières x_t sont fournies à des instants discrets t. L'échancier a une capacité maximale de N valeurs. Au temps t, la valeur x_t est insérée dans l'échéancier (opération d'insertion). L'échancier est capable de fournir la plus grande des valeurs qu'il contient (opération d'extraction). L'échéancier possède la propriété suivante : quel que soit le nombre N de valeurs qu'il est capable de mémoriser, le temps nécessaire à une insertion ou à une extraction est constant, indépendant de N. En outre, il n'utilise que $log_2(N)$ processeurs, et une mémoire de taille N. Enfin, l'adressage de la mémoire de chaque processeur est particulièrement simple, car il n'utilise qu'un mécanisme de décalage.

Ce dispositif a de nombreuses applications potentielles. La plus immédiate concerne le transfert de données par ATM (Asynchronous Transfer Mode). Dans ce mode de transfert, les données sont transférées par cellules de taille fixe, et le respect de la qualité de service ou le contrôle du respect par les usagers des débits des données nécessite le tri des cellules arrivant sur des multiplexeurs, afin d'émettre à tout moment la cellule la plus prioritaire. Une autre application concerne le tri d'objets graphiques suivant leur profondeur dans un dispositif de visualisation d'objets en trois dimensions.

Dans ce document, nous présentons successivement les principes du dispositif (paragraphe 5.2), puis l'architecture de l'échéancier (paragraphe 5.3), et ensuite son application à l'ATM (paragraphe 5.4). Le paragraphe 5.5 décrit l'historique de ce travail et conclut le chapitre.

5.2 Principe du dispositif

Dans ce qui suit, on suppose que l'on cherche à extraire le minimum de N nombres, mais le dispositif peut bien entendu être appliqué à la recherche du maximum. L'algorithme repose sur le principe du tri par tas (*heap sort*, voir [77]), illustré ci-dessous. Un tas est un arbre binaire dont chaque sommet contient une valeur, et cette valeur est inférieure à celles qui sont contenues dans ses deux fils. Cette propriété s'appelle la propriété du tas. La figure 5.1 présente un tel tas. Les sommets vides sont supposés contenir la valeur $+\infty$ (voir aussi [150]).

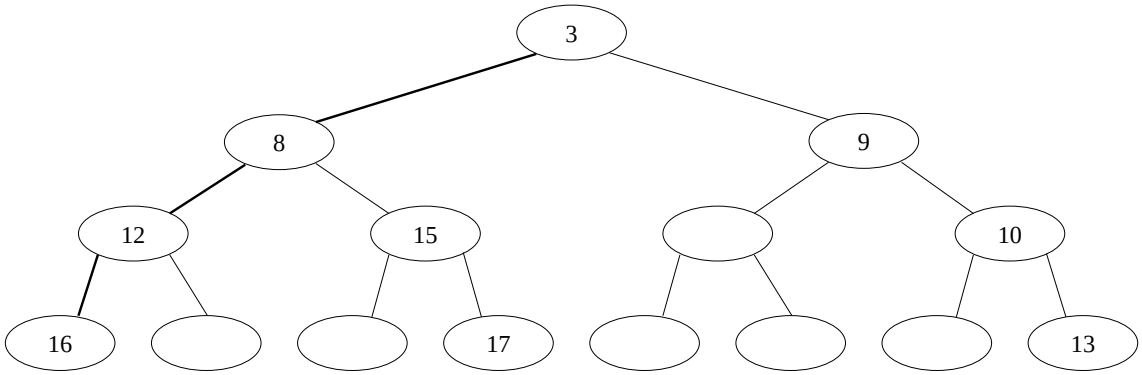


FIG. 5.1 – *Un tas d'entiers positifs. Le chemin critique est indiqué en trait épais.*

Le *chemin critique* d'un tas est le chemin partant de la racine, et qui relie tout sommet avec le plus petit de ses fils. Dans l'exemple de la figure 5.1, le chemin critique est formé des sommets 3, 8, 12 et 16. On peut faire la remarque suivante. Si l'on extrait l'élément contenu dans la racine, et qu'on décale vers la racine les éléments du chemin critique, le nouvel arbre binaire ainsi obtenu possède encore la propriété du tas. La figure 5.2 illustre cette remarque.

Pour transformer cette méthode en échéancier, il faut pouvoir insérer des données, de telle façon qu'à tout moment, la valeur minimale soit disponible. L'insertion doit aussi être faite de telle façon qu'un tas puisse mémoriser jusqu'à $N = 2^n$ valeurs. Pour ce faire, on associe à chaque sommet un compteur qui donne le nombre de sommets non encore occupés dans son sous-arbre. Par exemple, dans le tas de la figure 5.2, la racine a 7 places vides. On peut donc insérer une valeur dans un des sous-arbres qui contient une place vide (si le tas est plein, on peut choisir différentes politiques dont la plus simple consiste à rejeter la valeur qui arrive). Il suffira, lors d'une insertion, de décrémenter les compteurs, et lors d'une extraction, d'incrémenter les compteurs des sommets du chemin critique. L'élément inséré doit l'être de façon que la propriété du tas soit vérifiée. Pour cela, on range le nouvel élément dans le premier sommet rencontré qui contient une valeur plus grande.

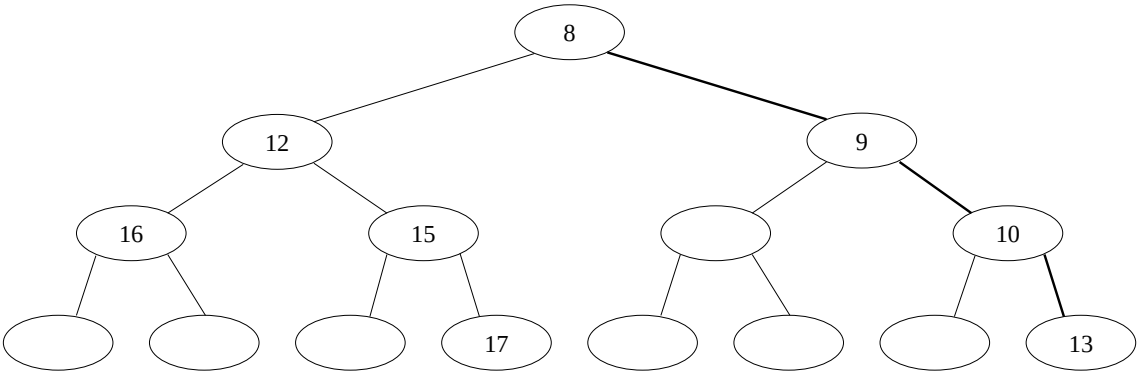


FIG. 5.2 – *Le tas de la figure 1 après extraction d'un élément. Le chemin critique est modifié, mais la propriété du tas est toujours respectée. Cette remarque fonde l'algorithme de tri : ranger les données à trier en tas, puis extraire les valeurs.*

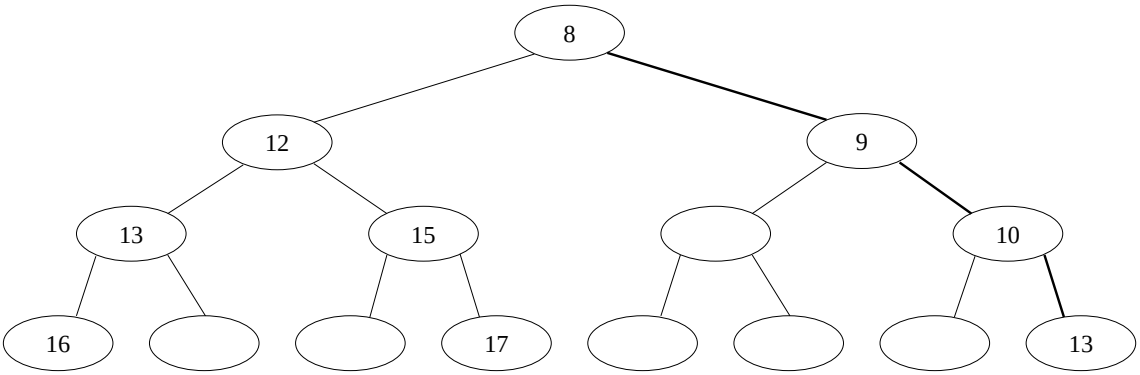


FIG. 5.3 – *Le tas de la figure 5.2 après insertion de l'élément 13. Cet élément a été rangé entre l'élément 12 et l'élément 16, en cherchant à remplir l'arbre en commençant par les éléments les plus à gauche. On peut remarquer que de la sorte, on conserve la propriété du tas*

5.3 Architecture de l'échéancier

L'architecture proposée est de type systolique, et repose sur les principes suivants :

1. On peut effectuer une extraction en un cycle, et réarranger les valeurs dans l'arbre par décalages successifs des données le long du chemin critique, en autant de cycles qu'il y a de niveaux dans l'arbre. En d'autre termes, il n'est pas nécessaire que le réarrangement du chemin critique soit effectué immédiatement pour que l'échéancier fonctionne.
2. On peut effectuer une insertion de la même façon, en faisant progresser une donnée d'un niveau de l'arbre par cycle.
3. Ces deux opérations peuvent se "pipeliner" sans difficulté. Autrement dit, bien qu'une opération d'insertion ou d'extraction prenne plusieurs cycles pour être terminée, on peut enchaîner deux opérations successives à un cycle d'intervalle.
4. Pour chaque niveau du tas, un sommet au plus est concerné par une opération d'insertion, ou par une opération d'extraction à chaque cycle. Il suffit donc d'associer un processeur par niveau de l'arbre.

L'architecture proposée comporte n processeurs (voir figure 5.4), un par niveau (pour $2^n - 1$ valeurs). Le processeur de niveau i, noté PE_i, a une mémoire de $2^{(i-1)}$ cellules dont les adresses vont de 0 à $2^{(i-1)} - 1$. Chaque cellule contient une valeur, et un compteur donnant le nombre de cellules vides dans le sous-arbre correspondant. Le processeur contient en outre un registre appelé registre d'extraction, et dont l'usage sera précisé plus loin. Dans l'exemple de la figure 5.4, le processeur de gauche "gouverne" la totalité de l'arbre binaire, soit 15 valeurs. Le second processeur gouverne les deux sous-arbres fils de la racine de l'arbre, chacun comportant 7 valeurs.

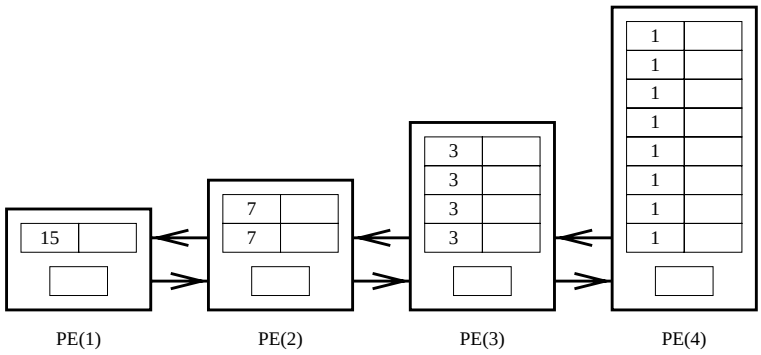


FIG. 5.4 – *Description de l'architecture pour $n = 4$, dans son état initial. chaque cellule contient une valeur (ici, les cellules sont vides), et un compteur donnant le nombre de cellules vides dans le sous-arbre qu'elle gouverne. La mémoire est représentée avec des adresses croissantes de bas en haut.*

5.3.1 Insertions

Les données sont insérées par le processeur de gauche. La donnée insérée, x, est traitée successivement par les processeurs 1 à n. Le processeur 1 garde la donnée, si son unique mémoire est vide (voir l'exemple de la figure 5.5). Si sa mémoire est déjà occupée par une valeur y, il conserve la valeur $\min(x, y)$ et transmet $\max(x, y)$ pour insertion à la cellule suivante (voir la figure 5.6). L'architecture fonctionne de façon alternée : lors d'un cycle "impair", les processeurs 1, 3, 5, etc. sont seuls actifs, et inversement, lors d'un cycle "pair", les processeurs 2, 4, 6, etc. sont seuls actifs.

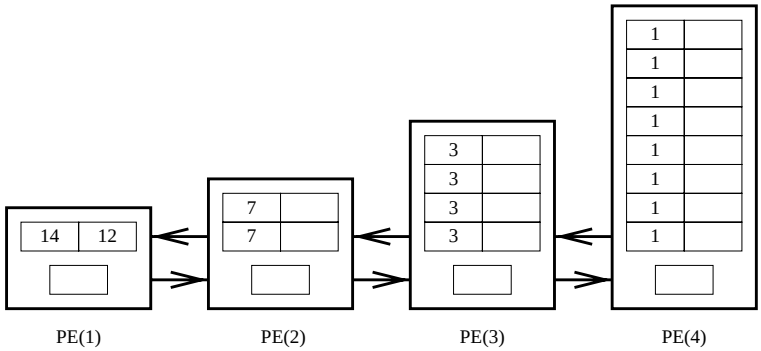


FIG. 5.5 – *Insertion de la valeur 12 dans l'échéancier. Cette valeur reste dans le processeur de gauche, et le nombre de cellules vides est diminué de 1.*

Le mécanisme le plus important n'apparaît qu'à partir de l'insertion d'une valeur dans le processeur 3. La figure 7 représente l'insertion de la valeur 15. Celle-ci, après être passée par les processeurs 1 et 2, doit être rangée dans la cellule 0 du processeur 3. L'adresse de cette cellule est déterminée par le processeur 2 : celui-ci, lors du passage de la valeur 15, a vu que le sous-arbre dont le sommet contient 12 (cellule 0), contient encore 6 places libres. Il indique donc au processeur 3 que la valeur 15 sera rangée dans le sous-arbre correspondant, à savoir soit les cellules 0 ou 1, si l'une d'entre elles est vide, soit dans un processeur suivant. En d'autres termes, le processeur a fourni une partie de l'adresse de la cellule de rangement, en excluant les cellules 2 et 3 du processeur 3, et d'une façon générale, la moitié supérieure des mémoires de tous les processeurs qui suivent. De cette façon, un processeur adresse au plus un groupe de deux cellules consécutives, et l'adresse de ce groupe de cellules lui est fournie par le processeur précédent, au moment de l'insertion. La figure 5.8 présente un cas d'insertion un peu plus complexe.

5.3.2 Extraction

L'extraction d'une valeur suit un mécanisme similaire, illustré par la figure 5.9. Cette figure reprend la situation atteinte après insertion de la valeur 21 (figure 5.8). Le premier processeur est toujours en mesure de fournir la valeur minimale. Il transmet l'ordre

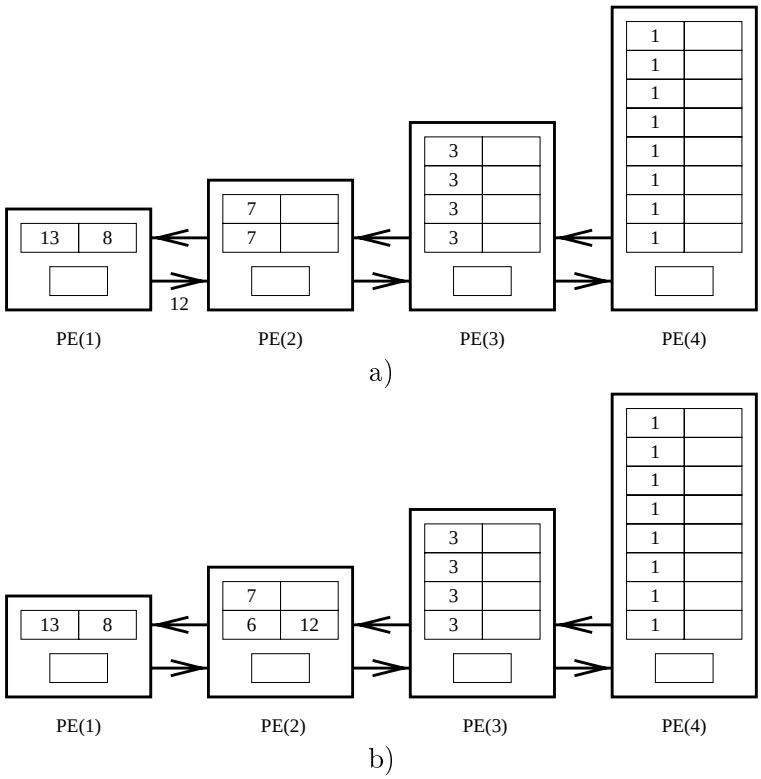


FIG. 5.6 – *Insertion de la valeur 8. Cette insertion a lieu en deux étapes : lors du cycle impair (a), le processeur 1, déjà occupé, donne la valeur 12 au processeur 2. Celui-ci, lors du cycle pair suivant (b), la mémorise dans sa cellule d'adresse 0.*

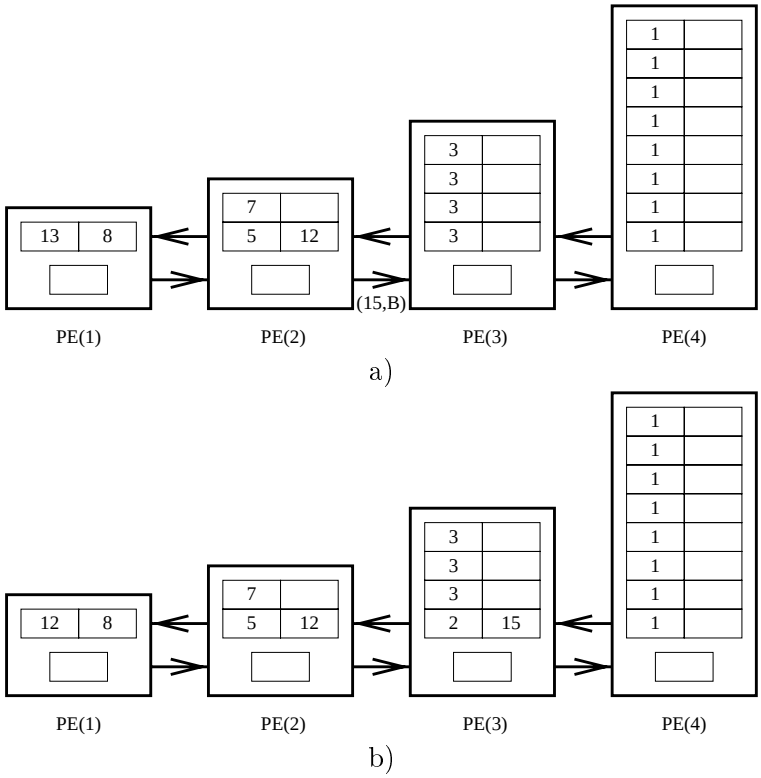


FIG. 5.7 – *Insertion de la valeur 15. Cette valeur est passée par les deux premiers processeurs. Le processeur 2 a déterminé que cette valeur sera rangée dans la moitié basse (B) des cellules des processeurs suivants (a). Au cycle suivant, 15 est rangée dans la cellule 0 du processeur 3.*

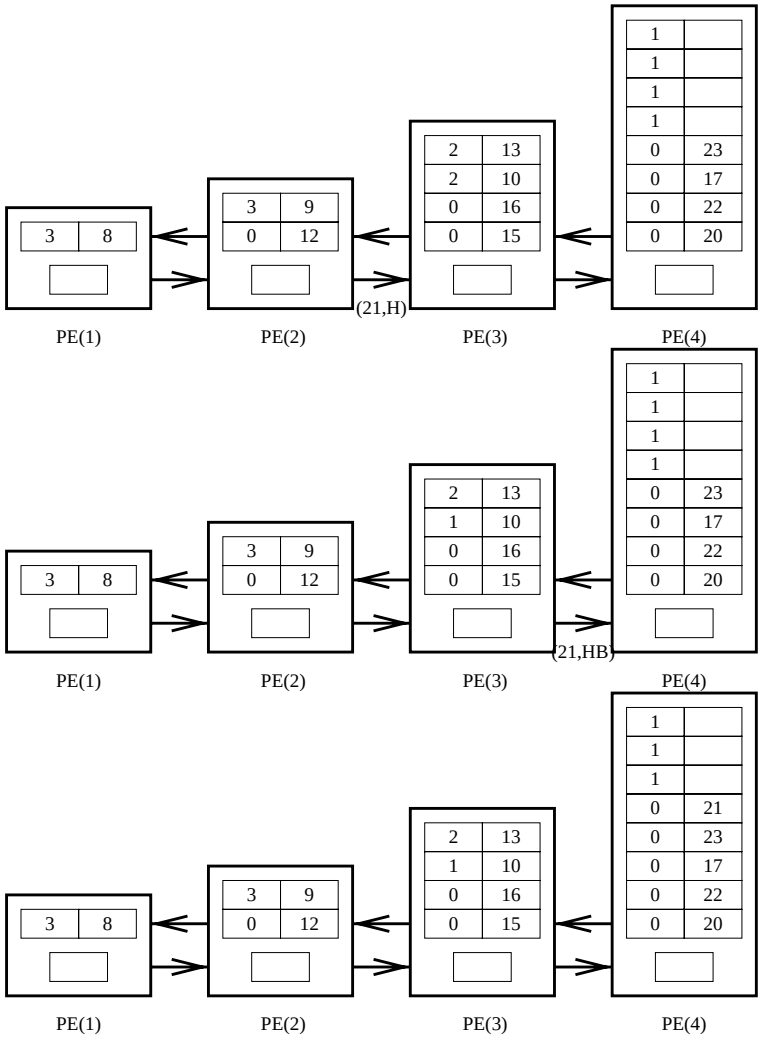


FIG. 5.8 – *Un exemple plus complexe. La valeur 21, après être passée par les processeurs 1 et 2, doit être insérée dans la moitié haute (H) de la mémoire (a). Au cycle suivant, le processeur 3 détermine qu'elle doit être rangée dans la partie basse de la partie haute (HB) (b). Elle est donc rangée dans la cellule 4 du processeur 4. Ce mécanisme fonctionne donc comme une adresse dont on fournit les bits en commençant par les poids forts.*

d'extraction au processeur suivant (figure 5.10), après avoir mémorisé dans son registre d'extraction l'adresse à laquelle la valeur extraite devra être placée. (Dans le cas du processeur 1, ce registre est inutile, mais nous l'avons conservé pour garder l'homogénéité avec le fonctionnement des processeurs suivants). Le processeur 2 fournit alors la valeur la plus petite de ses deux cellules. Il mémorise en même temps dans son registre d'extraction l'adresse à laquelle sera placée la valeur qui proviendra du processeur 3 ultérieurement.

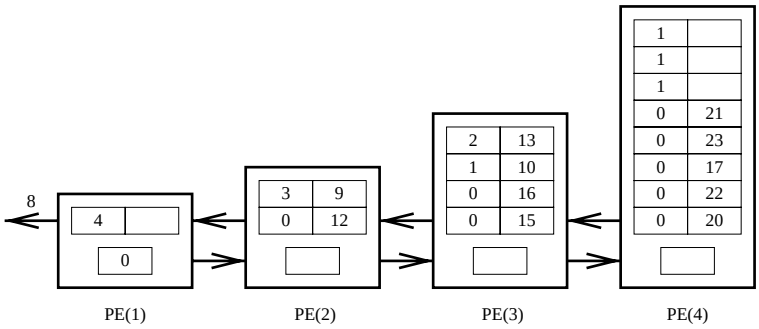


FIG. 5.9 – *Extraction d'une valeur par le premier processeur. La valeur extraite est 8.*

À l'étape suivante (figure 5.11), le processeur 2 envoie au processeur 3 un ordre d'extraction, en indiquant que celle-ci doit avoir lieu dans la partie haute (H) de la mémoire. Ce mécanisme d'adressage est identique à celui de l'insertion. Au cours du même cycle, le processeur 1 range la valeur 9 dans sa mémoire. Notons que ce processeur est prêt à effectuer une autre opération, qu'elle soit d'insertion ou d'extraction.

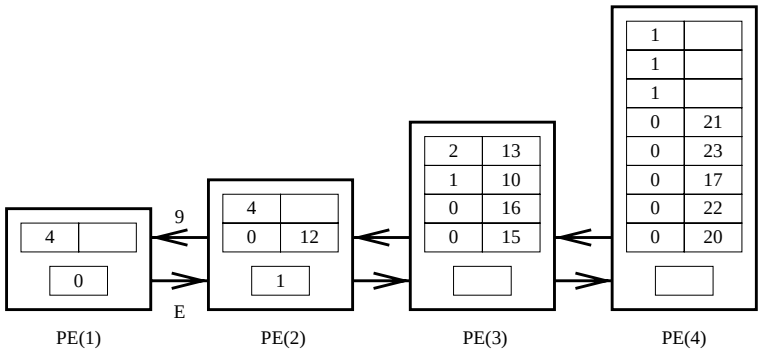


FIG. 5.10 – *Le processeur 2 fournit la valeur la plus petite qu'il contient, à savoir 9, au processeur 1. En même temps, il mémorise dans son registre d'extraction l'adresse, 1, où sera rangée la valeur extraite par le processeur 3.*

Le processeur 3 effectue une extraction dans la partie mémoire haute. Il délivre donc la valeur 10 au processeur 2, et note dans son registre d'extraction l'adresse de cette valeur 10, soit 2, comme adresse de rangement pour la valeur extraite du processeur 4.

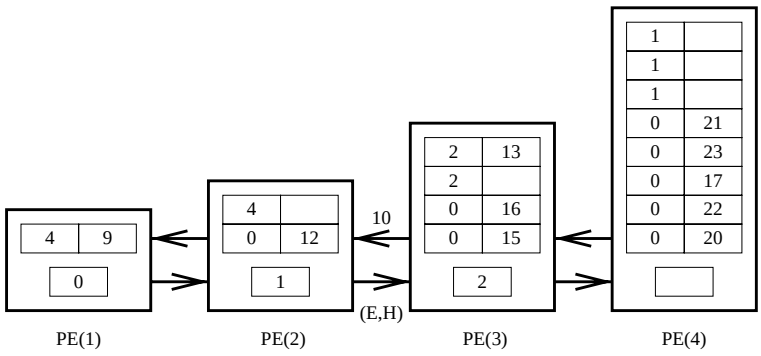


FIG. 5.11 – *Extraction par le processeur 3 de la valeur 10, et rangement par le processeur 1 de la valeur 9.*

La figure 5.12 illustre la fin de l'extraction. Le processeur 3 envoie au processeur 4 un ordre d'extraction en indiquant que la recherche doit être faite dans les cellules 4 et 5 (adresse HB). Le processeur 4 fournit donc la valeur 21, pendant que le processeur 2 range la valeur 10 (a). Ensuite, la valeur 21 est rangée dans la mémoire 2 du processeur 3 (b).

5.3.3 Pipeline

Les opérations d'insertion et d'extraction peuvent être effectuées de façon pipelinée, à raison d'une opération tous les deux cycles. En effet, les processeurs sont actifs seulement tous les deux cycles. Au cycle impair, les processeurs 1, 3, ..., $2p+1$, ..., et au cycle pair, les processeurs 2, 4, ..., $2p$, ...

Il est sans doute un peu moins évident de se convaincre que l'architecture fournit à tout moment la valeur minimale de celles qu'elle contient, alors même qu'une insertion peut être effectuée au cycle précédent, et continuer à affecter l'architecture pendant $\log_2 n$ cycle dans le pire cas. La raison est que dans tous les cas, le premier processeur contient la valeur minimale.

La figure 5.13 présente une situation d'insertion et d'extraction simultanée.

5.4 Utilisation de cette architecture

Cette architecture peut être utilisée pour mettre en œuvre l'algorithme de temps virtuel dans un réseau ATM. Plusieurs dispositifs ont été brevetés dans ce but par Siemens, LSI Logic, et le CNET. Aucun de ces dispositifs ne permet des opérations en temps constant. Un échéancier systolique du domaine public possède cette propriété, mais il nécessite 2^n processeurs (un par donnée). Le dispositif que nous proposons

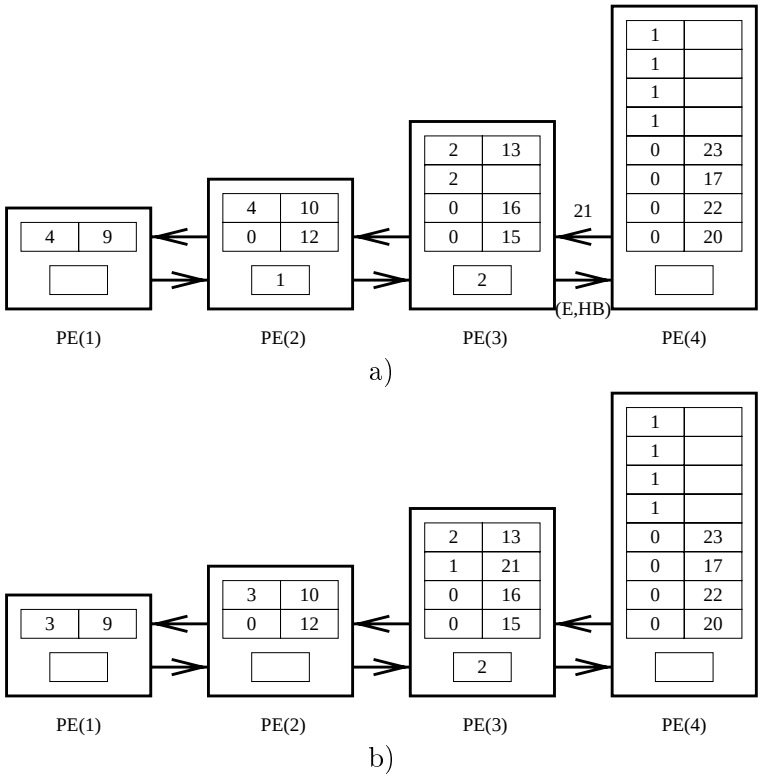


FIG. 5.12 – *Fin de l'extraction.*

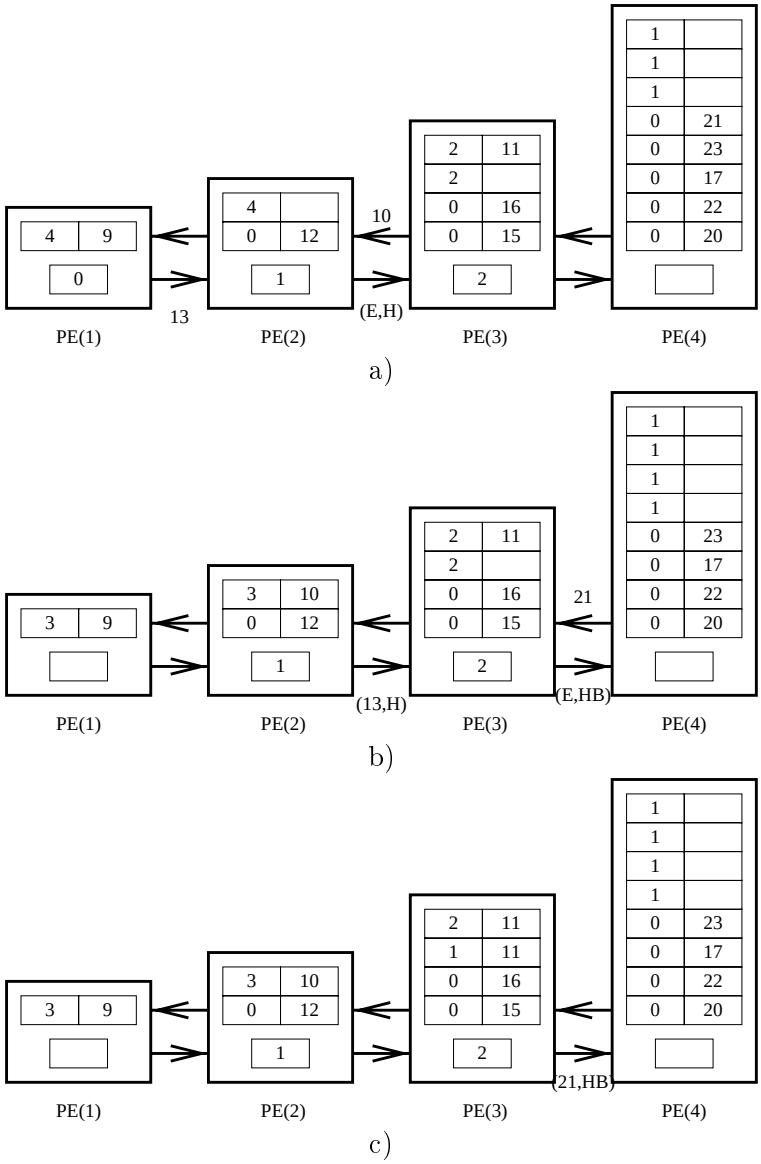


FIG. 5.13 – *Insertion et extraction simultanée. La valeur 13 arrive sur le processeur 2, en même temps que la valeur extraite 10, provenant du processeur 3. Le processeur 2 conserve la plus petite de ces valeurs, et envoie 13 au processeur 3. Au cycle suivant, celui-ci garde 13, et renvoit la valeur 21 au processeur 4.*

nécessite seulement n processeurs, chacun doté d'une mémoire nécessaire à un niveau de l'arbre. Une mise en œuvre sur circuit intégré est donc réaliste, car des mémoires sont beaucoup plus compactes que la logique pour les processeurs.

Dans un réseau ATM, les cellules qui sont reçues, doivent être réémises à une date appelée *heure théorique de réémission*. Pour cela, on dispose de deux modes de réémission :

- Le mode dit d'*espacement réel*. Dans ce mode, on attend que l'heure associée à la cellule soit arrivée à échéance. Dans un tel cas, l'heure théorique de réémission doit donc toujours être inférieure ou égale à l'heure courante.
- Le mode dit d'*espacement virtuel* dans lequel on n'attend pas l'échéance pour réémettre une cellule. Les cellules sont périodiquement émises tant que la mémoire de réémission n'est pas vide. Cependant, ces cellules sont émises dans le même ordre à savoir, de la plus petite heure d'émission à la plus grande.

Dans tous les cas, on voit qu'il est nécessaire de disposer d'un mécanisme capable de fournir à chaque instant la cellule ayant la plus petite heure afin qu'elle soit éventuellent émise. Un tel mécanisme aurait donc à gérer deux actions : l'*insertion* et l'*extraction* d'une cellule dans la mémoire prévue à cet effet. C'est donc à ce niveau que ce situe l'intérêt d'une file de priorité pouvant plus ou moins faciliter les opérations ci-dessus.

Plusieurs solutions ont été proposées pour résoudre de problème. Certaines sont du domaine public tandis que d'autres sont protégés par des brevets. La classification faite ici est inspirée de [44].

5.4.1 GDR [15, 137]

Le circuit GDR – pour *Gestion De Ressource* – permet de connaître, pour toute heure future, si une cellule lui est associée. Il permet aussi de trouver le premier élément libre ou occupé à partir d'une position donnée.

Un GDR de taille n represente donc les n prochaines heures d'émission futures sur le multiplex de sortie. Toute case C_i telle que $C_i = 1$ avec $0 < i < n$, porte une cellule à émettre au temps i. Toute case C_j telle que $C_j = 0$ avec $0 < j < n$, indique que le temps cellule j est libre.

En accès recherche, le GDR fournit la première adresse k du temps cellule libre à partir de l'adresse donnée i qui est par ailleurs égale à la partie entière inférieure de l'heure de réémission de la cellule courante. Une fois la cellule écrite à l'adresse k envoyée par le GDR, la case mémoire C_k est marquée occupée (i.e $C_k = 1$). Ici, les conflits pour un intervalle de temps donné sont résolus par l'ordre d'arrivée des cellules.

On voit donc qu'ici, le procédé d'*extraction* (en temps réel) est en temps constant tandis que l'insertion ne l'est pas. Au pire des cas l'*insertion* nécessite un temps égal à n (lorsque le GDR est plein). Le gros inconvénient dans le GDR réside dans son mode d'*insertion* car des retards de réémission peuvent être observés et une cellule peut ne pas avoir de place dans le GDR alors que celui-ci n'est pas plein.

Une implementation à été brevetée par le CNET.

5.4.2 GDR amélioré

Le dispositif de base est le même que le précédent mais sauf qu'ici, l'adresse d'une cellule est fonction de la partie entière de l'heure de réémission et d'une partie de sa décimale. Ceci réduit les possibilités de perte de cellule et les retards de réémission mentionnées dans le cas précédent car en fait on a une plus grande marge de manoeuvre dans le codage de l'heure de réémission du fait qu'on considère aussi la partie décimale de sa représentation.

5.4.3 Liste de pointeurs [161, 162]

Ce principe, breveté par SIEMENS, est une implémentation matérielle des listes chaînées couramment utilisées en logiciel.

Soit un tableau de taille fixe N. Chacune de ses cases correspond à une heure de réémission sur le multiplexeur de sortie. Chaque case contient un pointeur sur la liste de toutes les cellules qui se disputent l'intervalle de temps considéré. Lorsqu'une nouvelle cellule a été contrôlée, elle est ajoutée en queue de la liste de la case correspondant à son heure théorique de réémission.

La base de temps, incrémentée à chaque temps cellule, parcourt la liste principale. Si la case est marquée non vide, la chaîne complète est accrochée à la fin de la liste d'émission. Le pointeur sur la tête de liste d'émission désigne la prochaine cellule à émettre.

Ici, comme dans le cas du GDR, les conflits sont gérés par le principe FIFO. La complexité est du même ordre que dans le cas du GDR amélioré et les deux comportements sont semblables. Le gros inconvénient de ce dispositif est qu'il ne prévoit pas de déplacement du pointeur de lecture sur le tableau. Donc l'émission avec la technique d'espacement virtuel est impossible et de plus, il n'est pas possible d'insérer une cellule dans la liste de sortie. Par conséquent, l'*insertion* n'est pas possible à tout moment en respectant la priorité, ce qui viole le principe des échéanciers.

5.4.4 Les registres à décalage [23, 24]

Le principe ici consiste à implémenter en matériel le principe d'insertion d'un élément dans une liste triée. Il est propriété de Bell Lab's et protégé par un brevet.

La liste est en permanence triée suivant l'heure de réémission et représente donc l'ordre d'émission des cellules. A chaque émission, l'ensemble de la liste est décalée vers la droite. Lorqu'une nouvelle cellule arrive, son heure de réémission est diffusée à tous les éléments de la liste et comparée à ces derniers. Un encodage de priorité détermine l'emplacement où doit être insérée la cellule. Un décalage à gauche de tous les éléments permet de créer un emplacement pour enregistrer l'heure de réémission de la cellule.

L'élément de base du trieur est constitué d'un comparateur et d'un registre contenant l'heure de réémission et l'adresse de la cellule concernée.

Avec ce dispositif, l'ensemble des opérations est effectué en un temps d'horloge. Mais, le nombre de comparateurs est important, puisqu'il y en a un par donnée mémorisée.

5.4.5 Le tri systolique [25, 164, 166, 123, 150]

Il s'agit du tri systolique classique *odd-even* qui utilise $\frac{n}{2}$ éléments de base (comparateur + registre) pour trier n valeurs. Chaque élément envoie à sa gauche la plus petite des deux valeurs qu'il possède et à droite la plus grande. A un instant donnée, la liste n'est pas nécessairement triée mais la plus petite valeur se trouve toujours à gauche du premier processeur.

Ainsi, l'*extraction* est immédiate tandis que l'insertion ne l'est pas mais elle est couverte par le mécanisme de pipeline.

L'inconvénient majeur de ce dispositif est le nombre important de comparateurs qu'il requiert. Notons tout de même que le pipeline de l'insertion permet de ce fait de mener plusieurs insertions sans que cela ne nuise à une éventuelle *extraction*.

5.4.6 Arbre binaire [77]

Il s'agit d'une application du tri par *tas*. Dans un premier temps, on constitue un *tas*, c'est à dire un arbre binaire dans lequel le plus petit élément de chaque sous-arbre se trouve dans sa racine. De ce fait, le plus petit élément se trouve à la racine de l'arbre principal. Chaque cellule qui arrive est insérée de manière à concerver la propriété du *tas*.

La performance du trieur est en $n\log_2(n)$, mais une procedure d'insertion ou d'extraction s'exécute en $\log_2(n)$ opérations. En effet, une cellule est extraite et une celulle est insérée par temps cellule.

5.4.7 Mémoire associative

Le principe consiste à utiliser des mémoires associatives dont le principe est d'accéder aux emplacements mémoires par le contenu et non par les adresses. Ceci permet de réduire la complexité des éléments de base car le contenu d'une cellule peut être retrouvé uniquement à partir de son heure de réémission.

Ce mécanisme ne permet pas un espacement virtuel car les mémoires associatives fonctionnent par égalité et non par comparaison. Pour y remédier, il faudrait donc placer un comparateur par élément, ce qui conduit encore à un nombre de comparateurs important si on veut réaliser une *extraction* à chaque top d'horloge.

5.4.8 Tri hiérarchique

Le principe de base est un arbre dont chaque nœuds est un comparateur, seules les feuilles portent les heures de réémissions à trier. De proche en proche, la plus petite heure de réémission va se retrouver au sommet de l'arbre d'où elle sera extraite.

L'*extraction* n'est malheureusement pas faite en un temps d'horloge. De plus, le nombre de comparateurs est important.

5.4.9 Avantages de notre solution

La solution que nous présentons possède les avantages suivants :

1. L'*extraction* est immédiate, et sa durée ne dépend pas du nombre d'éléments mémorisés.

2. Les insertions peuvent être concurrentes grâce au mécanisme de pipeline.
3. Le temps d'insertion est raisonnable ($\log_2(n)$ cycles).
4. On peut faire une *extraction* pendant que des insertions sont en cours, sans attendre qu'elles soient totalement terminées.
5. Le nombre de comparateurs est raisonnable ($\log_2(n)$) et varie très peu avec la taille des données, de sorte qu'une implémentation sous forme de circuit spécialisé est envisageable et donnerait lieu à un dispositif robuste.
6. Le dispositif est modulaire en ce sens qu'il peut facilement être adapté pour traiter des données en plus grand nombre.
7. On peut à tout moment savoir si la file est vide (pour stopper les *extractions*) ou pleine (pour stopper les *insertions*).

5.5 Historique et conclusion

Le résultat que nous avons présenté dans ce chapitre a été inspiré au départ d'un algorithme de tri par tas que nous avions mis au point durant l'année 1995 dans notre travail de maîtrise dirigé par Maurice Tchuente. L'idée de l'agorithme était de considérer le tri par tas classique, de le décomposer en une suite d'opérations élémentaires identiques, et d'allouer chaque couche de la structure arborescente à un processeur. On obtenait ainsi un réseau systolique linéaire de $log(n)$ processeurs, capable de trier n valeurs en temps $2n$. Deux années plus tard, ce résultat sera présenté à Patrice Quinton lors d'un cours de DEA. Patrice Quinton avait quelques mois plutôt réalisé l'intérêt des échéanciers rapides, à travers des contacts avec le CNET de Lannion et le suivi de la thèse de Olivier Dugeon [44]. C'est ainsi que nous avons entrepris de transformer l'algorithme initial en un échéancier systolique tel que nous l'avons présenté ici. Ceci nous a conduit à l'idée d'un brevet d'invention, lequel n'a pu être déposé qu'en janvier 1999 (Document INPI numéro 990540). Indépendamment de ce travail, Ha-duong et Moreira avaient déposé un brevet en mai 1997 [67], mais pendant une période de 18 mois, les inventions ne sont pas divulguées et au moment de la recherche d'antériorité (septembre 1988), ce brevet n'était pas encore public (il ne l'est devenu qu'en novembre 1988). Cet algorithme a été ensuite publié par ses auteurs à Colmar en juin 1999 [68], et est actuellement utilisé dans des commutateurs ATM de la société Ericsson.

La description qui est donnée ici fait explicitement reférence au concept d'algorithme systolique. Le tri systolique, connu depuis la fin des années 1970, peut être vu comme une adaptation d'un registre à décalage, en pipelinant l'insertion d'une clé afin de rendre constante la durée d'une insertion. En remarquant qu'on peut faire fonctionner cette architecture comme un échéancier parce qu'il n'est pas nécessaire d'attendre la fin d'une insertion ou d'une extraction avant d'entamer une nouvelle opération, on obtient la propriété d'opération en temps constant, propriété fondamentale pour une implémentation matérielle. De la même façon, l'écheancier présenté ici tire parti de cette remarque: il n'est pas nécessaire d'attendre que le tas soit stabilisé pour effectuer une nouvelle opération. Une importante leçon qu'on peut retenir c'est qu'un principe relativement ancien a pu trouver une application concrète deux à trois décennies plus tard.

Conclusion

Globalement, l'ensemble des travaux présentés dans cette thèse se situe autour de deux principales problématiques. En amont, nous devions étudier les différentes méthodologies de conception d'algorithmes parallèles, ainsi que les différentes approches d'analyse de complexité. En aval, nous devions nous pencher sur des problèmes précis pour lesquelles une solution parallèle était attendue. Il ressort de ces études que divers points de vue cohabitent en général autour des problèmes de construction d'algorithmes parallèles, et leur appréciation dépend fortement de la nature de la solution et même de son origine. Nous avons aussi remarqué que dans certains cas, une reformulation du problème à paralléliser peut permettre d'accroitre l'efficacité de l'algorithme résultant. Ceci justifie alors le fait que le parallélisme est une discipline dans laquelle la pluridisciplinarité est un facteur particulièrement appréciable. D'un autre côté, le point de vue matériel n'est pas à négliger car même si l'on dispose d'un algorithme théoriquement efficace, réaliser une bonne implémentation est une tâche non moins évidente, qui peut nécessité de bien connaitre le comportement de la machine cible. Bien que des efforts de standardisation visant à réduire cette influence donnent déja des résultats notables, il reste quelques pas à franchir avant d'en arriver à une situation où les choses sont complètement maitrisées.

Concernant le problème du produit tensoriel, nous sommes parvenu à construire un schéma algorithmique permettant une implémentation parallèle efficace. Notre solution a été expérimentée sur des machines parallèles telles que l'INTEL PARAGON, la NEC CENJU3, et la CRAY T3E. D'un point de vue théorique, nous avons établi des bornes inférieures sur le surcoût de temps dû aux communications, et montré que nos algorithmes pouvaient atteindre ces performances optimales. Tel sera le cas lors de nos expérimentations, de sorte que, des chercheurs de l'ex-équipe MODEL ont trouvé en cette solution, une motivation pour poursuivre certains objectifs de calculs auparavant assez difficiles à gérer.

A propos du problème du *chemin algébrique*, sujet ayant fait l'objet d'intenses recherches, nous avons développé des solutions efficaces et modulaires sur des réseaux linéaires. Un aspect que nous avons jugé assez spécial et important fut l'étude des versions en blocs. En effet, nos solutions étaient toutes au départ dédiées à des implémentations sur VLSI, c'est à dire qu'elles étaient à grains très fins. Sur ce type d'architecture, de telles approches se justifient par le fait que les communications ont un coût presque nul et sont parfaitement synchronisées avec les opérations scalaires. Sur des machines parallèles standards, ces solutions deviennent inappropriées du fait du coût considérable des

communications, lequel devient une expression affine du volume des données. Dériver des versions en blocs de nos algorithmes initiaux constituait alors un recours naturel pour passer du modèle VLSI au modèle standard. Toutefois, les performances dépendant fortement de la taille des blocs, nous avons été confrontés à un problème d'optimisation non linéaire que nous avons pu résoudre chaque fois grâce à des hypothèses simplificatrices judicieuses. Des expérimentations sur les machines INTEL PARAGON et CRAY T3E nous ont permi de valider notre étude.

S'agissant de l'échéancier systolique, ce fut une expérience assez particulière, un parfait mélange de recherche et de gestion des enjeux. Sur le plan scientifique, le passage de la solution initiale (qui était un réseau de tri) à un échéancier systolique fut une activité à fort potentiel pédagogique. On comprend une fois de plus que la position d'un résultat scientifique résulte d' une conjonction entre une idée et une motivation bien choisie.

Toutes ces expériences, y compris celles issues de nos travaux dans le domaine des mathématiques discrètes et combinatoires, nous auront aussi permis de mettre au point une méthodologie de conception d'ordonnancements parallèles, dénommée *ordonnancement canonique*. Cette technique, d'un aspect analytique et combinatoire, semble permettre de faire face à une large classe de problèmes.

Nous pensons qu'à travers ce travail, nous pensons avoir fait un pas dans la compréhension des activités et des enjeux du calcul parallèle, et aussi dans les disciplines algorithmiques et combinatoires, en passant par le calcul scientifique, et plus généralement le calcul haute performance. Nous souhaitons poursuivre ces efforts en élargissant les horizons autant que possible, mais tout en suivant un chemin où chaque chose s'ajoute à une autre.

5.6 Bibliographie personnelle

5.6.1 Articles dans les Journaux à commité de lecture

1. **C. Tadonki** and B. Philippe, *Parallel Multiplication of a vecteur by a Kronecker product of matrices*, PDCP, (4)2, 2000.
2. **C. Tadonki** and B. Philippe, *Parallel Multiplication of a vecteur by a Kronecker product of matrices (Part II)*, PDCP, To appear.
3. **C. Tadonki**, *Synthèse d'ordonnancements parallèles par reproductions canoniques*, Calculateurs Parallèles, A Paraitre.
4. R. Ndoundam, M. Tchuente, and **C. Tadonki**, *Parallel chip firing game associated with ncube orientations*, Discret Mathematics, A paraître.
5. D. Cachera, S. Rajopadhye, T. Risset, and **C. Tadonki**, *Parallelization of the algebraic path problem on linear simd/spmd arrays*, (soumis à IEEE TPDS)
6. S. Rajopadhye, **C. Tadonki**, et T. Risset, *Le problème algébrique sur réseaux linéaires*, (soumis à TSI)

5.6.2 Publications dans les Conférences

1. **C. Tadonki**, *Equation récurrente et algorithmes parallèles pour la multiplication d'un vecteur par un produit tensoriel de matrices*, Renpar11, Rennes, 8-11, juin 1999.
2. S. Rajopadhye, **C. Tadonki**, and T. Risset, *The algebraic path problem revisited*, Europar99, Toulouse, Lncs, No 1685, pp. 698-707, Août 1999.
3. **C. Tadonki**, *Ordonnancements canoniques*, Renpar2000, Besancon, 19-23 juin 2000.
4. **C. Tadonki**, *Parallel Cholesky factorization*, Workshop on parallel matrix algorithm and application, Neuchatel, 18-21 Août 2000.
5. P. Quinton, **C. Tadonki**, M. Tchuente, *Un échéancier systolique et son utilisation dans l'ATM*, CARI2000, Madagascar, 16-19 Octobre 2000.
6. **C. Tadonki**, et B. Philippe, *Méthodologie de conception d'algorithmes efficaces pour le produit tensoriel*, CARI2000, Madagascar, 16-19 Octobre 2000.

5.6.3 Rapports de Recherche

1. **C. Tadonki** and B. Philippe, *Parallel multiplication of a vector by a kronecker product of matrices*, RR IRISA No , juillet 1998.
2. D. Cachera, S. Rajopadhye, T. Risset, and **C. Tadonki**, *Parallelization of the algebraic path problem on linear simd/spmd arrays*, RR INRIA/IRISA No 1346, Août 2000.
3. P. Quinton, **C. Tadonki**, M. Tchuente, *Un échéancier systolique et son utilisation dans l'ATM*, PI IRISA No 1348, Août 2000.
4. **C. Tadonki**, *Synthése d'ordonnancements parallèles par reproductions canoniques*, RR INRIA/IRISA No 1349, Août 2000..

Bibliographie

1. T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, and M. Snir, *SP2 system architecture*, Scalable Parallel Computing, Vol. 34, No. 2, 1995.
2. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
3. Y. Amir, B. Awerbuch, A. Barak, R.S. Borgstrom, and A. Keren, *The Java Market: Transforming the Internet into a Metacomputer*, Technical report CNDS-98-1, 1998.
4. G. Authier, A. Ferreira, J. L. Roch, G. Villard, J. Roman, C. Roucairol, B. Virot, *Algorithmes parallèles - Analyse et conception*, Hermes, 1994.
5. M. Baker, *Cluster Computing Review*, NPAC Technical Report SCCS-748, 1995.
6. U. Banerjee, *AN introduction to a formal theory of dependence analysis*, The Journal of Supercomputing, 2:133-149, 1988.
7. U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Acdemic Publisher, New York, USA, 1988.
8. D.F. Bacon, S.L. Graham, and O.J. sharp, *Compiler Transformations for High-Performance Computing*, Hermes, 1994.
9. L. M. Baptist, and T. H. Cormen *Multidimensional, Multiprocessor, Out-of-Core FFTs with Distributed Memory Parallel Disks.*, SPAA'99, Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, 1999.
10. A. Benaini, P. Quinton, Y. Robert, Y. Saouter, and B. Tourancheau. Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 15:135–158, 1990.
11. A. Benaini and Y. Robert. Space-time minimal systolic arrays for gaussian elimination and the algebraic path problem. In *ASAP 90: International Conference on Application Specific Array Processors*, pages 746–757, Princeton, NJ, September 1990. IEEE Press.
12. C. Berge, *Graphes et Hypergraphes*, Dunod, 1970.
13. G. E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, 1990.
14. G. E. Blelloch and G.W. Sabot, *Compiling collection-oriented languages onto massively parallel computers*, Journal of Parallel and Distributed Computing, 8(2), 1990.
15. P. Boyer, M. Servel, and F. Guillemin. The Spacer-Controller: an efficient UPC/NPC for ATM Netwoks. In *Proceedings of ISS'92*, number A9.3 in volume 2, pages 316–320, Yokohama, Japan, October 1992.
16. Gilles Brassard et Paul Brathley, *Algorithmes: Conception et Analyse*, Masson, 1982.
17. R.P. Brent, *The parallel evaluation of arithmetic expressions in logarithmic time*, Dans Compexity of Sequential and Parallel Numerical Algorithms, pp. 83-102, J.F. Traub (ed.), Academic Press, New York, 1973.
18. R. P. Brent, *Some Parallel Algorithm for Integer Factorization.*, Europar'99 Parallel Processing, Lecture Note in Computer Science, No. 1685, pages 1-22, Springer Verlag, 1999.
19. P. Brucker, *Scheduling Algorithms*, Springer, 1998.

20. K. Bromley, S.Y. Kung, and E. Swartzlander (eds.), *Internation Conference on Systolic arrays*, IEEE Computer Society Press, 1982.
21. D. Cachera, S. Rajopadhye, T. Risset, and C. Tadonki, *Parallezation of the Algebraic Path Problem on Linear SIMD/SPMD Arrays*, Rapport de recherche INRIA/IRISA no 1346, Août 2000 (Soumis à IEEE TPDS).
22. P. Y. Chang and J. C. Tsay. A family of efficient regular arrays for the algebraic path problem. *IEEE Transactions on Computers*, 43(7):769–777, July 1994.
23. J. Chao. A general architecture for link-layer congestion control in ATM networks. In *Proceedings of ISS'92*, number P14 in volume 1, pages 229–223, Yokohama, Japan, October 1992.
24. J. Chao. A novel architecture for queue management in the ATM network. *IEEE Journal Selec. Areas in Commum.*, 9(7), October 1992.
25. H.-H. Choi and M. Malek. A fault-tolerant VLSI sorter. In *Proceedings of ICCD'85*, October 1985.
26. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, MIT Press, Cambridge, Massachussetts, 1990.
27. P. Chrétienne, *Scheduling and Parallelism*, Dans Algorithmique Parallèle, pages 297-312,Masson, 1992.
28. Lee-Chung Lu, *A unified framework for systematic loop transformation*, In Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 28-38, April 1991.
29. A. Church, *An unsolvable problem in elementary number theory*, American Journal of Mathematics (58), pp. 345-363, 1936.
30. P. Clauss, *Synthèse d'Algorithmes Systoliques et Implantation Optimale en place sur Réseaux de Processeurs Synchrones*, Thèse de Doctorat de l'Université de Franche-Comte, 1990.
31. P. Clauss, C. Mongenet, and G-R. Perrin. Synthesis of size-optimal toroidal arrays for the algebraic path problem: A new contribution. *Parallel Computing*, 18:185–194, 1992.
32. S. A. Cook, *Towards a complexity theory of synchronous parallel computation*, L'Enseignement Mathématique II 27(1-2), pp. 99-124, 1981.
33. S. A. Cook and H.J. Hoover, *A depth-universal circuit*, SIAM Journal of Computing, 14(4):833-839, 1985.
34. M. Cosnard, M. Nivat, et Y. Robert, *Algorithmique parallèle*, Masson, 1992.
35. M. Cosnard et D. Trystram, *Algorithmes et architectures parallèles*, InterEditions, 1993.
36. P. Chrétienne and C. Picouleau, *The Basic Scheduling Problem with Interprocessor Communication Delays*, RR MASI, 91-06, 1991.
37. L. Dagum and R. Menon, *OpenMP: An industry-standard API for shared-memory programming*, IEEE Computational Science and Engineering, vol. 5, 1, pp. 46-55, 1998.

38. G. B. Dantzig, *Linear Programmimg and Extensions*, Princeton University Press, 1963.
39. A. Darte, *Techniques de parallélisation automatique de nids de boucles*, Thèse de Doctorat, ENS Lyon, 1993.
40. A. Darte, T. Risset, and Y. Robert, *Synthetizing systolic arrays: some recent developments*, In M. Valero et al., editors, Application specific arra processors, pages 372-386. IEEE Computer Society Press, 1991.
41. M. Davio, *Kronecker Products and Shuffle Algebra.*, IEEE Trans. Comput., Vol. C-30, No. 2, pp. 116-125, 1981.
42. J.J. Dongarra, H.W. Meuer, and E. Strohmaier (eds.), *Supercomputer 60/61*, Supercomputer, Special Issue THE 1994 TOP500 REPORT, vol. 11, no. 2/3 (voir aussi www.enseeiht.fr/NetLib/benchmark/top500/), juin 1995.
43. Y. Dinitz, S. Even, R. Kupershtok, and M. Zapolotsly, *Some Compact Layout of the Butterfly*, In SPAA'99, Saint-Malo, France, pp. 54-63, 1999.
44. O. Dugeon. *Machine d'admission pour le réseau d'infrastructure ATM.* PhD thesis, Rennes, France, November 1997.
45. H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, 1994.
46. R. Faure, C. Roucairol, et P. Tolla, *Chemins et flots, ordonnancements*, Gauthier-Villard, 1976.
47. P. Feautrier, *Parametric integer Programming*, RAIRO: Recherche Opérationnelle, 22:243-268, , 1988.
48. P. Feautrier, *Automatic Parallelization in the Polytope Model*, —-, 1999.
49. P. Feautrier, J.F. Collard, M. Barreteau, D. Barthou, A. Cohen, V. Lefebvre, *L'interaction expansion-ordonnancement dans PAF*, Rapport technique du PRISM, 1998.
50. P. Fernandes, B. Plateau, and W. J. Stewart, *Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks*, INRIA Rapport de recherche interne No. 2935 , July 1996.
51. M. J. Flynn, *Very High Speed Computing Systems*, In proceedings IEEE (12), pp. 1901-1909, 1966.
52. I. Forster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, In the International Journal of Supercomputer Applications, 11(2):115-128 (also www.globus.org/research/papers.htm), 1997.
53. J. A. B. Fortes, K. S. Fu, and B. W. Wah, *Systematic approaches to the design of algorithmic specified systolic arrays*, in Proc. IEEE/ICASSP, Tampa,Mar. 26-29, 1985.
54. S. Fortune and J. Wyllie, *Parallelism in Random Access Machines*, In proceedings of STOC-10, pp. 114-118, 1978.
55. T.J. Foutain, M.J. shute (eds.), *Multiprocessors Computer Architecture*, North-Holland, 1990.
56. F. Galilée, J.L. Roch, G.G.H. Cavalhero, and M. Doreille, *Athapascan-1: On-Line*

Building Data Flow Graph in a Parallel Language, International Cobference on Parallel Architectures and Compilation Techniques, Paris, Octobre 1998.

57. M. R. Garey, D. S. Johnson, R. E. Tarjan, and M. Yannakakis, *Scheduling Opposing Forests*, SIAM J. Alg. Disc. Math., 4, no 1, pp. 72-92, 1983.
58. M. R. Garey, D. S. Johnson, *Computer and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman, New-York, USA, 1979.
59. C. Germain and J.P. Sansonnet, *Les ordinateurs massivement parallèles*, Armand Colin, 1991.
60. A. Geist, A. Beguelin, J. Dongara, W. Jiang, R. Manchek, V. Sunderam, *PVM:Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.
61. Marc Gengler, Stéphane Ubéda, Frédéric Desprez, *Initiation au parallélisme*, Masson, 1995.
62. F. W. Glover (ed.), *Journal of Heuritics*, www.wkap.nl/journalhome.htm/1381-1231.
63. J. Granta, M. Conner, and R. Tolimieri, *Recursive fast algorithms and the role of the tensor product*, IEEE Transaction on Signal Processing, 40(12):2921-2930, December 1992.
64. L. Guibas, H. T. Kung, and Clark D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Conference on Very Large Scale Integration: Architecture, Design and Fabrication*, pages 509–525, January 1979.
65. J.L. Gustafson, *Reevaluating Amdhal's Law.*, Communication of the ACM, 31(5):532-533, 1988.
66. M. Griebl, and C. Lengauer, *The Loop Parallelizer LooPo-Announcement*, D. C. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, D. A. Padua (Eds.): Languages and Compilers for Parallel Computing, 9th International Workshop, LCPC'96, San Jose, California, USA, August 8-10, 1996, Proceedings. Lncs, Vol. 1239, pp. 603-604, Springer, 1997.
67. T. Ha-Duong, et S. Moreira , *Dispositif de tri d'éléments de données à arbre binaire et espaceur ATM comportant un tel dispositif.*, Brevet national INPI numero 9705828, mai 1997.
68. T. Ha-Duong, et S. Moreira, *The Heap-Sort Based ATM Cells spacer.*, $2^n d$ In. Conf IEEE on ATM, Université d'Alsace, Colmar, France, juin 1999.
69. J.L. Hennessy and D.A. Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann, chap. 3, 1996.
70. W.D. Hillis, *The Connection Machine*, MIT Press, 1985.
71. H. Horos et B. Roy, *Problèmes de circulation et flots dans un graphe*, Dunod, 1970.
72. K. Högstedt, L. Carter, and J. Ferrante, *Selecting tile shape for minimal execution time*, Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'99, pp. 201-211, May 1987.
73. F. Irigoin, and R. Triolet, *Supernode partitioning*, In Symposium on Principles of Programming Languages, pages 319-328, January 1988.

74. R. M. Karp, *Reductibility Among Combinatorial Problems*, R.E. Miller and J.W. Thatcher(eds.), Complexity of Computer Computations, Plenum Press, New York, pp. 85-103, 1972.
75. R. M. Karp, R.E. Miller, and S. Winograd, *The organization of computations for uniform recurrences equations*, Journal of the ACM, 14(3):563-590, 1990.
76. R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, Chapter 17, pp. 869-941, Handbook of Theoretical Computer Science, J. Van Leeuvwen (ed.), MIT Press, 1990.
77. D. Knuth. *The Art of Computer Programming, Seminumerical Algorithms, 2nd edition*, volume 3. Addison-Wesley, 1969.
78. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L.Jr. Seele, and M. E. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, USA, 1994.
79. C.P. Kruskal, L. Rudolph, and M. Snir, *A complexity theory of efficient parallel algorithms*, Theoretical Computer Science, 71, pp. 95-132, 1990.
80. H. T. Kung, *Systolic Communication*, In International Conference on Systolic Arrays, pp. 695-703, May 1988.
81. H. T. Kung and C. E. Leiserson, *Algorithms for VLSI Processor Arrays*, Dans Introduction to VLSI Systems, Addison-Wesley, 1980.
82. S. Y. Kung, S. C. Lo, and P. S. Lewis, An optimal systolic design for the transitive closure and the shortest path problems. *IEEE Transactions on Computers*, C-36(5):603–614, May 1987.
83. S. Y. Kung and S. C. Lo. A spiral systolic algorithm/architecture for transitive closure problems. In *ICCD 85: International Conference on Circuit Design*, pages 622–626, Rye Town, NY, 1985. IEEE.
84. J. Kuntzmann, *Theorie des réseaux*, Dunod, 1972.
85. L. Lamport, *The parallel execution of DO Loops*, Communication of the ACM, 17:83-93, 1974.
86. H.W. Lawson, *Parallel Processing in Industrial Real-Time Application*, Prentice-Hall, 1992.
87. F. T. Leighton, *Introduction to Parallel Algorithm and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kaufmann, San Mateo CA, 1991.
88. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan, and D.B. Shmoys, *Sequencing and Scheduling Algorithms and Complexity*, Report BS-R8909 CWI, K.M. Van Hee, and H.G. Sol, Eindhoven University of Technology, 1989.
89. T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, Englewood Cliffs, USA, 1992.
90. P. Lignelet, *Manuel complet du langage Fortran90 et Fortran95*, MASSON, 1996.
91. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, The International Journal of Supercomputer Applications and High Performance Computing,8, 1994.
92. D. I. Moldovan, *On the design of algorithms for VLSI systolic arrays*, Proc. IEEE, vol. 71, Jan. 1983.

93. C. Mongenet, P. Clauss, and G. R. Perrin, *Geometrical tools to map systems of affine recurrence equations on regular arrays*, LIB Report, pp. 30-90, 1990.
94. C. Mongenet, P. Clauss, and G. R. Perrin, *Implantations optimales d'équations récurrentes affines*, Dans Algorithmique Paralèle, pages 331-341,Masson, 1992.
95. R. E. Morley and T. J. Sullivan, *A Massively Parrallel Systolic Array Processor System*, In International Conference on Systolic Arrays, pp. 217-225, May 1988.
96. H.S. Morse, *Practical Parallel Computing*, AP Professional, 1994.
97. J. F. Myoupo and C. Fabret. A modular systolic linearization of the warshall-floyd algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):449–455, may 1996.
98. J. G. Nash and S. Hansen. Modified faddeew algorithm for matrix multiplication. In *Proc., SPIE Workshop on Real-Time Signal Processing*, pages 39–46. SPIE, 1984.
99. R. Ndoundam, M. Tchuente, and C. Tadonki, *Parralel Chip Firing Game Associated with n-cube Orientations*, Soumis a Discret Mathematics, 1999.
100. Omega, *The Omega Project*, http://www.cs.umd.edu/projects/omega/.
101. OpenMP, *OpenMP: A proposed industry standard api for shared memory programming*, http://www.openmp.org/.
102. P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.
103. C. Papadimitriou, and M. Yannakakis, *Scheduling interval-ordered taks*, SIAM Journal of Computing, 8, 405-409, 1979.
104. I. Parbery, *Parallel Complexity Theory*, Pitman, London, 1987.
105. G.F. Pfister, *In Search of Clusters- The Coming Battle in Lowing Parrallel Computing*, Prentice-Hall, 1995.
106. C. Picouleau, *Two new NP-Complete Scheduling Problems with Communication Delays and Unlimited Number of Processors*, RR IBP-MASI,91-24, 1991.
107. http://www.cri.ensmp.fr/ pips/index.html.
108. B. Plateau, *On the stochastic structure of parallelism and synchronization models for distributed algorithms*, In ACM Sigmetrics Conference on Measurement and Modelling of Computer systems, Austin,pp. 147-154, August 1985.
109. C.D. Polychronopoulos, *Parallel Programming and Compiler*, Kluwer Acdemic Publicher, New York, USA, 1990.
110. V. K. Prasanna Kumar and Y-C Tsai. Designing linear systolic arrays. *Journal of Parrallel and Distributed Computing*, (7):441–463, may 1989.
111. W. Pugh and D. Wonnacutt, *Static analysis of upper and lower bounds on dependences and paralelism*, ACM Transactions on Programming Languages and Systems, 1993.
112. P. Quinton, C. Tadonki, et M. Tchuente *Un échéancier systolique et son utilisation dans l'ATM*, Publication interne IRISA no 1348 (A paraitre dans le document de CARI2000), Août, Octobre 2000.
113. P. Quinton, *The systematic design of systolic arrays*, Automata networks in Computer Sciences, pp. 229-260, Manchester University Press, 1987.

114. P. Quinton et Yves Robert, *Algorithmes et architectures systoliques*, Masson, Paris, 1989.

115. Patrice Quinton, *Automatic synthesis of systolic arrays from uniform recurrent equations*, in Proc. 11th Annu. Symp. Comput. Architecture, pp. 208-214, 1984.

116. S. V. Rajopadhye, *Synthesis, Verification and Optimization of Systolic Arrays*, PhD Thesis, The University of Utah, Deptartment of Computer Science, December 1986.

117. S. V. Rajopadhye and Richard M. Fujimoto, *Synthetizing systolic arrays from recurrence equations*, Parallel Computing **14** pp. 163-189, June 1990.

118. S. V. Rajopadhye. An improved systolic algorithm for the algebraic path problem. *INTEGRATION: The VLSI Journal*, 14(3):279–296, Feb 1993.

119. Sanjay Rajopadhye, Claude Tadonki, and Tanguy Risset, The algebraic path problem revisited. In *Lecture Note in Computer Science: EuroPar'99 Parallel Processing*, Springer-Verlag, No. 1685, pages 698–707, August 1999.

120. J. Ramanujam and P. Sadayappan, *Tiling multidimensional iteration spaces for nonshared memory machines*, In Supercomputing, November 1991.

121. J.R. Rice, *Computational Science and the Future of Computing Research*, IEEE Computational Science and Engineering, Vol. 2, No. 4, pp. 35-41, 1995.

122. D. A. Reed, L. M. Adams, and L. Patrick, *Stencils and problem partitionings: Their influence on the performance of multiple processor systems*, IEEE Transaction on Computer, 36(7):845-858, July 1987.

123. J. Roberts, P. Boyer, and M. Servel. A real time sorter with application to ATM traffic control. In *Proceedings of ISS'95*, number Pb.1 in volume 1, Berlin, Germany, April 1995.

124. T. Risset and Y. Robert. Synthesis of processors arrays for the algebraic path problem: Unifying old results and deriving new architectures. *Parallel Processing Letters*, 1:19–28, 1991.

125. Y. Robert and M. Tchuenté. Résolution systolique de systèmes linéaires denses. *RAIRO Modélisation et Analyse Numérique, Technique et Sciences Informatiques*, 19(2):315–326, 1985.

126. Y. Robert and D. Trystram. Systolic solution of the algebraic path problem. Parallel Computing, vol. 39, 1987.

127. J.L. Roch, *Complexité parallèle et algorithmique PRAM*, Dans Algorithmes parallèles-Analyse et conception, Chap.5, pp.105-128, 1994.

128. Günter Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219, 1985.

129. J. de Rumeur, *Communications dans les réseaux de processeurs*, Masson, 1994.

130. R.M. Russell, *The CRAY-1 computer system*, Communication of the ACM, 21(1):63-72, 1978.

131. Y. Saad, *Data communication in parallel architectures*, Parallel Computing, (11), 1989.

132. I. Sakho, and M. Tchuente, *Une méthodologie de conception d'algorithmes systoliques pour réseaux réguliers.*, Technique et Science Informatique, 1989.

133.V. Sarkar, *Partitionning and Scheduling Parallel Programs for Execution on Multi-Processors*, MIT Press, 1989.

134.Chris J. Schieman and Peter R. Cappello. A processor-time minimal systolic array for transitive closure. In *International Conference on Application Specific Array Processors*, pages 19–30, Princeton, NJ, September 1990. IEEE Computer Society, IEEE Computer Society Press.

135.A. Schrijver, *Theory of Linear and Integer Programming*, Wiley-Interscience series in Discrete Mathematics, John Wiley and Sons, 1986.

136.C.L. Seitz, *The Cosmic Cube*, Communication of the ACM, 28, no 1, pp. 22-33, January 1985.

137.M. Servel, P. Gonet, and J. François. Circuit pour mémoriser des états de disponibilité de ressources logiques, telles que cellules de mémoire, et établir des adresses de ressources libres. Brevet national INPI no 87 09068, 1987.

138.B. Shiver and B. Smith, *The Anatomy of a High Performance Microprocessors - A systems Perspective*, IEEE Computer Society, July 1998.

139.P. W. Shor, *Quantum Computing*, Proceeding of the ICM Conference, 1998.

140.M. Snir, S. Otto, S.H. Lederman, D. Walker, and J. Dongarra, *MPI - The Complete Reference*, The MIT Press, 1995.

141.V.D. Steen, *Overview of recent supercomputers*, NCF Report, The Hague, January 1995

142.C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices*, Journal of Parallel and Distributed Computing and Practices, To appear, 1999.

143.C. Tadonki, *Equations récurrentes et algorithmes parallèles pour la multiplication d'un vecteur par un produit tensoriel de matrices*, Renpar'11, Rennes, 8-11 Juin 1999.

144.C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part II)*, Parallel and Distributed Computing and Practices, To appear, 2000.

145.C. Tadonki, *Ordonnanments canoniques*, Renpar2000, Besancon, France, pp. 13-18, 2000 (A paraitre dans *Calculateurs Parallèles*).

146.C. Tadonki, *Synthèse d'Ordonnancements Parallèles par Reproduction Canonique*, Calculateurs Parallèles (A paraitre), 2000.

147.C. Tadonki, *Méthodologie de Conception d'Algorithmes Efficaces pour le Produit Tensoriel*, CARI2000, Madagascar, 16-19 Octobre 2000.

148.T. Takaoka and K. Umehara. An efficient VLSI circuit for the all pairs shortest path problem. *Journal of Parallel and Distributed Computing*, 16:265–270, 1992.

149.C. Tayou Djamegni, P. Quinton, S. Rajopadhye, and T. Risset. Derivation of systolic algorithms for the algebraic path problem by recurrence transformations. *Parallel Computing*, To appear, 1999.

150.M. TCHUENTE, *Parallel Computation on Regular Arrays*, Manchester University Press and Jhon Wiley and Sons, 1990.

151. C.Tong and P. N. Swarztrauber, *Ordered Fast Fourier Transforms on Massively Parralel Hypercube Multiprocessor*, Journal of Parallel and Distributed Computing 12, 50-59, 1991.

152. A.H. Toub (ed.), *John Von Neumann - Collected Works*, Volume 5, Pergamon Press, New York, USA, 1961.

153. D. Trystram, *Communication dans les grilles de processeurs*, In Algorithmique Parallèle, pp.159-169, Masson 1992.

154. A. M. Turing, *On computable numbers with an application to the entscheidungsproblem*, Proc. London Mathematical Soc. Ser 2,42:230-265, 1936.

155. J. Ullman, *NP-complete scheduling problems*, Journal of Computer Sciences, 10, 384-393, 1975.

156. H. Le Verge, C. Mauras, and P. Quinton, *The ALPHA language and its use for the design of systolic arrays*, Journal of VLSI Signal Processing, 3:173-182, 1991.

157. P. Van Emde Boas, *Machine models and simulation*, Chapter 1, pp. 1-66, Handbook of Theoretical Computer Science, J. Van Leeuvwen (ed.), MIT Press, 1990.

158. C. Van Loan, *Computational Framework for the Fast Fourier Transform*, SIAM, 1992.

159. M. J. Wolfe, *Iteration space tiling for memory hierarchies*, In Parallel Processing for Scientific Computing, pages 357-361, 1987.

160. M. J. Wolfe, *More iteration space tiling*, In supercomputing, pages 655-664, 1989.

161. E. Wallmeier and T. Worster. A cell spacing and policing device for multiple virtual connections on one ATM pipe. In *RACE Workshop on Network Planning and Evolution*, London, April 1991.

162. E. Wallmeier and T. Worster. The spacing policer, an algorithm for efficient peak bit rate control in atm networks. In *Proceedings of ISS'92*, number A5.5 in volume 2, pages 22–26, Yokohama, Japan, October 1992.

163. D.B.Wilson, *Embedding Leveled Hypercube Algorithms into Hypercubes*, SPAA'92, 4th Annual Symposium on Paralle Algorithms and Architectures, pp. 264-270, 1992.

164. F. Wong and M.R. Ito. Parallel sorting on a Re-circular Systolic Sorter. *The Computer Journal*, 27(3):260–269, 1984.

165. H.P. Yap, *Some topics in graph theory*, London Math. Soc., Cambridge University Press, 1986.

166. H. Yasuura. The parallel enumeration sorting scheme for VLSI. *IEEE Transaction on computers*, C-31(12):1192–1201, December 1982.

167. H. Zima and B. Chapman, *Supercompilers for Parallel Vector Computers*, ACM Press, New York, USA, 1991.

Table des matières

Table des figures

Résumé

Dans ce travail, nous avons étudié les techniques de conception d'algorithmes parallèles, et proposé des schémas efficaces pour quelques problèmes particuliers. S'agissant des techniques de parallélisation, nous avons défini et illustré une méthodologie originale d'ordonnancement parallèle basée sur l'usage des isomorphismes de graphes. La technique est assez explicite et s'applique à une classe moins restrictive de problèmes. Les ordonnancements obtenus, que nous apellons *ordonnancements canoniques*, permettent un partitionnement naturel dans le cas où le nombre de processeurs est réduit. Globalement, les solutions dérivées sont assez régulières, et leur efficacité dépend d'un choix judicieux des paramètres de la méthode. S'agissant de la parallélisation d'algorithmes, nous avons étudié les problèmes du *produit tensoriel*, du *chemin algébrique*, et des *files de priorité systoliques*. Nos solutions ont pour la plupart été exérimentées sur les machines INTEL PARAGON, NEC CENJU3, et CRAY T3E.

Mots clés : Algorithmes parallèles, complexité, combinatoire, graphe, équation récurrente, produit de Kronecker, chemin algébrique, machine parallèle, architecture systolique.

Abstract
In this work, we have studied some parallelization methods, and we have proposed parallel algorithms for a number of problems. Concerning the parallelization technics, we have defined and illustrated a new method of designing parallel schedules, based on on graph isomorphisms. The method is quite explicit, and it can be applied on a less restricted class of problems, since it does not depend on the nature of dependancies. Moreover, the solutions derived, named *canonical schedules*, can be trivially adapted to run on a fewer number of processors. Globally, canonical schedules are quite regular, and their efficiency depend on a good choice of starting parameters. C oncerning the parallelization of algorithms, we have studied the problem of the *Kronecker product*, the *algebraic path problem*, and a *systolic priority queue*. Our parallel algorithms have been experimented on standard parallel machine such as INTEL PARAGON, NEC CENJU3, and CRAY T3E.

Keywords: Parallel algorithm, complexity, combinatoric, graph, recurrence equations, Kronecker product, algebraic path, parallel machine, systolic architecture.