INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

THESE

pour obtenir le grade de

DOCTEUR DE L'Institut Polytechnique de Grenoble

***Spécialité :* "Mathématiques et Informatique"**

préparée au Laboratoire d'Informatique de Grenoble dans le cadre de
l'Ecole Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"

présentée et soutenue publiquement

par

Daouda Traoré

le 19 Décembre 2008

Algorithmes parallèles auto-adaptatifs et applications

Directeur de thèse : Denis Trystram
Co-Directeur de thèse : Jean-Louis Roch

JURY

ZOLTAN SZIGETI	Grenoble INP	Président
OLIVIER BEAUMONT	INRIA Bordeaux-Sud-Ouest	Rapporteur
CHRISTOPHE CÉRIN	Université Paris 13	Rapporteur
JEAN-YVES L'EXCELLENT	INRIA Rhône-Alpes	Examinateur
DENIS TRYSTRAM	Grenoble INP	Directeur de thèse
JEAN-LOUIS ROCH	Grenoble INP	Co-Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier les membres de mon jury qui ont accepté de s'intéresser à mon travail malgré leurs emplois du temps surchargés. Merci à Zoltan Szigeti pour avoir accepté de présider le jury de ma soutenance. Merci à Olivier Beaumont et Christophe Cérin pour avoir pris le temps de lire mon manuscrit et d'avoir grandement contribué à son amélioration par leurs remarques et commentaires. Ayant été plusieurs fois en contact avec Christophe Cérin durant ces trois années, je voudrais, tout particulièrement le remercier pour son enthousiasme et sa grande sympathie. Merci à Jean-Yves L'Excellent pour avoir accepté d'examiner ce travail. Merci à Denis Trystram de m'avoir fait confiance pour être mon directeur de thèse, je le remercie aussi pour son accueil et ses qualités humaines. Un très grand merci à mon encadrant et co-directeur de thèse Jean-Louis Roch qui m'a encadré depuis mon Master 2 Recherche jusqu'à la fin de cette thèse. Je ne saurai jamais lui exprimer toute ma gratitude pour ses qualités humaines, son aide, son soutien, sa grande disponibilité durant ces trois années. Sans lui cette thèse n'aurai pas abouti. Merci beaucoup à toi Jean-Louis.

Je remercie tous ceux qui m'ont aidé à faire des corrections orthographiques de ce manuscrit. Merci donc à Marc Tchiboukdjian, Gérald Vaisman, Jérôme Vienne, Moussa Tembely, Fatouma Goundo Camara.

Je tiens à remercier mes soeurs maliennes qui m'ont aidé à préparer le pot de ma soutenance de thèse, qui a été un grand succès. Je remercie donc Mariam Dibo, Fatoumata Goundo Camara, Mariam Haidara ma chérie.

Je remercie tous les membres des deux équipes du laboratoire LIG et INRIA (MOAIS et MESCAL), sans citation de leurs noms pour ne pas en oublier, pour leur gentillesse et leur accueil.

Je tiens à remercier Serge Guelton pour ses aides techniques et sa gentillesse. Je tiens à remercier mes co-bureaux Ihab Sbeity et Afonso Sales pour leurs aides, leurs conseils et leur accueil. Je tiens à remercier tous ceux qui m'ont aidé, soutenu et encouragé pendant cette thèse.

Enfin, je remercie ma famille pour m'avoir soutenu et encouragé durant les huit années d'études en France, depuis ma première année à l'université jusqu'aux études doctorales.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Introduction

Le parallélisme est utilisé depuis longtemps en informatique pour résoudre des problèmes scientifiques (*e.g.* simulation, météorologie, biologie, jeux vidéo) le plus rapidement possible. Le principe de base du parallélisme est d'utiliser plusieurs ressources (*e.g.* processeurs) qui fonctionnent concurremment pour accroître la puissance de calcul pour la résolution d'un même algorithme pour un problème donné. L'objectif du parallélisme est non seulement de résoudre les problèmes le plus rapidement, mais aussi de pouvoir résoudre des problèmes de plus grande taille.

Pour pouvoir mettre en œuvre le parallélisme en pratique, des systèmes ou architectures ont été développés et ne cessent d'évoluer. Ces avancées dans le domaine des architectures parallèles au cours de ces dernières années, ont permis le développement considérable de nouvelles architectures parallèles et distribuées. Ainsi, les architectures de processeurs multi-cœurs, les processeurs graphiques (*GPUs*), les machines SMP (*Symetric MultiProcessors*), les grappes ou les grilles de calcul sont apparues. Ce qui fait qu'aujourd'hui l'utilisation du parallélisme devient de plus en plus omniprésente dans tous les systèmes. Les ordinateurs standards (fixes et portables) d'aujourd'hui deviennent de plus en plus des machines constituées de plusieurs unités de calcul (cœurs) qui sont capables de traiter simultanément des données. Les systèmes parallèles de grande taille d'aujourd'hui comme les grilles de calcul sont capables d'exécuter rapidement des calculs scientifiques demandant plusieurs mois de calcul sur un seul processeur. Cependant plusieurs éléments doivent être pris en compte dans le développement d'algorithmes parallèles efficaces pour ces nouveaux systèmes. Car dans ces nouveaux systèmes :

- Le nombre de processeurs peut varier d'une machine à l'autre.
- Les processeurs peuvent être hétérogènes ; de fréquences différentes (classique dans les grilles de calcul comme grid5000) ou même de fonctionnement différent (*GPU*).
- Dans les systèmes multi-processus (*e.g.* processeurs multi-cœurs, machines SMP), plusieurs applications peuvent être concurrentes sur une même unité de traitement. Donc la vitesse par rapport à une application sur une ressource dépend de la charge extérieure de

la ressource.

Pour bénéficier pleinement de la puissance de ces nouvelles architectures, les programmes parallèles doivent les exploiter efficacement. La construction d'un algorithme parallèle efficace pour un problème donné admettant une solution séquentielle demande de prendre en considération plusieurs éléments qui sont par exemple : la taille du problème à résoudre, les paramètres de configuration matérielle de l'architecture d'exécution (*e.g.* nombre de processeurs). Dans le passé, les algorithmes parallèles étaient construits spécialement pour des systèmes dédiés. Actuellement, les systèmes parallèles sont composés, de machines hétérogènes, de machines à charges variables, de machines dynamiques. Il est difficile pour un utilisateur de choisir un algorithme adéquat pour un système donné, vu l'évolution continuelle des systèmes. En effet, un algorithme parallèle optimal pour une architecture dédiée, n'est pas performant quand toutes les ressources de celle-ci ne sont pas disponibles. L'exploitation efficace des ces systèmes, nécessite alors le développement d'algorithmes capables de s'adapter automatiquement au contexte d'exécution.

Pour réaliser la construction d'un algorithme indépendant de l'architecture, la manière classique consiste à découper le programme en plusieurs tâches (une suite d'instructions qui sera exécutée séquentiellement), qui seront ensuite placées dynamiquement sur les ressources disponibles de l'architecture cible. Mais une telle construction a des inconvénients. En effet, si le nombre de tâches créées est petit, cela suppose que la taille (nombre d'instructions à exécuter) d'une tache est grande, ce qui ne convient pas dans le contexte où le nombre de ressources et leurs puissances peuvent être variables à tout moment de l'exécution. Si le nombre de tâches créées est supérieur au nombre de ressources disponibles, le placement et l'entrelacement de ces nombreuses tâches entraînent un surcoût qui peut être : arithmétique, copie de données, synchronisation entre tâches. Aussi, si le nombre de tâches créées est très élevé, les surcoûts de créations de tâches peuvent beaucoup influencer le temps d'exécution globale de l'application.

Pour limiter le surcoût de créations de tâches, des techniques [35, 76, 71] ont été mises en œuvre dans l'ordonnancement par vol de travail [3, 18]. Ces techniques diminuent le surcoût d'ordonnancement de l'algorithme parallèle, mais ne s'attaquent pas au surcoût arithmétique lié à la parallélisation, qui peut s'avérer très couteux en comparaison avec un algorithme séquentiel.

Dans cette thèse, nous étendons un schéma algorithmique générique original proposé par Daoudi et al [26]. Ce schéma est basé sur le couplage de deux algorithmes distincts, l'un séquentiel optimal minimisant le nombre d'opérations et l'autre parallèle minimisant le temps d'exécution. La génération du parallélisme n'est effectuée qu'en cas d'inactivité d'un processeur.

1.2 Objectifs et contributions

L'objectif de ce travail est de spécifier et de valider un schéma générique qui permet de développer des programmes parallèles adaptatifs ayant des garanties d'efficacité sur une grande

classe d'architectures parallèles, en particulier dans le cas pratique usuel où les processeurs sont de vitesse variable par rapport à l'application, du fait de leur utilisation par d'autres processus (système ou autres applications). Ce schéma est basé sur un couplage, récursif et dynamique, entre plusieurs algorithmes résolvant un même problème. Les contributions apportées par ce travail de recherche sont les suivantes :

- Nous proposons un nouvel algorithme statique optimal pour le calcul parallèle des préfixes sur des processeurs identiques, l'optimalité de cet algorithme a été prouvée, une analyse expérimentale montre que cet algorithme se comporte mieux en pratique sur des machines multicœurs que l'algorithme optimal statique de Nicolau et Wang [94].
- Nous montrons une borne inférieure du calcul des préfixes sur des processeurs à vitesses variables. Puis nous proposons un algorithme adaptatif du calcul parallèle des préfixes pour des processeurs dont la vitesse ou le nombre peut varier en cours d'exécution. Des analyses théoriques et expérimentales montrent les bonnes performances de cet algorithme.
- Nous proposons un algorithme adaptatif indépendant du nombre de processeurs de la fusion parallèle de deux séquences triées et son analyse théorique. Cet algorithme a permis la parallélisation adaptative de l'algorithme de tri stable. Des garanties théoriques et expérimentales de performances sont obtenues par rapport au temps séquentiel. Nous proposons aussi un algorithme adaptatif de la partition basé sur le schéma générique. Cet algorithme a permis la parallélisation adaptative de l'algorithme de tri introspectif [64]. Des garanties théoriques et expérimentales de performances sont obtenues par rapport au temps séquentiel.
- Nous proposons un schéma algorithmique parallèle générique avec une analyse théorique montrant ses garanties d'efficacité. Puis nous donnons une interface générique basée sur ce schéma qui permet le développement d'algorithmes parallèles adaptatifs. Elle a été développée en C++ et utilise le moteur de vol de travail de l'environnement de programmation parallèle Athapascan/Kaapi [39].
- Une parallélisation adaptative de plusieurs algorithmes avec des conteneurs à accès aléatoires de la librairie standard C++ a pu être réalisée en utilisant notre interface générique. A notre connaissance, nous sommes les premiers à fournir des algorithmes adaptatifs optimaux de : `partial_sum` `unique_copy` et `remove_copy_if`. Les expérimentations menées montrent le bon comportement des algorithmes de la STL que nous avons adaptés par rapport aux librairies MCSTL (Multi-Core Standard Template Library qui est actuellement utilisée par Gnu libstdc++ parallel mode) et TBB (Threading Building Blocks).

Cette thèse a conduit à six publications dans des conférences internationales et nationales. Ces publications sont :

[1] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. **Deque-free work-optimal parallel STL algorithms**. In LNCS 5168 Springer-Verlag, editor, EUROPAR'2008, pages 887-897, Canary Island, Spain, August 26-29^{th}.

[2] Jean-Louis Roch, Daouda Traoré, and Julien Bernard. **On-line adaptive parallel prefix computation**. In LNCS 4128 Springer-Verlag, editor, EUROPAR'2006, pages 843-850, Ger-

many, August 29^{th}-September 1^{st}.
[3] Julien Bernard, Jean-Louis Roch, and Daouda Traoré. **Processor-oblivious parallel stream computation**. In 16^{th} Euromicro International Conference on Parallel, Distributed and network-based processing (PDP'2008), pages 72-76, Toulouse, France, Feb 2007, IEEE Computer Society Press.
[4] Jean-Louis Roch, Daouda Traoré. **Un algorithme adaptatif optimal pour le calcul parallèle des préfixes**. In $8^{ème}$ Colloque Africain sur la Recherche en Informatique (CARI'2006), pages 67-74, Cotonou, Benin, Novembre 2006.
[5] Daouda Traoré, Jean-Louis Roch, Christophe Cérin. **Algorithmes adaptatifs de tri parallèle**. In $18^{ème}$ rencontre francophone sur le parallélisme (RenPar'18), Fribourg, Suisse, Feb. 2008.
[6] Daouda Traoré. **Algorithmes parallèles adaptatifs et applications**. In $5^{ème}$ Symposium malien des sciences appliquées (MSAS'2008), Bamako, Mali, 3-8 août 2008.

1.3 Organisation du manuscrit

L'organisation de ce document est la suivante :

- Le chapitre 2 présente quelques notions sur la programmation parallèle, à savoir les architectures parallèles, les modèles de programmation parallèle et les bibliothèques permettant le développement d'applications parallèles. Aussi, nous y présenterons les modèles de coûts qui seront utilisés tout au long de ce document.
- Le chapitre 3 étudie la notion d'algorithmes adaptatifs et leur intérêt. Pour illustrer les besoins des algorithmes adaptatifs, nous avons pris comme exemple le calcul parallèle des préfixes qui permet de mettre en évidence les problèmes qu'on peut rencontrer dans une parallélisation classique. Pour mieux mettre en évidence ces problèmes, nous avons d'abord proposé une borne inférieure du temps d'exécution du calcul des préfixes sur p processeurs à vitesses variables. Ensuite, nous avons rappelé un algorithme statique du calcul des préfixes sur des processeurs identiques, puis nous avons proposé un nouvel algorithme statique optimal pour le calcul des préfixes, et enfin nous avons rappelé un algorithme récursif du calcul des préfixes. Nous avons discuté sur les avantages et inconvénients de chaque algorithme. Pour terminer le chapitre, nous avons rappelé quelques approches d'adaptabilités existantes.
- Dans le chapitre 4, nous proposons un nouvel algorithme parallèle du calcul des préfixes qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles. Nous montrons théoriquement que son temps d'exécution est asymptotiquement optimal. Les résultats théoriques sont validés par des expérimentations menées sur deux machines SMP (*Symetric Multi-Processors*) de 8 et 16 cœurs.
- Le chapitre 5 présente deux algorithmes adaptatifs de tri. Le premier repose sur un algorithme parallèle adaptatif de fusion et le deuxième sur un algorithme adaptatif de la partition. En fait, le premier est la parallélisation adaptative de l'algorithme de tri par fusion et le deuxième est la parallélisation du tri introspectif [64] (le tri rapide amélioré).
- Le chapitre 6 présente le schéma générique adaptatif que nous proposons avec une analyse théorique montrant ces garanties de performance. L'interface générique permettant

l'écriture des algorithmes adaptatifs sera aussi présentée.
– Le chapitre 7 présente l'application du schéma générique pour la parallélisation de plusieurs algorithmes de la librairie standard C++ (*Standard Template Library*). Des comparaisons expérimentales par rapport à d'autres bibliothèques parallèles montrent les bons comportements des algorithmes parallélisés de manière adaptative avec notre schéma générique.
– Enfin le chapitre 8 conclut cette thèse avec les perspectives.

Chapitre 2

Programmation parallèle

L'objectif de ce chapitre est de présenter quelques notions sur la programmation parallèle. Nous y présentons, dans les cinq premières sections (section 2.1 à section 2.5), quelques types d'architectures de machines parallèles, les modèles de programmation et de coûts des programmes parallèles, la manière de concevoir et de modéliser un programme parallèle. Enfin dans la dernière section (section 2.6), nous terminons par la présentation de quelques bibliothèques de programmation parallèle.

2.1 Architectures parallèles

Pour rendre possible l'exécution des programmes parallèles, des architectures spécifiques de machines ont été développées. Il existe plusieurs types d'architectures parallèles caractérisées, principalement, par différents modèles d'interconnexions entre les processeurs et la mémoire. En 1972, Flynn [34] propose une classification des machines parallèles basée sur la notion de flot de données et de flot d'instructions. Dans cette classification, Flynn distingue quatre types principaux de machines parallèles :

- Les machines SISD (*Single Instruction Single Data*) : Dans ces machines, une seule instruction est exécutée et une seule donnée est traitée à tout instant. Les traitements dans ce type de machines sont donc séquentiels (sans parallélisme). Ce modèle correspond à une machine monoprocesseur.
- Les machines SIMD (*Single Instruction Multiple Data*) : Dans ces machines, tous les processeurs sont identiques et sont contrôlés par une unique unité de contrôle centralisée. A chaque étape, tous les processeurs exécutent la même instruction de façon synchrone mais sur des données différentes. Ce type de machines est utilisé pour les calculs spécialisés (*e.g.* calculs vectoriels). Ces machines correspondent par exemple à des processeurs graphiques (*graphics processing unit*, GPU), les unités de calcul en virgule flottante (*Floating Point Unit*, FPU). Dans l'exemple 2.1.1, l'opération d'addition ($+$) est exécutée simultanément par tous les processeurs.

Exemple 2.1.1. *Une boucle simple effectuant l'addition de deux vecteurs dans un troisième (tous disjoints en mémoire).*
$forall(i = 0; i < 10; i++)$
$a[i] = b[i] + c[i]$
Dans cet exemple tous les $a[i]$ ($0 \leq i < 10$) peuvent être calculés indépendamment.

- Les machines MISD (*Multiple Instructions Single Data*) : Ces machines peuvent exécuter plusieurs instructions sur la même donnée. Ce modèle englobe certaines architectures de type pipeline et les architectures tolérant les pannes par réplication de calcul.
- Les machines MIMD (*Multiple Instructions Multiple Data*) : Dans ce modèle, chaque processeur est autonome, dispose de sa propre unité de contrôle et exécute son propre flot d'instructions sur son propre flot de données. Ces machines sont les plus courantes aujourd'hui : on retrouve les machines à mémoire partagée (SMPs, cc-NUMAs) et les machines à mémoire distribuée sur lesquelles nous reviendrons.

De ces quatre modèles d'architectures, les modèles MIMD et SIMD sont prépondérants et correspondent aux deux principales techniques de parallélisation.

Depuis l'apparition des processeurs vectoriels (Machines SIMD) dans les années 1980, les architectures parallèles sont de plus en plus présentes. Ainsi des architectures de petite et de grande taille sont apparues dont nous donnons ci-dessous une liste non exhaustive de ces architectures :

- **Systèmes multiprocesseurs (*MPSoC*) et Processeurs multicœurs**. Un processeur multicœur est composé d'au moins deux unités de calcul (cœurs) gravées au sein d'une même puce. Les cœurs des processeurs dans la plupart des cas sont homogènes (identiques). Mais *IBM*, *Sony* et *Toshiba* ont exploité le cas des cœurs hétérogènes (différents) et spécialisés dans des domaines bien précis (audio, affichage, calcul pur). Depuis 2005, les processeurs multicœurs sont sur le marché.
- **SMPs** (*Symetric MultiProcessors*). Cette architecture est composée de plusieurs processeurs identiques qui partagent la même mémoire (les processeurs lisent et écrivent dans la même mémoire, mais ont généralement chacun leur propre cache). Le temps d'accès mémoire est identique entre processeurs. Cependant, l'accès à la mémoire constitue un goulot d'étranglement sur ce type d'architecture dès que le nombre de processeurs devient important. Ce nombre se limite à une trentaine aujourd'hui.
- **cc-NUMAs** (*Cache Coherence Non-Uniform Memory Access*). Sur cette architecture, chaque processeur dispose de sa propre mémoire vive. Les processeurs communiquent entre eux par un système de communication qui les relie. Contrairement aux SMPs, le temps d'accès mémoire d'un processeur à un autre n'est pas uniforme.
- **Grappes** (*Cluster*). Une grappe de calcul est composée par un ensemble homogène de nœuds interconnectés entre eux par un réseau local rapide. Les communications entre les nœuds se font par échange de messages.
- **Grilles** (*Grid*). Une grille est un ensemble de grappes interconnectés entre elles par des réseaux de très haut débit tels que des WAN. C'est une architecture très hétérogène puisque les matériels et les systèmes qui la constituent peuvent être très disparates. En France, il existe un projet fédérateur d'une grille pour la recherche, appelé Grid 5000 (www.grid5000.fr).

2.2 Modèles de programmation parallèle

Un modèle de programmation parallèle permet d'exprimer la façon dont une application parallèle sera programmée. Plusieurs modèles existent pour cette programmation, mais ils sont pour la plupart liés à l'architecture sous-jacente. Un des premiers modèles est le modèle SPMD (*Single Program, Mutiple Data*) : l'application parallèle est décrite par un seul et même programme séquentiel : ce programme est exécuté par un nombre fini de machines, chaque machine l'exécute avec des données en entrée spécifiques (comme par exemple l'identification de la machine qui exécute le programme). Ce modèle SPMD convient par exemple pour les architectures MIMD. Les modèles basés sur les architectures à mémoire partagée et distribuée sont souvent les plus couramment utilisés pour la programmation d'applications parallèles. Ils font souvent appel au modèle SPMD. Dans les sections 2.2.1 et 2.2.2, nous présentons ces deux modèles de mémoire, partagée et distribuée.

2.2.1 Parallélisme à mémoire partagée

Dans ce modèle, l'ensemble des processeurs de la machine parallèle partagent le même espace mémoire dans lequel ils peuvent lire et écrire de manière indépendante et asynchrone. Les coûts des accès mémoire peuvent être identiques pour chacun des processeurs (on parle de machine *UMA : Uniform Memory Access*) ou dépendant du processeur et des adresses accédées (on parle de machine *NUMA*). Les exécutions simultanées sont effectuées par des fils d'exécutions (*thread* en anglais). Les fils d'exécutions partagent le même espace mémoire et les échanges de données entre eux se font simplement par des lecteurs/écritures en mémoire. Il faut donc garantir que deux fils d'exécutions ne modifient pas au même moment la valeur d'une donnée dans une même zone de la mémoire partagée. Ceci peut se faire en utilisant des outils de synchronisation comme des verrous, des sémaphores ou des barrières de synchronisation. Aussi les données peuvent être de différentes versions dans les caches des différents processeurs à un instant donné ; ce qui peut conduire à des résultats incohérents. Pour garantir cette cohérence, des modèles de cohérence doivent être utilisés.

Le principal désavantage de ce modèle est que les sections critiques du code doivent être bien analysées par le programmeur afin d'éviter une diminution significative de la performance de la machine parallèle. L'avantage est la simplicité de programmation.

2.2.2 Parallélisme à mémoire distribuée

Dans ce modèle, chaque processeur de la machine parallèle possède sa propre mémoire locale. Les mémoires des processeurs sont physiquement distantes. Les communications des données entre les processeurs se font par échange de messages et aussi les synchronisations entre les processeurs. Les communications peuvent être synchrones ; lors de l'envoi d'un message par un émetteur, le récepteur les reçoit et sa réponse doit être explicite. Les communications peuvent

être bi-points ou collectives, c'est à dire faire participer tous les processeurs aux échanges. Les communications collectives peuvent être de type diffusion (*broadcast*)-un processeur envoie la même donnée à tous les processeurs- ; dispersion/regroupement (*scatter/gather*) ; tous vers tous (*all-to-all*) ; réduction(*reduction*).

Le principal désavantage de ce modèle est la lourdeur de la programmation, un processeur ne peut accéder qu'à des données dans sa propre mémoire. Les mémoires virtuelles partagées (*MVP*) permettent d'offrir une vision d'un espace global (*NUMA*) à partir d'une mémoire de pagination.

2.3 Modèles de coûts des programmes parallèles

Le coût ou temps d'exécution d'un algorithme séquentiel est généralement exprimé en fonction de la taille des données en entrée. Le temps d'exécution d'un programme parallèle ne dépend pas seulement de la taille des données en entrée, mais aussi du nombre de ressources utilisées pour le traitement des données, de leur vitesse relative et du temps de communication inter-processeur. L'évaluation du coût d'un programme parallèle doit permettre de mettre en évidence l'intérêt d'utiliser le parallélisme, par exemple le programme s'exécute t-il beaucoup plus rapidement que le programme séquentiel correspondant ? Donc, pour pouvoir mener à bien cette évaluation, il est fondamental de définir un modèle d'évaluation de coût des programmes parallèles. Ce modèle doit permettre de mener des analyses de complexité (montrer qu'un algorithme donné est optimal, établir la complexité minimale d'un problème, ou montrer qu'un algorithme parallèle s'exécute plus rapidement qu'un algorithme séquentiel sérialisé), de donner une classification des algorithmes parallèles d'une manière analogue à l'algorithmique séquentielle [73].

Le modèle théorique PRAM (*Parallel Random Acces Machine*) [49, 73, 55, 69, 74] est largement utilisé pour analyser la complexité des algorithmes parallèles. Il est composé d'un ensemble de processeurs (en nombre infini) fonctionnant de manière synchrone et d'une mémoire partagée, à laquelle les processeurs accèdent. Chaque processeur exécute une opération à chaque étape du calcul (durée unitaire) et il est supposé que le temps d'accès à la mémoire prend un temps unitaire égal à zéro. Ce modèle s'adapte bien aux architectures parallèles à mémoire partagée. Comme il permet aux processeurs d'accéder à n'importe quelle case de la mémoire en une unité de temps, et que sur une machine réelle l'accès simultané par plusieurs processeurs peut conduire à des résultats incohérents, des règles ont été définies pour spécifier le comportement en cas d'accès concurrent à la même case mémoire. Dans le modèle EREW (*Exclusive Read Exclusive Write*, lecture et écriture exclusive), deux processeurs différents ne peuvent ni lire ni écrire à une même étape sur une même case mémoire. Dans le modèle CREW (*Concurrent Read Exclusive Write*, lecture concurrente et écriture exclusive), des processeurs différents peuvent lire à une même étape le contenu d'une même case mémoire, mais ne peuvent pas écrire simultanément dans une même case ; c'est le modèle le plus usuel.

Dans toute la suite de ce document les notations suivantes seront utilisées :

- W_{seq} représente le nombre total d'opérations élémentaires unité du meilleur algorithme séquentiel. On parle aussi de **travail** de l'algorithme séquentiel. Il représente aussi le temps d'exécution de l'algorithme séquentiel car, à une constante multiplicative près, le temps est confondu avec le nombre d'opérations.
- T_{seq} est le temps d'exécution (sans instruction parallèle) du meilleur algorithme séquentiel. Sur un processeur de vitesse Π_{seq}, on a $T_{seq} = \frac{W_{seq}}{\Pi_{seq}}$.
- W est le nombre d'opérations de l'algorithme parallèle. On parle aussi de **travail** de l'algorithme parallèle.
- T_p est le temps d'exécution de l'algorithme parallèle sur p processeurs.
- D est le nombre maximal d'opérations élémentaires unité en précédence lors de l'exécution de l'algorithme sur une infinité de processeurs. C'est aussi la **profondeur** du graphe représentant le programme parallèle. La profondeur d'un graphe est le plus long chemin du graphe (chemin critique) en nombre d'opérations élémentaires unité ; un chemin dans le graphe représente en effet une suite de tâches obligatoirement exécutées séquentiellement.
- Pour modéliser la vitesse (nombre d'opérations élémentaires effectuées par unité de temps), nous utilisons le modèle proposé par Bender et Rabin [8]. A tout instant t, la vitesse $\Pi_i(t)$ d'un processeur i, $1 \leq i \leq p$, dépend du nombre de processus (correspondant à d'autres applications) ordonnancés sur le processeur physique auquel il est affecté. Pour une exécution parallèle de durée T (en temps écoulé) sur p exécuteurs, on définit la vitesse moyenne $\Pi_{ave} = \frac{\sum_{t=1}^{T}\sum_{i=1}^{p}\Pi_i(t)}{p.T}$ et la vitesse maximale de tous les processeurs par $\Pi_{max} = max_{i=0,\ldots,p-1;t=1,\ldots,T}\Pi_i(t)$.
- L'**accélération** (*speedup*) d'un programme parallèle est le rapport entre le temps du meilleur algorithme séquentiel et le temps de l'algorithme parallèle résolvant le même problème sur une machine parallèle donnée, elle est désignée par $S_p = \frac{T_{seq}}{T_p}$, si $S_p = p$ l'accélération est dite **linéaire**. L'**efficacité** traduit le taux d'utilisation des p processeurs de la machine parallèle, elle est désignée par $E_p = \frac{S_p}{p} = \frac{T_{seq}}{pT_p}$.
  En pratique, une accélération supérieure à p peut être obtenue (c'est ce qu'on appelle **accélération super-linéaire** : $S_p > p$), ce qui est possible pour des algorithmes non déterministes et de recherche, ou phénomène dû à l'exploitation efficace d'une mémoire cache de taille p fois supérieure avec p processeurs qu'avec un seul.

2.4 Conception d'un programme parallèle

La construction d'un algorithme parallèle pour un problème donné est beaucoup plus complexe que la construction d'un algorithme en séquentiel du même problème car elle demande la prise en compte de plusieurs facteurs qui sont par exemple : la partie du programme qui peut être traitée en parallèle, la manière de distribuer les données, les dépendances des données, la répartition de charges entre les processeurs, les synchronisations entre les processeurs. Il y a essentiellement deux méthodes pour construire un algorithme parallèle, l'une consiste à détecter et à exploiter le parallélisme inhérent dans un algorithme séquentiel déjà existant, l'autre

consistant à inventer un nouvel algorithme dédié au problème donné [69]. Dans cette section, nous allons présenter deux techniques très courantes utilisées pour construire un algorithme parallèle.

2.4.1 Les techniques de découpe

2.4.1.1 Partitionnement

Partitionnement simple

Le partitionnement simple consiste à découper le problème en des parties indépendantes, chaque partie pourra être exécutée par un processeur différent. Dans la majorité des problèmes celui-ci est fait en fonction du nombre de processeurs utilisés. Il est utilisé lorsque le traitement d'une donnée est complètement indépendant des autres.

Partitionnement avec communication

Dans la plupart des problèmes, les processeurs ont besoin de communiquer leurs résultats entre eux. Dans le partitionnement avec communication, la résolution du problème se fait en deux phases. Dans la première phase, le problème est découpé en plusieurs parties où chaque partie pourra être exécutée par un processeur différent. Dans la deuxième phase, les processeurs communiquent entre eux pour fusionner leurs résultats afin de terminer la résolution du problème. Comme dans le partitionnement simple, dans la plus grande majorité des problèmes, ce partitionnement est fait en fonction du nombre de processeurs utilisés.

2.4.1.2 Découpe récursive

La découpe récursive est une technique qui permet de paralléliser efficacement certains problèmes, elle est basée sur une restriction de la méthode **"diviser pour régner"**. Dans cette restriction, le problème est d'abord divisé en plusieurs sous-problèmes pouvant être traités en parallèle. Chacun des sous-problèmes est récursivement divisé en sous-problèmes de plus petite taille. La division s'arrête lorsque la taille du sous-problème est en dessous d'un seuil ou grain. Un grain définit un bloc de calculs (resp. de données) qui sera évalué (resp. accédé) localement de manière séquentielle sur un processeur. Un grain trop faible entraîne un surcoût de communication ce qui peut alors dégrader les performances du calcul. Un grain trop élevé entraîne peu de communication, mais diminue le degré de parallélisme potentiel.

2.4.2 Répartition de charge

La répartition de charge parfois appelée partage de charge (*Load Sharing*) consiste à utiliser au maximum les ressources disponibles d'une machine. Le cas idéal est d'occuper en permanence les ressources. Les performances d'un programme parallèle dépendent beaucoup de la manière dont les charges sont reparties entre les processeurs. Par exemple, prenons le cas des processeurs identiques avec des charges différentes sur chaque processeur, la performance globale sera celle du processeur le plus chargé, car les processeurs moins chargés ayant fini doivent attendre ce dernier. Si les vitesses et les temps de calcul sont connus, la répartition de charge peut être décidée de manière statique. Dans le cas contraire, on peut mettre en place une répartition dynamique. Par exemple, une fois que les processeurs ont fini leurs calculs, ils viennent demander de nouvelles charges. Pour que cette répartition de charge puisse être menée à bien, il est nécessaire d'avoir une représentation décrivant le programme. La section suivante décrit comment un programme peut être modélisé par un graphe.

2.5 Modélisation de l'exécution d'un programme parallèle

Pour qu'un algorithme parallèle puisse être ordonnancé et exécuté efficacement sur plusieurs machines, il est nécessaire d'avoir une représentation du programme décrivant l'algorithme avant le début de l'exécution. Cette représentation est faite soit avant le début de l'exécution du programme, soit à des moments clés de l'exécution du programme. Par exemple la création, la terminaison d'une tâche ou l'inactivité d'un processeur. Dans cette section, nous présentons les différentes représentations de l'exécution d'un algorithme parallèle.

Généralement trois grandes familles de graphes sont utilisées pour représenter l'exécution des programmes parallèles : graphe de dépendance, graphe de précédence et graphe de flot de données.

Le graphe de dépendance.
Un graphe de dépendance est un graphe non orienté. Les noeuds du graphe représentent les tâches voire aussi les données du programme et les arêtes représentent les dépendances entre les tâches (communications bi-points), ou entre une tâche et une donnée.

Le graphe de précédence.
Un graphe de précédence est un graphe orienté sans circuit. Les noeuds du graphe représentent les tâches du programme et les arêtes représentent les dépendances entre les tâches.
Les dépendances sont introduites pour régler des conflits d'accès à des données, elles peuvent être interprétées comme des communications entre les tâches. Dans ce type de graphe, une tâche est considérée prête dès que toutes les tâches qui la précèdent dans le graphe sont terminées.
Ce type de graphe est utilisé pour des programmes parallèles dynamiques comme, par exemple, les programmes récursifs ou les programmes contenant des boucles parallèles dont on ne connaît pas les bornes avant l'exécution.

Le graphe de flot de données.
Les dépendances entre tâches sont considérées comme des communications d'échange de données ; il est possible d'ajouter des informations sur des données communiquées dans le graphe : une donnée accédée par une tâche peut être caractérisée par le type d'opération que la tâche va effectuer sur elle :

- lecture seule : la tâche ne modifiera pas la donnée mais se contentera de lire sa valeur ;
- écriture seule : la tâche ne peut utiliser la donnée que pour l'affecter ;
- écriture en accumulation : plusieurs tâches vont effectuer une opération associative et commutative sur une donnée en écriture (un calcul de produit itéré par exemple).
- modification, ou lecture : la tâche va modifier la valeur de la donnée.

Ce graphe est appelé alors graphe de flot de données (GFD) [79].

Définition 2.5.1. *Le graphe de flot de données associé à un algorithme est le graphe* $G = (V, E)$ *tel que les tâches (*V_t*) et les données (*V_d*) forment l'ensemble* $V = V_t \cup V_d$ *des noeuds, et les accès des tâches sur les données forment l'ensemble E des arêtes. Ce graphe est biparti :* $E \subset (V_t \times V_d) \cup (V_d \times V_t)$.

La signification d'une arête est la suivante : soient $t \in V_t$ une tâche et $d \in V_d$ une donnée, l'arête $(t, d) \in E$ traduit un droit d'accès en écriture de la tâche t sur la donnée d, et l'arête (d, t) un droit d'accès en lecture de la tâche t sur la donnée d.
La différence entre les graphes de précédente et de flot de données se situe au niveau des synchronisations. Dans les graphes de flot de données, une tâche est considérée prête dès que les données qu'elle a en entrée sont disponibles. Notons néanmoins qu'un graphe de flot de données contient des informations d'un graphe de précédence.
Pour la régulation de charge, ce type de graphe fournit plus d'informations : puisque les données communiquées entre les tâches sont identifiées, les tailles des données peuvent également être fournies. Cette information peut être utilisée pour exploiter la localité des données lors du placement de tâches ou pour le routage des données accédées vers le site d'exécution d'une tâche.

2.6 Bibliothèques de programmation parallèle

2.6.1 OpenMP et Pthreads

OpenMP [41] et Pthreads [63] sont des bibliothèques de programmation parallèle dédiées aux architectures de machines à mémoire partagée. La bibliothèque OpenMP (*Open Multi-Processing*) se présente sous la forme d'un ensemble de directives (par exemple elles peuvent

servir à définir les sections parallèles, séquentielles ou critiques du code), de fonctions (par exemple savoir le nombre de processus disponibles à l'exécution), de variables d'environnement (leurs valeurs sont prises en compte à l'exécution une fois qu'elles sont fixées). OpenMP possède une interface de programmation pour les langages de programmation C/C++ et Fortran, et il est supporté par de nombreuses plate-formes (*e.g.* Linux ou Windows).

Pthreads (*POSIX Threads*) est une bibliothèque permettant la création et la manipulation des processus légers (en anglais, *thread*). Un processus léger est une abstraction permettant d'exprimer des multiples flots d'exécutions indépendants au sein d'un processus système (appelé processus lourd). Les processus légers appartenant au même processus lourd se partagent son contexte mémoire, mais chacun possède sa pile d'exécution. La gestion des processus légers nécessite seulement la manipulation de leur contexte d'exécution tandis que celle des processus lourds demande aussi la gestion de zone mémoire et de ressources (fichiers ouverts, verrous, etc...) ; aussi la gestion des processus légers est moins coûteuse que celle des processus lourds. Par exemple un changement de contexte pour un processus léger est de 10 à 100 fois plus rapide que celui d'un processus lourd. Pthreads définit un standard pour les opérations de manipulation des processus légers comme par exemple la création ou la destruction d'un processus léger sur processeur donné, la synchronisation (verrou, sémaphore, barrière de synchronisation, exclusion mutuelle, etc.) entre processus légers. Pthreads est largement utilisé sur les plate-formes fonctionnant avec les systèmes d'exploitation *Linux* et *Solaris*, mais existe aussi souvent sur des plate-formes *Windows*.

2.6.2 MPI

MPI (*Message Passing Interface*) [93] est une bibliothèque de fonctions permettant la programmation d'applications parallèles sur des machines à mémoire distribuée. Un programme MPI est basé sur le modèle SPMD (cf section 2.2) : une copie du programme s'exécute sur chaque processeur, une fonction MPI retourne le numéro du processeur ce qui permet d'exécuter un code spécialisé sur chaque processeur selon la valeur de son numéro. A l'initialisation, un nombre fixé de processus est créé et généralement chaque processeur exécute un processus unique. Les processus communiquent entre eux par échange de messages. La communication entre deux processus peut être de type point à point (envoi et réception d'une donnée d'un émetteur vers un destinataire) ; de type collective (diffusion d'un message à un groupe de processus, opération de réduction, distribution ou redistribution des données envoyées). Dans un programme MPI, un **commutateur** permet de connaître l'ensemble des processus actifs. MPI définit des fonctions qui à tout moment permettent de connaître le nombre de processus gérés dans un commutateur et le rang d'un processus dans le commutateur. La bibliothèque met à la disposition de nombreuses fonctions permettant au programmeur d'effectuer différents types d'envoi et de réception de messages (point à point bloquant (synchrone), point à point non bloquant (asynchrone), collectifs) et des opérations globales (barrière, réduction, diffusion).

2.6.3 Cilk

Cilk [35] est un langage de programmation parallèle destiné aux machines à mémoire partagée. Il est basé sur le langage C et il offre des primitives pour l'expression explicite du parallélisme (créations de tâches) et les synchronisations. Il utilise l'ordonnancement par vol de travail (dès qu'un processeur est inactif, il demande du travail à un autre processeur) pour placer les tâches prêtes à être exécutées de manière dynamique sur des processeurs disponibles. La description du parallélisme se fait à l'aide du mot clé *spawn* placé devant un appel de fonction. La sémantique de cet appel diffère de celle de l'appel classique d'une fonction : la procédure appelante peut continuer son exécution en parallèle de l'évaluation de la fonction appelée au lieu d'attendre son retour pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut pas utiliser le résultat de la fonction appelée sans synchronisation. Cette synchronisation est explicite par utilisation de l'instruction *sync*. Le modèle de programmation de Cilk est de type série parallèle (la tâche mère s'exécute en concurrence avec ses filles, jusqu'à rencontrer l'instruction *sync*). L'exemple 2.6.1 illustre le calcul en parallèle du $n^{ème}$ entier F_n de la suite de Fibonacci en Cilk. Cette suite F_n est défini par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \, \forall \, i \in \mathbb{N}, \; i \geq 2. \end{cases} \tag{2.1}$$

La procédure $Fibo$ a la spécification suivante :

- Entrée : un entier n
- Sortie : la valeur de F_n.

Exemple 2.6.1. *Calcul en parallèle du $n^{ème}$ entier de la suite de Fibonacci en Cilk .*

```
1   cilk int Fibo(int n) {
2       if (n < 2) return n
3       else {
4           int x ;
5           int y ;
6           x=spawn Fibo(n − 1) ;
7           y=spawn Fibo(n − 2) ;
8           sync ;
9           return (x + y) ;
10      }
11  } ;
12  cilk int main(int argc, char* argv[]) {
13      int n
14      n= atoi(argv[1]) ;
15      result = spawn Fibo(n) ;
16      sync ;
17      printf("result : % d", result) ;
18      return 0 ;
19  } ;
```

2.6.4 Intel TBB

Intel TBB (*Intel Thread Building Blocks*) [70, 95] est une bibliothèque générique C++ développée par la société Intel pour l'écriture des programmes sur des processeurs multicœurs. La bibliothèque contient des structures de données génériques et des algorithmes qui permettent de manipuler les contenus de ces structures. Grâce à ces structures et algorithmes, elle permet de décharger le développeur de la gestion des processus légers (création, synchronisation ou terminaison). En plus, elle permet de faire abstraction de l'architecture sous-jacente (le programme ne dépend pas des caractéristiques de l'architecture cible). Un programme TBB est découpé récursivement en plusieurs tâches et celles-ci sont placées dynamiquement sur des ressources disponibles, TBB implémente une stratégie d'ordonnancement par vol de travail similaire à celle Cilk. Dans un programme TBB l'utilisateur spécifie les tâches à exécuter ; une tâche TBB est un ensemble d'instructions indivisibles. La bibliothèque fonctionne avec différents compilateurs : *Gnu Compiler Collection (gcc), Intel C++ Compiler, Microsoft Windows*.

Exemple 2.6.2. *Calcul en parallèle du $n^{ème}$ entier de la suite de Fibonacci en TBB .*

```
1   class FibTask : public task {
2   public :
3       const long n ;
4       long* const res ;
5       FibTask( long _n, long* _res) : n(_n), res(_res) { } ;

6       task* execute() {
7           if (n < 2) return n
8           else {
9               long x, y ;
10                FibTask& a = *new( allocate_child()) FibTask(n-1, &x) ;
11                FibTask& b = *new( allocate_child()) FibTask(n-2, &y) ;
12                set_ref_count(3) ;
13                spawn(b) ;
14                spawn_and_wait_for_all(a) ;
15                *res = x+y ;
16          }
17          return NULL ;
18  } ;
19  long ParallelFib(long n) {
20      long res
21      FibTask& a = *new(task : :allocate_root()) FibTask(n, &res) ;
22      task : :spawn_root_and_wait(a) ;
23      return res ;
24  } ;
```

2.6.5 Athapascan/Kaapi

La bibliothèque Athapascan [37, 29, 75] fournit à l'utilisateur deux mots-clés pour définir les tâches et données du graphe de flot de données (GFD). Le graphe GFD est orienté, sans circuit et bipartite avec 2 types de nœuds correspondant aux tâches de calcul et aux données en assignation unique. Un arc dans le GFD entre un nœud tâche et un nœud donnée (resp. un nœud donnée et un nœud tâche) représente un accès en écriture (resp. en lecture) de la tâche sur la donnée. Elle utilise Kaapi [39, 38, 46, 47] comme interface de bas niveau qui fournit l'ordonnancement par vol de travail.

- Définition des tâches : **Fork**. L'utilisateur définit explicitement la granularité de son programme en encapsulant les procédures qui seront appelées de façon asynchrone. Ces tâches sont créées par appel de la procédure générique Fork de la bibliothèque instanciée avec le type de la tâche à créer. Le corps de la tâche sera implanté comme une classe C++ fournissant l'opérateur(), qui prend en entrée les paramètres de la tâche.
- Définition des données : **Shared**. La bibliothèque Athapascan définit une mémoire partagée afin de permettre aux tâches de coopérer. Cette mémoire peut contenir des objets typés ; dans la mémoire partagée, un objet de type T est déclaré comme de type Shared<T> où T représente le type spécifié par l'utilisateur. Les données Shared seront ainsi les noeuds données du GFD.

Pour permettre à l'environnement Athapascan de gérer les arêtes du GFD (les accès des tâches sur les données), chaque tâche spécifie au moment de sa création les accès qui seront effectués sur les données de la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance : les différentes données qui seront accédées sont spécifiées dans la liste des paramètres de la tâche et la description de ces droits d'accès (lecture r, écriture w, accumulation cw, modification r_w) est faite à l'aide d'un mécanisme de typage : on distinguera ainsi les données Shared accédées en lecture (Shared_r<T>) de celles accédées en écriture (Shared_w<T>).

Une tâche a un fonctionnement entièrement asynchrone par rapport à la tâche qui l'a créée. Mais la sémantique est lexicographique, donc naturelle pour un programmeur séquentiel ; toute lecture d'une donnée partagée voit la dernière écriture dans l'ordre. Cela implique donc, entre autre, que la tâche mère ne peut accéder aux résultats de la tâche fille : ces résultats seront exploités par une tâche nécessairement différente. Le système exécutif d'Athapascan garantit (afin de permettre à d'éventuels ordonnanceurs de faire des estimations fiables sur la durée d'exécution des tâches) que l'exécution d'une tâche a lieu sans aucune synchronisation. Pour cela, une tâche ne pourra débuter son exécution que si toutes les données en lecture sont prêtes, c'est-à-dire que toutes les tâches qui écrivent sur cette donnée sont terminées.
L'exemple 2.6.3 utilise les outils offerts au programmeur Athapascan pour le calcul naïf du $n^{ème}$ entier F_n de la suite de Fibonacci.

Exemple 2.6.3. *Calcul en parallèle du $n^{ème}$ entier de la suite de Fibonacci en Athapascan.*

```
1  struct Sum {
2      void operator() ( Shared_w<int> res, Shared_r<int> a, Shared_r<int> b) {
3          res.write(a.read()+b.read()) ;
4      } ;
```

```
5
6     struct Fibo {
7         void operator() ( Shared_w<int> res, int n) {
8             if (n < 2) {
9                 res.write(n) ;
10            }
11            else {
12                Shared<int> res1 ;
13                Shared<int> res2 ;
14                Fork<Fibo>() ( res1, n-1) ;
15                Fork<Fibo>() ( res2, n-2) ;
16                Fork<Sum>() ( res, res1, res2 ) ;
17            }
18        }
19    } ;
20
21                Fork<Fibo>()(res, n) ; // Appel principal
```

Les lignes 14 et 15 de l'exemple 2.6.3 déclarent un prototype de tâche Athapascan qui prend un entier n et retourne, par écriture (Shared_w) dans une variable partagée res, le résultat de F_n.
Les ligne 12 et 13 déclarent deux variables partagées qui stockent les résultats de F_{n-1} et F_{n-2} par les tâches créées aux lignes 14 et 15.

2.7 Conclusion

Dans ce chapitre, nous avons présenté quelques notions sur la programmation parallèle. Nous avons vu que plusieurs types d'architectures parallèles existent pour rendre en pratique possible l'exécution des programmes parallèles. Mais chaque type d'architecture a ses particularités (*e.g.* mémoire partagée ou distribuée). Deux modèles basés sur des architectures à mémoire partagée et distribuée qui sont utilisés couramment ont été présentés. Nous avons vu qu'un programme parallèle peut être modélisé par un graphe (graphe de dépendance, graphe de précédence ou graphe de flot de données) qui permet d'abstraire l'architecture sous-jacente sur laquelle le programme est exécuté. Deux manières (dynamique et statique) de répartition les charges d'un programme sur des architectures parallèles ont été présentées. Enfin, nous avons présenté quelques bibliothèques permettant de réaliser la programmation parallèle.

La programmation parallèle est plus complexe que la programmation séquentielle car elle nécessite la prise en compte d'éléments supplémentaires comme la gestion des synchronisations entre les ressources de l'environnement d'exécution ou les copies de données. Dans le chapitre

suivant nous allons présenter quelques problèmes liés au parallélisme et l'utilité des algorithmes adaptatifs.

Chapitre 3

Algorithmes parallèles adaptatifs

Dans ce chapitre, nous étudions la notion d'algorithme adaptatif et nous précisons la signification que nous lui donnons dans la section 3.1. Dans la section 3.2, nous présentons l'intérêt d'utilisation des algorithmes parallèles adaptatifs. Dans la section 3.3, nous illustrons l'intérêt d'adapter des algorithmes en prenant comme cas d'étude le calcul des préfixes. Dans cette étude de cas, nous proposons un nouvel algorithme statique optimal pour le calcul des préfixes dans la section 3.3.3. Dans la section 3.2.3, nous présentons les différentes techniques existantes permettant d'adapter les algorithmes parallèles. Enfin dans la section 3.5, nous présentons quelques bibliothèques implémentant des algorithmes parallèles adaptatifs.

3.1 Qu'est qu'un algorithme adaptatif ?

Définition 3.1.1. *Un algorithme adaptatif est un algorithme qui est capable de changer automatiquement son comportement en fonction de son contexte d'exécution (données manipulées par l'algorithme, paramètres de configurations de l'environnement d'exécution, occupation des ressources) pour atteindre des performances optimales.*

Lorsque plusieurs choix de comportements sont possibles, une stratégie de décision doit être définie pour déterminer lesquels sont les meilleurs. Lors de son exécution, l'algorithme adaptatif doit respecter les contraintes qui lui sont imposées par cette stratégie afin d'atteindre les performances optimales. Cette stratégie de décision peut dépendre de la configuration matérielle de l'environnement ou en être indépendante. Elle peut être décidée avant l'exécution ou au cours de l'exécution. En fonction de la stratégie choisie, nous classifions les algorithmes adaptatifs en deux grandes classes.

Définition 3.1.2. *Un algorithme adaptatif est qualifié de* ***dépendant des ressources*** *(resource-aware en anglais) lorsque le choix de sa stratégie de décision est basé sur les paramètres de configurations de son environnement d'exécution (nombre de processeurs, taille des caches, largeur de la bande passante, etc...).*

Le choix de cette stratégie de décision suppose que les contraintes (gestion du gain de parallélisme, réduction des défauts de cache, etc..) pouvant permettre à l'algorithme adaptatif d'atteindre des performances optimales peuvent être connues avant l'exécution. Une fois ce choix fait, l'algorithme adaptatif peut se comporter d'une manière unique sans changer de comportement jusqu'à la fin de son exécution. Par exemple la bibliothèque d'algèbre linéaire (*e.g.* multiplications de vecteurs ou matrices) ATLAS [97] implémente des algorithmes adaptatifs **dépendants des ressources**. Dans ATLAS les calculs se font par blocs, la taille du bloc est choisie en fonction des mesures de performances (comparaisons de différentes implémentations) effectuées lors l'installation. Par exemple lors de l'appel à l'algorithme de multiplication de matrices, le choix de l'algorithme à exécuter a déjà été établi au préalable lors de l'installation.

Définition 3.1.3. *[36] Un algorithme adaptatif est qualifié de **indépendant des ressources** (resource-oblivious en anglais) lorsque sa stratégie de décision n'utilise aucun des paramètres de configuration de son environnement d'exécution (nombre de processeurs, hiérarchie de caches, . . .) et peut changer dynamiquement sa stratégie de résolution en fonction des variations de son environnement d'exécution (disponibilité variable des ressources, variation instantanée de la puissance des ressources, charge réseau, débit et volume mémoire, . . .). Dans ce cas la stratégie de décision est prise en cours d'exécution.*

Les algorithmes **cache-indépendants** (*cache-oblivious algorithms* en anglais) étudiés par Frigo et al [36] sont des exemples d'algorithmes de cette classe. Un algorithme **cache-indépendant** est conçu pour exploiter efficacement la mémoire cache du processeur sans tenir compte de la taille de celle-ci (ou sa longueur des lignes, etc). Frigo et al [36] ont proposé plusieurs algorithmes **cache-indépendants** asymptotiquement optimaux comme le calcul rapide des transformées de Fourier discrètes (*FFTW*), le tri, la multiplication de matrices, et plusieurs autres algorithmes. En générale, un algorithme **cache-indépendant** est un algorithme basé sur la méthode "diviser pour régner" qui permet de diviser le problème à résoudre en sous problèmes plus petits. La division du problème s'arrête quand la mémoire cache pourra contenir le sous-problème.

Dans cette thèse, nous allons étudier les algorithmes **processeur-indépendants**. Nous disons qu'un algorithme adaptatif est un algorithme **processeur-indépendant** si il peut atteindre des performances optimales sans dépendre du nombre de processeurs et de leurs vitesses respectives. Dans [11] Bernard et al ont étudié un algorithme **processeur-indépendant** pour le traitement parallèle de flux. Dans les chapitres suivants, nous présenterons plusieurs algorithmes **processeur-indépendants** comme le calcul parallèle des préfixes [90], la fusion de deux listes triées, la partition [89], et d'autres algorithmes [90].

Pour une classification détaillée des algorithmes adaptatifs, nous invitons le lecteur à lire l'article de Roch et al [23].

3.2 Pourquoi a t-on besoin d'adapter des algorithmes ?

Le travail global (arithmétique, copie de données, création de tâches de calculs, etc.) d'un algorithme statique (en opposition à un algorithme adaptatif) est toujours le même pour un même problème donné et une architecture donnée. En fait l'algorithme statique dépend soit de la taille des données en entrée, soit des caractéristiques de l'architecture cible (nombre de processeurs par exemple). Par contre, un algorithme adaptatif fera varier son travail en fonction des données qu'il traite ou en fonction de l'architecture sur laquelle il s'exécute. Cette variation de travail permet à l'algorithme adaptatif d'être plus efficace (moins d'opérations, moins de copie, etc.) que son équivalent statique. Dans cette section, nous allons présenter les besoins d'un algorithme adaptatif en programmation parallèle.

3.2.1 Programmation parallèle sur des processeurs à vitesses variables

Depuis 2005, les ordinateurs standards (fixes et portables) deviennent des machines multicœurs. Et pour pouvoir bénéficier pleinement de la puissance de calcul fournie par ces machines, les programmes doivent exploiter le parallélisme. Mais la parallélisation efficace d'un programme sur de telles architectures est difficile. Car elle est généralement exploitée en concurrence par plusieurs applications en contexte multi-utilisateurs ; aussi le nombre de processeurs physiques et leurs vitesses relatives par rapport à une application peut varier en cours d'exécution de manière non prédictible. De plus, les différents cœurs peuvent être hétérogènes (fréquences différentes, mémoire non uniforme). Or, tout algorithme parallèle introduisant un surcoût, en cas de surcharge, un algorithme séquentiel peut s'avérer plus performant en temps écoulé qu'un algorithme parallèle. L'implémentation efficace nécessite alors un algorithme parallèle qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles.

3.2.2 Programmation parallèle sur des architectures hétérogènes et dynamiques

Dans le développement d'algorithmes efficaces sur des architectures hétérogènes et dynamiques comme la grille de calcul, plusieurs facteurs doivent être nécessairement pris en compte. En effet, les composants (*e.g.* machines, infrastructures de communication) dans ces types d'architectures sont souvent différents et le nombre de ressources disponibles peut varier à tout moment du calcul. Dans la grille de calcul par exemple, les machines peuvent être des machines multiœurs, SMP, de bureau, ou des portables. Les réseaux de communications qui sont utilisés pour interconnecter ces machines peuvent être aussi divers (Ethernet, Myrinet, ou Infini Band). L'exploitation efficace de ces ressources hétérogènes, distribuées et dynamiques nécessite alors le développement d'applications capables de s'adapter automatiquement au contexte d'exécution, qui varie dynamiquement en fonction de l'occupation des ressources et des données manipulées par l'algorithme.

3.2.3 Minimiser le surcoût lié au parallélisme

La parallélisation d'un algorithme introduit des surcoûts très importants par rapport à l'algorithme séquentiel efficace. Ces surcoûts peuvent être dus à la création de tâches, des copies de données, de communication de données, des synchronisations, des surcoûts arithmétiques. D'une manière classique, paralléliser un programme consiste à définir chacune des tâches à exécuter (concurrentes ou non) ainsi que les données nécessaires pour leurs exécutions ; le programme fixe ainsi la granularité des calculs et la granularité des données. Nous rappelons qu'un grain de calculs (resp. données) est un bloc de calculs (resp. données) qui sera évalué localement de manière séquentielle par un seul processeur. La performance du programme parallèle dépend énormement du choix de ce grain. Deux points de vue antagonistes sont considérés pour choisir ce grain [26]. D'un point de vue algorithmique pratique, le grain peut être fixé en fonction du nombre de ressources, généralement supposées identiques, disponibles lors de l'exécution. L'inconvénient d'une telle approche est que le nombre de ressources et leurs puissances peuvent être variables, comme c'est le cas dans la programmation sur des processeurs multi-cœurs, sur grappe en contexte multi-utilisateur ou, plus encore, sur grille de machines. D'un point de vue algorithmique théorique [49], le grain est fixé pour minimiser le temps d'exécution sur un nombre infini de processeurs tout en conservant un nombre total d'opérations proche du meilleur algorithme séquentiel (on parle d'algorithme parallèle *optimal*). L'inconvénient d'une telle approche est que, lors d'une exécution effective sur un nombre toujours restreint de ressources, le placement et l'entrelacement de ces nombreuses tâches entraîne un surcoût.

Ainsi, supposons que l'exécution du programme parallèle soit en fait effectuée sur un seul processeur, les autres processeurs étant mobilisés pour d'autres calculs. Dans ce cas, exécuter l'algorithme parallèle au grain préalablement choisi peut être pénalisant par rapport à l'exécuter séquentiellement en évitant tous les surcoûts incontournables liés au parallélisme : création de tâches, copie de données.

3.3 Le calcul parallèle des préfixes : cas d'étude

Étant donnés $x_0, x_1, \ldots x_n$, les n préfixes π_k, pour $1 \leq k \leq n$, sont définis par $\pi_k = x_0 \star x_1 \star \ldots \star x_k$ où $\star$ est une loi associative. Le calcul séquentiel de ces préfixes peut être effectué en n opérations par une boucle itérative simple :

$\pi[0] = x[0]$
$Pour\ i = 1$ à n
    $\pi[i] = \pi[i-1] \star x[i]$

Le calcul des préfixes est un problème très étudié [58, 17] en parallélisme sur lequel nous reviendrons en détail dans le chapitre 4. Ici, nous l'utilisons pour illustrer les trois besoins de l'adaptatif expliqués dans les sections précédentes. Avant ces illustrations, nous donnons une borne de ce calcul sur des processeurs à vitesses variables dans la section 3.3.1. Puis dans la section 3.3.2, nous rappelons un algorithme parallèle sur processeurs identiques pour les préfixes, ensuite nous proposons un nouvel algorithme statique optimal pour des processeurs

identiques dans la section 3.3.3, et enfin dans la section 3.3.4, nous présentons un algorithme récursif . Pour chaque algorithme, nous montrons ces limites.

3.3.1 Borne inférieure

Dans cette section, nous donnons une borne inférieure du calcul des préfixes sur p processeurs à vitesse variable. Nous rappelons qu'à un instant t la vitesse $\Pi_i(t)$ d'un processus i, $1 \leq i \leq p$, dépend du nombre de processus (correspondant à d'autres applications) ordonnancés sur le processeur physique auquel il est affecté. Pour une exécution parallèle de durée T (en temps écoulé) sur p exécuteurs, on définit la vitesse moyenne $\Pi_{ave} = \frac{\sum_{t=1}^{T}\sum_{i=1}^{p}\Pi_i(t)}{p.T}$ et la vitesse maximale de tous les processeurs par $\Pi_{max} = max_{i=0,\ldots,p-1;t=1,\ldots,T}\Pi_i(t)$. Le théorème suivant donne une borne inférieure du temps d'exécution du calcul parallèle des préfixes sur p processeurs à vitesse variable.

Théorème 3.3.1. *On suppose qu'une opération $\star$ prend un temps unité. Une borne inférieure du temps d'exécution T_p de tout calcul parallèle des préfixes avec $n+1$ données en entrée sur p processeurs de vitesse moyenne Π_{ave} et de vitesse maximale Π_{max} est*

$$T_p \geq \frac{2n}{p.\Pi_{ave} + \Pi_{max}}$$

Démonstration. Soit G le graphe de profondeur d représentant l'exécution de l'algorithme parallèle. Soit A le sous-graphe de G qui contient tous les nœuds opérations prédécesseurs de π_n. Soit B le sous-graphe qui contient tous les noeuds de G qui ne sont pas dans A : $B = G - A$. Dans $\pi_n = x_0 \star x_1 \star \ldots \star x_n$, chacun des x_i ne peut apparaître qu'une seule fois. Donc, pour toute entrée $x \in \{x_0, x_1, \ldots, \}$, il ne peut exister dans A qu'un unique chemin entre x et π_n ; A est donc un arbre.
De plus, tout noeud interne représente une opération $\star$ qui est binaire. Donc A est un arbre binaire et qui a $n+1$ feuilles ; il contient donc exactement n nœuds internes correspondant aux opérations $\star$. Donc A effectue n opérations $\star$, car le nombre de nœuds est ici égal au nombre d'opérations $\star$. De plus, soit π_k un préfixe obtenu en sortie d'un noeud $\star$ de A. Ce noeud est nécessairement un successeur dans A de la feuille x_0. Tout préfixe de sortie fait intervenir la valeur x_0 en entrée. Tout nœud interne qui calcule un préfixe dans A est donc successeur de x_0. Comme A est de profondeur d, x_0 admet au plus d nœuds successeurs dans A. Donc A calcule au plus d préfixes π_k. Comme A calcule au plus d préfixes, B calcule au moins $n-d$ préfixes π_k. De plus un nœud $\star$ de B ne peut calculer au plus qu'un préfixe (éventuellement 0 si c'est un calcul intermédiaire). Donc B contient au moins $n-d$ nœuds ; donc le nombre d'opérations dans G est au moins égal à $2n-d$ (à noter que la première partie de cette preuve est similaire à celle établie dans [33], théorème 2, pour circuits restreints avec $d = \log n$).
Durant T_p unités de temps, au plus $p.\Pi_{ave}$ opérations ont été effectuées ; alors $p.\Pi_{ave}.T_p \geq 2n-d$ (1). De plus, puisque G a un chemin critique de d opérations, $\Pi_{max}.T_p \geq d$ (2). En additionnant (1) et (2) on obtient $(p.\Pi_{ave} + \Pi_{max}).T_p \geq 2n$. ❏

Dans le cas des processeurs identiques ($\Pi_{ave} = \Pi_{max} = 1$), la borne inférieure est de $\frac{2n}{p+1}$. Nous allons maintenant présenter dans la section suivante trois algorithmes parallèles, dont nous proposons un qui atteint cette borne sur des processeurs identiques.

3.3.2 Un algorithme statique sur processeurs identiques

Dans cette section, nous rappelons un algorithme parallèle pour le calcul des n préfixes $(\pi_i)_{i=1,\ldots,n}$ qui est optimal asymptotiquement sur p processeurs identiques. Il est basé sur une découpe en $p+1$ blocs $B_0, \ldots B_p$ de même taille (à un élément près) des $n+1$ éléments en entrée $(x_i)_{i=0,\ldots,n}$. Pour simplifier, on suppose que "$n+1$" est multiple de "$p+1$" : chaque bloc B_i contient $K = \frac{n+1}{p+1}$ éléments consécutifs.

1. **Etape 1** : En parallèle pour $i = 0, \ldots, p-1$, on calcule sur le processeur i les préfixes séquentiels (les préfixes calculés séquentiellement) du bloc B_i. Soient α_i le dernier préfixe du bloc B_i. On remarque que les préfixes $(\pi_j)_{j=1,\ldots,K}$ du bloc B_0 sont ainsi calculés.
2. **Etape 2** : On calcule les p préfixes $\beta_0 = \alpha_0$, $\beta_1 = \alpha_0 \star \alpha_1, \ldots, \beta_{p-1} = \beta_{p-2} \star \alpha_{p-1}$ des valeurs $\alpha_0, \ldots, \alpha_{p-1}$.
3. **Etape 3** : Pour $i \ldots p$, soit B'_i le bloc de K éléments obtenus en insérant β_{i-1} en tête du bloc B_i. En K tops, sur chaque processeur i, on calcule les préfixes du bloc B'_i. On obtient ainsi tous les préfixes π_i.

La figure 3.1 illustre l'algorithme décrit.

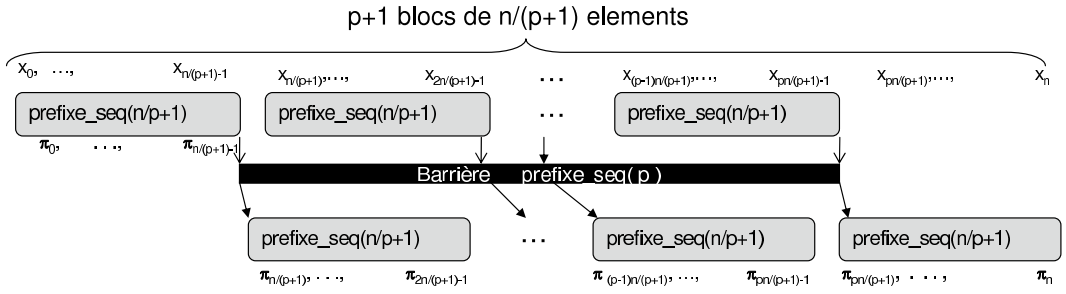

FIG. 3.1 – Algorithme parallèle du calcul des préfixes sur p processeurs identiques

Le temps d'exécution de cet algorithme est $2K + p - 2 = \frac{2(n+1)}{p+1} + p - 2 \sim_{n\to+\infty} \frac{2n}{p+1}$ donc asymptotiquement optimal. Son nombre d'opérations $p(K-1) + p - 1 + pK = 2(n+1)\frac{p}{p+1} - 1 \sim_{n\to+\infty} 2n\frac{p}{p+1}$.

Ce nombre total d'opérations effectuées est toujours le même, même si les p processeurs initialement prévus pour l'exécution ne sont pas tous disponibles. Ainsi supposons qu'un seul processeur exécute l'algorithme, les autres processeurs étant mobilisés pour d'autres calculs : le nombre d'opérations effectuées par ce processeur est alors donc $2n\frac{p}{p+1}$ qui est à un facteur de $\frac{2p}{p+1}$ du nombre d'opérations de l'algorithme séquentiel optimal. Cet algorithme parallèle n'est optimal que sur p processeurs identiques et tous dédiés au calcul. Il n'est pas performant

si l'on dispose d'une machine avec des processeurs différents, ou d'une machine utilisée par plusieurs utilisateurs car la charge des processeurs varie en cours d'exécution, ou bien encore si le temps d'une opération varie. Le temps d'exécution de l'algorithme sera alors le temps de terminaison du processeur le plus lent. Si nous supposons que la vitesse du processeur le plus lent est de Π_{min}, alors cet algorithme est à un facteur de $\frac{p.\Pi_{ave}+\Pi_{max}}{(p+1)\Pi_{min}}$ de l'optimal. Pour traiter ce problème, nous devons recourir à un algorithme adaptatif qui peut s'adapter automatiquement et dynamiquement aux processeurs effectivement disponibles.

3.3.3 Un nouvel algorithme statique optimal sur processeurs identiques

Dans cette section, nous proposons un algorithme statique optimal qui atteint la borne $\left\lceil \frac{2n}{p+1} \right\rceil$ pour $n >= p(p+1)-1$ sur des processeurs identiques. Le surcoût de synchronisation de l'algorithme proposé est $O(p)$, qui est très faible rapport au surcoût de synchronisation de l'algorithme de Nicolau et Wang [94]. Donc cet algorithme s'exécute efficacement même si le temps d'une opération $\star$ est très petit (grain fin).

3.3.3.1 Algorithme

Nous rappelons tout d'abord que le tableau en entrée contient $n+1$ éléments ($x_0, x_1, \ldots x_n$). Pour effectuer le calcul parallèle des préfixes d'une manière statique de ces $n+1$ éléments, nous partitionnons le tableau en $p+1$ blocs de tailles éventuellement différentes. Soit $n+1 = s(p+1)+q$ avec $0 \leq q < p+1$ et $s \geq p$ où q et s sont des entiers. Nous désignons par B_i pour $i = 0, \ldots, p$ le bloc de rang i. Nous calculons la taille de chaque bloc selon la valeur de q.

- Si $q = 0$, tous les blocs B_i ($i = 0, \ldots, p-1$) sont de même taille égale à $s+1$ sauf le bloc B_p qui est de taille égale [1] $s-p$.
- Si $q \geq 1$, on a pour $0 \leq i \leq p-q$ le bloc B_i contient $s+1$ éléments et pour $p-q+1 \leq i \leq p-1$, le bloc B_i contient $s+2$ éléments et le bloc B_p contient $s-p+1$ éléments.

L'algorithme que nous proposons est décrit par l'algorithme 3.3.1. Dans l'algorithme x désigne un pointeur sur le tableau en entrée, res un pointeur sur le tableau en sortie, $size$ la taille du tableau, p le nombre de processeurs exécutant l'algorithme et *op* l'opérateur binaire associatif. Sur la ligne 4 de l'algorithme, un processeur exécute la fonction *Calcul_Taille_Bloc* qui calcule les indices de début ($B[i].start$) et de fin ($B[i].end$) du bloc B_i ($0 \leq i \leq p$) selon la valeur de q. Chaque processeur calcule séquentiellement les préfixes (lignes 8 et 9) du bloc i dont l'indice lui a été attribué. On remarque que les préfixes finaux $(\pi_i)_{i=0;\ldots,B[0].end-1)}$ du bloc B_0 sont ainsi calculés. Ces calculs séquentiels sur p blocs correspondent à l'étape 1. Les étapes 2 à p (ligne 12) correspondent aux calculs des $p(p-1)$ préfixes finaux. A chaque étape $j+1$ ($1 \leq j \leq p-1$), $p-1$ processeurs finalisent les $p-1$ premiers préfixes partiels (ligne 17) du bloc B_j à l'aide du dernier préfixe final calculé ($\pi_{B[j-1].end-1}$) du bloc B_{j-1}, et un processeur

[1] On remarque que si $s = p$, alors le bloc B_p est de taille nulle. Il y a alors p blocs seulement.

anticipe (ligne 19) le calcul du préfixe final ($\pi_{B[j].end-1}$) permettant de finaliser les préfixes partiels du bloc suivant. Les préfixes finaux du bloc B_0 ayant été calculés à l'étape 1, et p préfixes finaux du bloc B_k ($1 \leq k \leq p-1$) ayant été calculés à l'étape $k+1$, l'étape $p+1$ consiste à finaliser (ligne 25) les préfixes partiels restants du bloc B_k ($1 \leq k \leq p-1$) et calculer les préfixes finaux (ligne 27) du bloc B_p. Les fonctions *Calcul_Taille_Bloc*, *Prefix_seq* et *Prefix_final* sont données sur les lignes 29, 38 et 42.

Algorithme 3.3.1

```
1  Parallel_Prefix_Computation(Elt* x, Elt* res, int size, int p, BinOp op) {
2      /* Un seul processeur calcule les taille des blocs ; les autres doivent attendre avant de
3      passer aux étapes suivantes */
4      Calcul_Taille_Bloc(B, size, p) ;
5      /* Etape 1 : Les p processeurs calculent en parallèle des préfixes séquentiels par bloc */
6      Parallel
7          Pour i = 0 à p − 1 { /* exécutée par p processeurs */
8              res[B[i].start] = x[B[i].start] ;
9              Prefix_seq(x, res, B[i].start + 1, B[i].end, op) ;
10         }
11     /* Etape 2 à p */
12     Pour j = 1 à p − 1 {
13     /* A l'étape j + 1, p − 1 processeurs calculent chacun un préfixe final
14         pendant que le dernier processeur calcule le prochain préfixe. */
15         Parallel
16             Pour i = 0 à p − 2 /* exécutée par p-1 processeurs */
17                 res[B[j].start + i] = op(res[B[j − 1].end − 1], res[B[j].start + i]) ;
18             /* pendant qu'un processeur calcule le prochain préfixe. */
19             res[B[j].end − 1] = op(res[B[j − 1].end − 1], res[B[j].end − 1]) ;
20     } /* Il reste maintenant : s − p + 1 préfixes à calculer dans le dernier bloc
21     et s − p + 1 ou s − p + 2 préfixes à finaliser dans chacun des p − 1 blocs précédents.*/
22     /* Etape p + 1 */
23     Parallel
24         Pour i = 1 à p − 1 /* exécutée par p-1 processeurs */
25             Prefix_final(res, B[i].start + p − 1, B[i].end − 1, p, op) ;
26         /* pendant qu'un processeur calcule le préfixe final du dernier bloc. */
27         Prefix_seq(x, res, B[p].start, B[p].end, op) ;
28 }
29 Calcul_Taille_Bloc(int *B, int size, int p) {
30     s = size/(p + 1) ; q = size%(p + 1) ; B[0].start = 0 ; B[0].end = s + 1 ; r = s + 1 ;
31     Pour i = 1 à p − 1 {
32         Si(i > p − q) { B[i].start = r ; r+ = s + 2 ; B[i].end = r ; }
33         Sinon { B[i].start = r ; r+ = s + 1 ; B[i].end = r ; }
34     }
35     Si(q==0) { B[p].start = r ; r+ = s − p ; B[p].end = r ; }
36     Sinon { B[p].start = r ; r+ = s − p + 1 ; B[p].end = r ; }
37 }
38 Prefix_seq(Elt* x, Elt* res, int start, int end, BinOp op) {
39     Pour j = start à end − 1
40         res[j] = op(res[j − 1], x[j]) ;
41 }
42 Prefix_final(Elt* res, int start, int end, int p, BinOp op) {
43     Pour j = start à end − 1
44         res[j] = op(res[start − p], res[j]) ;
45 }
```

Algorithme 3.3.1: Algorithme parallèle statique du calcul des préfixes. Les étapes sont synchrones c'est à dire l'étape k ne peut commencer que si l'étape $k-1$ est terminée.

Exemple 3.3.1. *L'exécution de l'algorithme est illustrée par la figure 3.2 pour* $n = 20$ *et* $p = 4$ *et l'opérateur* $\star$ *est la somme de deux entiers machine (int). Le tableau est divisé en* 5 *blocs dont les* 4 *premiers blocs sont de taille* 5 *et le dernier bloc de taille* 1*. Initialement les* 4 *premiers blocs sont attribués aux* 4 *processeurs ; ensuite chaque processeur effectue localement sur son bloc un calcul séquentiel de préfixe (étape 1 sur la figure 3.2). A la fin de l'exécution de la première étape les préfixes finaux du premier bloc sont déjà calculés. Pendant la deuxième étape chaque processeur finalise un préfixe partiel du deuxième bloc ; les processeurs* P_0*,* P_1 *et* P_2 *finalisent à l'aide du dernier préfixe calculé du premier bloc (qui est égal à* 15*) les* 3 *premiers préfixes partiels (*$21 = 15{+}6\,, 28 = 15{+}13\,, 36 = 15{+}21$ *) et le processeur* P_3 *finalise le dernier préfixe partiel (*$55 = 15 + 40$*) qui permettra de finaliser les préfixes partiels du troisième bloc. A la troisième étape les processeurs* P_0*,* P_1 *et* P_2 *calculent (*$66 = 55{+}11\,, 78 = 55{+}23\,, 91 = 55{+}36$ *et* P_3 *calcule* $120 = 55 + 65$*. A la quatrième étape les processeurs* P_0*,* P_1 *et* P_2 *calculent* $136 = 120 + 16\,, 153 = 120 + 33\,, 171 = 120 + 51$ *et* P_3 *calcule* $210 = 120 + 90$*. A la cinquième étape* P_0 *finalise les préfixes partiels restants du deuxième bloc (*$45 = 15 + 30$*),* P_1 *finalise les préfixes partiels restants du troisième bloc (*$105 = 55 + 50$*),* P_2 *finalise les préfixes partiels restants du quatrième bloc (*$190 = 55 + 70$*) et* P_3 *calcule séquentiellement les préfixes du dernier (cinquième) bloc (*$231 = 210 + 21$*).*

3.3.3.2 Analyse théorique

Dans toute la suite on suppose que le temps d'une opération $\star$ est constant et prend une unité de temps. Le théorème suivant donne une borne supérieure du temps d'exécution de l'algorithme proposé.

Théorème 3.3.2. *Si* $0 \leq q \leq 1$ *ou* $\frac{p+3}{2} \leq q \leq p$*, alors si nous négligeons le surcoût de synchronisation entre les étapes de l'algorithme, le temps d'exécution de l'algorithme 3.3.1 sur* p *processeurs vérifie*

$$T_p = \left\lceil \frac{2n}{p+1} \right\rceil$$

Démonstration. Soient $s_{max} = max_{i=0,\ldots,p-1} taille(B_i)$, $T^{(1)}$ le temps d'exécution de l'étape 1, $T^{(2)}$ le temps d'exécution des étapes 2 à p et $T^{(3)}$ le temps d'exécution de l'étape $p+1$.
L'étape 1 dure $(s_{max} - 1)$ unités de temps (préfixe séquentiel par bloc, ligne 9 de l'algorithme 3.3.1). Les étapes 2 à p durent $p-1$ unités de temps, car chacune de ces étapes fait une unité de temps (lignes 17 et 19 de l'algorithme 3.3.1). L'étape $p+1$ dure $(s_{max} - p)$ (lignes 25 et 27 de l'algorithme 3.3.1) unités de temps.
Soit $T_p = T^{(1)} + T^{(2)} + T^{(3)}$ le temps total de l'algorithme, on a :
$T_p = (s_{max} - 1 + p - 1 + s_{max} - p) = 2s$ si $0 \leq q \leq 1$ car dans ce cas $s_{max} = s + 1$.
$T_p = (s_{max} - 1 + p - 1 + s_{max} - p) = 2(s + 1)$ si $q > 1$ car dans ce cas $s_{max} = s + 2$.
Puisque $n + 1 - q = s(p + 1)$, donc $\frac{2n}{p+1} = 2s + 2\frac{q-1}{p+1}$, et on a :
si $0 \leq q \leq 1$, on a $-1 < 2\frac{q-1}{p+1} \leq 0$, d'où $\left\lceil \frac{2n}{p+1} \right\rceil = 2s = T_p$.
Si $\frac{p+3}{2} < q \leq p$, on a $1 < 2\frac{q-1}{p+1} < 2$, d'où $\left\lceil \frac{2n}{p+1} \right\rceil = 2s + 2 = T_p$. ❏

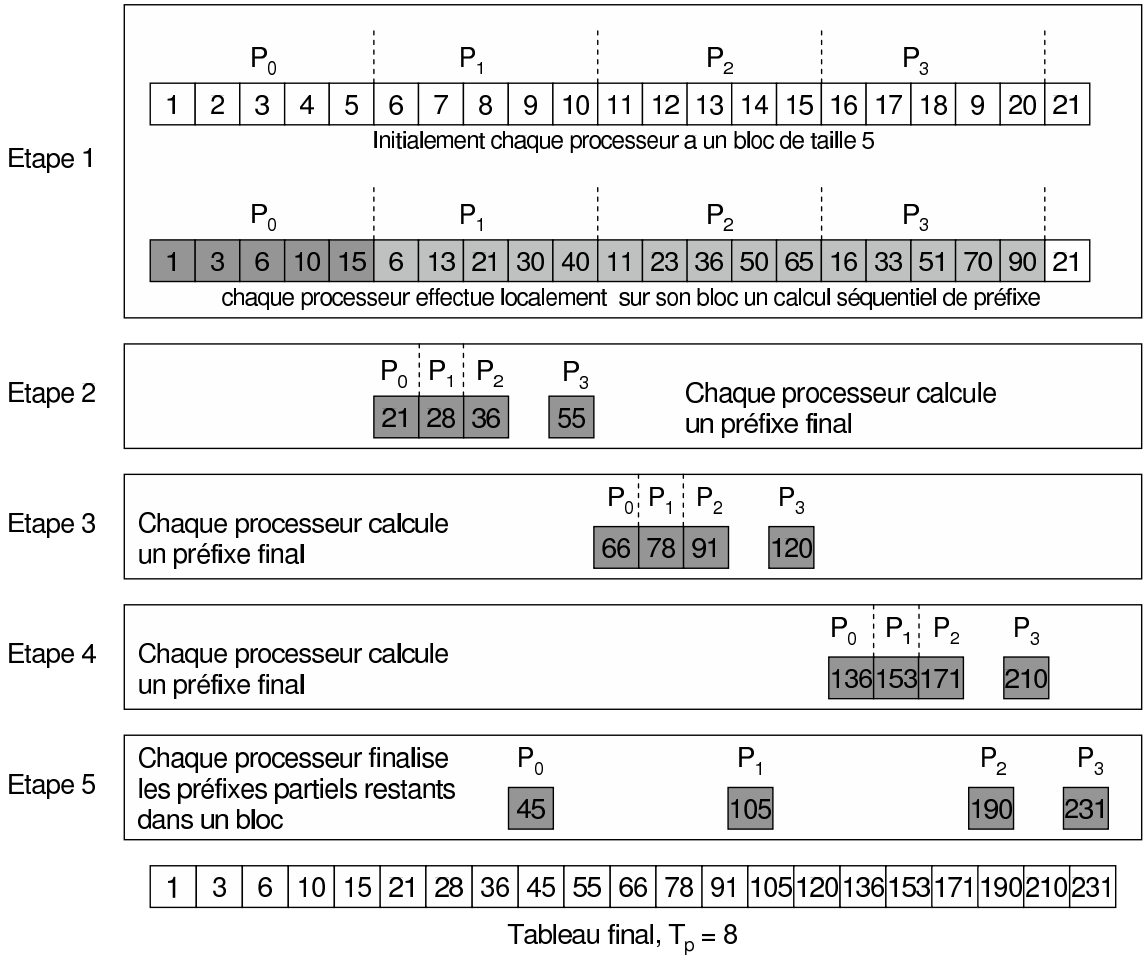

FIG. 3.2 – Illustration de l'algorithme 3.3.1 pour $n = 20$ et $p = 4$ et l'opérateur $\star$ est la somme de deux entiers machine (int)

Théorème 3.3.3. *Si $2 \leq q \leq \frac{p+3}{2}$, alors si nous négligeons le surcoût de synchronisation entre les étapes de l'algorithme, le temps d'exécution de l'algorithme 3.3.1 sur p processeurs vérifie*

$$T_p = \left\lceil \frac{2n}{p+1} \right\rceil + 1$$

Démonstration. Dans la preuve du théorème 3.3.2 on a montré que si $q > 1$, $T_p = 2(s+1)$, donc pour $2 \leq q \leq \frac{p+3}{2}$, $T_p = 2(s+1)$. On a :
si $2 \leq q \leq \frac{p+3}{2}$, on a $0 \leq 2\frac{q-1}{p+1} \leq 1$, d'où $\left\lceil \frac{2n}{p+1} \right\rceil = 2s+1 = T_p - 1$, d'où $T_p = \left\lceil \frac{2n}{p+1} \right\rceil + 1$. ❑

Exemple 3.3.2. *Soit $n+1 = 200, p = 9$; on aura donc $s = 20$ et $q = 0$.*
On a donc $T^{(1)} = 20, T^{(2)} = 8$ et $T^{(3)} = 12$ d'où $T_9(200) = 40$, et on a $T_9{}^{borne-inf}(200) = \left\lceil \frac{2*(199)}{10} \right\rceil = \lceil 39,8 \rceil = 40$.

Soit $n+1 = 101, p = 2$; on aura donc $s = 33$ et $q = 2$.

On a donc $T^{(1)} = 34, T^{(2)} = 1$ *et* $T^{(3)} = 33$ *d'où* $T_9(101) = 68$, *et on a* $T_2{}^{borne-inf}(101) = \left\lceil \frac{2*(100)}{3} \right\rceil = \lceil 66, 67 \rceil = 67$.

Théorème 3.3.4. *Si* $0 \leq q < \frac{p+3}{2}$, *alors si nous désignons par* O_p *le nombre d'opérations effectuées de l'algorithme 3.3.1 sur* p *processeurs,* O_p *vérifie*

$$O_p = \left\lceil \frac{2np}{p+1} \right\rceil$$

Démonstration. A l'étape 1 les p processeurs font $sp+q-1$ opérations cumulées si $q \neq 0$ sinon sp, soit $O^{(1)} = sp + q - 1$ si $q \neq 0$ sinon $O^{(1)} = sp$. A l'étape 2 ils font $p(p-1)$ opérations cumulées, soit $O^{(2)} = p(p-1)$. A l'étape 3 ils font $(s+1-p)(p-1) + s - p + q$ opérations cumulées. On a $O_p = O^{(1)} + O^{(2)} + O^{(3)} = 2sp + 2q - 2$ si $q \neq 0$ sinon $O_p = 2sp - 1$.
Puisque $n + 1 - q = s(p+1)$, donc $\frac{2np}{p+1} = 2sp + 2\frac{(q-1)p}{p+1} = 2sp + 2(q-1) - \frac{2(q-1)}{p+1}$, et on a :
si $q = 0$, on a $-2 < 2\frac{(q-1)p}{p+1} \leq -1$, d'où $\left\lceil \frac{2np}{p+1} \right\rceil = 2sp - 1 = O_p$.
Si $1 \leq q < \frac{p+3}{2}$, on a $0 \leq 2\frac{q-1}{p+1} < 1$, d'où $\left\lceil \frac{2np}{p+1} \right\rceil = 2s + 2(q-1) = O_p$. ❑

Théorème 3.3.5. *Si* $\frac{p+3}{2} \leq q \leq p$, *alors si nous désignons par* O_p *le nombre d'opérations effectuées de l'algorithme 3.3.1 sur* p *processeurs,* O_p *vérifie*

$$O_p = \left\lceil \frac{2np}{p+1} \right\rceil + 1$$

Démonstration. Dans la preuve du théorème 3.3.4 on a montré que si $q \neq 0$, $O_p = 2s+2(q-1)$, donc pour $\frac{p+3}{2} \leq q \leq p$, $O_p = 2s + 2(q-1)$. On a :
si $\frac{p+3}{2} \leq q \leq p$, on a $1 \leq 2\frac{q-1}{p+1} < 2$, d'où $\left\lceil \frac{2np}{p+1} \right\rceil = 2s + 2(q-1) - 1 = O_p - 1$, d'où $O_p = \left\lceil \frac{2np}{p+1} \right\rceil + 1$. ❑

	n	Temps	nombre d'opérations	Nombre d'étapes
Algorithme proposé	$n \geq p(p+1) - 1$	$\left\lceil \frac{2n}{p+1} \right\rceil$ si $q \leq 1$ *et* $q \geq \frac{p+3}{2}$ sinon $\left\lceil \frac{2n}{p+1} \right\rceil + 1$	$\left\lceil \frac{2np}{p+1} \right\rceil$ si $0 \leq q < \frac{p+3}{2}$ sinon $\left\lceil \frac{2np}{p+1} \right\rceil + 1$	$p+1$
Algorithme de Nicolau-Wang	$n \geq \frac{p(p+1)}{2} - 1$	$\left\lceil \frac{2n}{p+1} \right\rceil$	$\left\lceil \frac{2np}{p+1} \right\rceil$	$\left\lceil \frac{2n}{p+1} \right\rceil$
Algorithme statique (cf sec 3.3.2)	$n \geq 0$	$\left\lceil \frac{2n}{p+1} \right\rceil + p - 1$	$\left\lceil \frac{2np}{p+1} \right\rceil + p - 1$	3
Borne inférieure	-	$\left\lceil \frac{2n}{p+1} \right\rceil$	$\left\lceil \frac{2np}{p+1} \right\rceil$	-

TAB. 3.1 – Comparaison de trois algorithmes du calcul parallèle des préfixes sur p processeurs

3.3.3.3 Expérimentations

Nous avons implanté en SWARM [5] trois algorithmes parallèles du calcul des préfixes qui sont :

- implantation "statique-proposé" : l'implantation de l'algorithme statique proposé.
- implantation "Nicolau-Wang" : l'implantation de l'algorithme de Nicoalu-Wang [94].
- implantation "statique" : l'implantation de l'algorithme statique présenté dans la section 3.3.2.

Les expérimentations ont été faites sur la machine *idkoiff*. Cette machine est composée de huit processeurs Dual Core AMD Opteron 875, soit 16 coeurs au total à 2,2Ghz, dispose de huit bancs de 4Go de mémoire. Sur cette machine, nous avons utilisé le noyau Lunix 2.6.23, le compilateur gcc 4.2.3 et l'option de compilation -02.

Nous avons mené trois types d'expériences qui sont :

- **Expériences de type 1** : la taille du problème en entrée est petite (entre 20 et 500), et le temps d'une opération est assez élevé (entre une milliseconde et une seconde).
- **Expériences de type 2** : la taille du problème en entrée est assez grande (entre 10^4 et 10^6), et le temps d'une opération est assez élevé (entre une milliseconde et 10 millisecondes).
- **Expériences de type 3** : le temps d'une opération est très petit, de l'ordre de la nano-seconde (addition de deux entiers machine (int) ou de deux réels machine (double) par exemple).

Nous étions seuls sur la machine, et chaque implantation a été lancée 10 fois. Tous les programmes ont été compilés avec le même compilateur g++ 4.2.3 (avec l'option -O2).

1) Expériences de type 1 : n petit, temps d'opération $\star$ grand.
La figure 3.3 compare les accélérations obtenues par les trois implantations ("statique-proposé", "Nicolau-Wang", "statique") à l'accélération théorique optimale $(p+1)$ sur 4 processeurs. Nous observons que les accélérations obtenues par les implantations "statique-proposé" et "Nicolau-Wang" sont très proches de l'optimal, et que les accélérations obtenues par l'implantation "statique" sont un peu loin. Les accélérations de l'implantation "statique" sont moins bonnes car l'algorithme utilisé dans cette implantation est à $p-2$ de l'optimal. Les accélérations obtenues par les implantations "statique-proposé", "Nicolau-Wang" sont très bonnes car le surcoût de synchronisations est négligeable devant le surcoût de calcul.

2) Expériences de type 2 : n grand, temps d'opération $\star$ grand.
La figure 3.4 compare les accélérations obtenues par les trois implantations ("statique-proposé", "Nicolau-Wang", "statique") à l'accélération théorique optimale $(p+1)$ sur 4 processeurs. Nous observons que les accélérations obtenues par les implantations "statique-proposé" et "statique" sont très proches de l'optimal, et que les accélérations obtenues par l'implantation "Nicolau-Wang" sont un peu loin. Nous pensons que la mauvaise performance de l'implantation "Nicolau-Wang" est due au surcoût de synchronisations qui est de l'ordre $\left\lceil \frac{2n}{p+1} \right\rceil$. Les bonnes performances obtenues par l'implantation "statique" sont dues au fait que $p-2$ devient négligeable de-

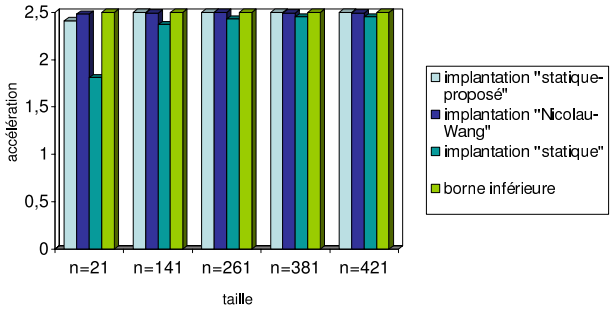

FIG. 3.3 – Comparaison des accélérations des trois implantations sur 4 processeurs pour n petit et temps de l'opération $\star$ grand

vant $\frac{2n}{p+1}$ quand n devient grand. Les bonnes performances obtenues par l'implantation "statique-proposé" sont dues au fait que le surcoût de synchronisations $p+1$ devient négligeable devant $\frac{2n}{p+1}$ quand n devient grand.

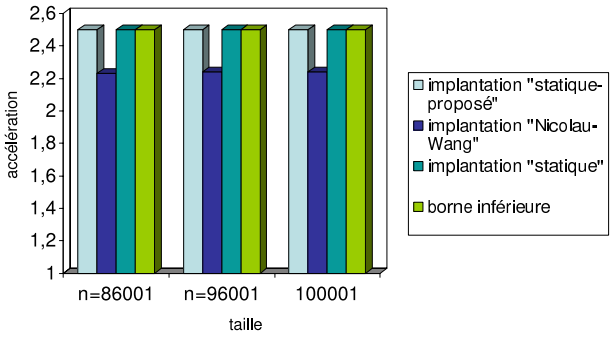

FIG. 3.4 – Comparaison des accélérations des trois implantations sur 4 processeurs pour n grand et temps de l'opération $\star$ grand

3) Expériences de type 3 : n grand, temps d'opération $\star$ petit.
La figure 3.5 donne les temps d'exécution sur 1 processeur par les implantations "statique-proposé", "Nicolau-Wang", "statique" et "séquentielle" en C. Nous observons que, les temps obtenus par l'implantation "statique-proposé" sont proches de ceux de l'implantation "séquentielle", les temps de l'implantation "statique" sont les mêmes que ceux de l'implantation "séquentielle" en C. Sur la figure des grandes différences sont remarquables, ce sont les différences entre les temps obtenus par l'implantation "Nicolau-Wang" et les trois implantations, qui sont environ 3 fois plus rapides que l'implantation "Nicolau-Wang". Cette mauvaise performance de l'implantation "Nicolau-Wang" est due aux surcoûts des barrières de synchronisation dans l'implantation, qui ne sont plus négligeables devant le surcoût de calcul.

Le tableau 3.2 compare les temps obtenus sur 4 processeurs de l'implantation "statique-proposé" et de l'implantation "Nicolau-Wang". Nous avons pris $n = 2.10^8$ pour l'implantation "statique-proposé", mais nous avons pris $n = 2.10^7$ pour l'implantation "Nicolau-Wang" car en prenant $n = 2.10^8$ l'exécution prend 50mn environ. Nous pensons que cette très mauvaise

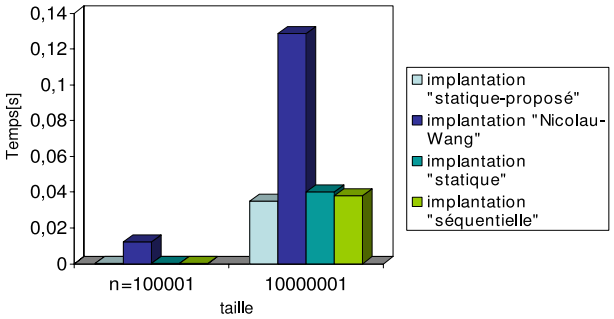

FIG. 3.5 – Comparaison du séquentiel pur aux temps sur un processeur des trois implantations pour n grand et temps de l'opération $\star$ petit

performance est due aux surcoûts de synchronisation dans l'algorithme et aussi au nombre de défauts de cache.

	l'implantation "statique-proposé" $n = 210^8$	l'implantation "Nicolau-Wang" $n = 210^7$
Minimum	0,3163	298,607
Maximum	0,3238	306,169
Moyenne	0,3236	302,125

TAB. 3.2 – Comparaison des temps d'exécution de l'implantation "statique-proposé" et celui de l'implantation "Nicolau-Wang" sur 4 processeurs avec n grand sur la machine AMD opteron

La figure 3.6 compare l'accélération obtenue par l'implantation "statique-proposé" et l'accélération théorique optimale de 1 à 16 processeurs pour $n = 10^4$.

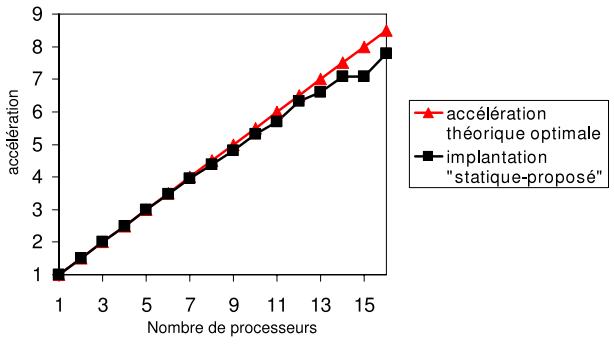

FIG. 3.6 – L'accélération en fonction du nombre de processeurs de l'implantation "statique-proposé" pour $n = 10^4$ et $\tau_{op} = 1,58ms$

3.3.3.4 Conclusion

Nous avons proposé un nouvel algorithme statique optimal du calcul des préfixes. Nous avons montré l'optimalité de cet algorithme par des analyses théoriques. Nous avons validé ces résultats théoriques par les expérimentations menées sur une machine SMP à 16 processeurs. Nous avons constater que notre algorithme statique optimal se comporte mieux en pratique sur des machines multicœurs que l'algorithme optimal de Nicolau et Wang [94]. Mais cet algorithme n'est efficace que sur des processeurs identiques. Il n'est pas adapté dans le cas des processeurs différents et à vitesse variable car il effectue toujours le même nombre ($\left\lceil \frac{2np}{p+1} \right\rceil$) même si tous les processeurs initialement prévus à l'exécution ne sont pas disponibles. Sur des processeurs différents, le temps d'exécution de cet algorithme est le temps d'exécution du processeur le plus lent qui est à un facteur de $\frac{p.\Pi_{ave}+\Pi_{max}}{(p+1)\Pi_{min}}$ de l'optimal. Pour rendre adaptatif le calcul des préfixes, nous proposons dans le chapitre 4 un algorithme adaptatif.

3.3.4 Un algorithme récursif

Le troisième algorithme que nous allons montrer est l'algorithme de Ladner et Fischer [58]. Cet algorithme permet de monter l'impact du coût du parallélisme.

L'algorithme parallèle de Ladner et Fischer est basé sur une approche "Diviser Pour Régner", donc pour simplifier les calculs nous supposons que $n = 2^k$ où k est un entier positif. Il est construit récursivement à partir des étapes suivantes :

1. **Etape 1** : on résout le problème en calculant en parallèle les produits des entrées groupées par 2. On calcule donc les quantités β_i :
   $\beta_0 = x_0 \star x_1, \beta_1 = x_2 \star x_3, \ldots, \beta_{\frac{n}{2}-1} = x_{n-2} \star x_{n-1}$
   et on est ramené au problème réduit du calcul des préfixes des $\frac{n}{2}$ valeurs β_i.
2. **Etape 2** : on résout le problème réduit en calculant (par un appel récursif des 3 étapes décrites ici, jusqu'à obtenir un problème à deux entrées) les préfixes des $\frac{n}{2}$ valeurs β_i : $(\beta_0), (\beta_0 \star \beta_1), \ldots, (\beta_0 \star \beta_1 \ldots \star \beta_{\frac{n}{2}-1})$. Ainsi, on a calculé tous les π_{2k+1} pour $k = 0, \ldots, \frac{n}{2} - 1$.
3. **Etape 3** : à partir de la solution du problème réduit, on construit la solution du problème initial. Les résultats qui manquent sont tous ceux ayant un indice pair : ils peuvent être obtenus en calculant : $x_0, (\pi_1 \star x_2), (\pi_3 \star x_4), \ldots, (\pi_{2k-1} \star x_{2k}), \ldots$ en parallèle.

Une illustration de cet algorithme est décrite sur la figure 3.7.

Le nombre total d'opérations exécutées par cet algorithme est donné par la récurrence suivante : en supposant que $W(2) = 1$, on aura $W(n) = W(\frac{n}{2}) + \frac{n}{2} + \frac{n}{2} = 2n - 3$ et la profondeur en nombre d'opérations $\star$ (ou chemin critique) est calculée par la récurrence $D(2) = 1$ et $D(n) = D(\frac{n}{2}) + 2 = 2 \log n$.

On constante que cet algorithme fait deux fois plus d'opérations que l'algorithme séquentiel

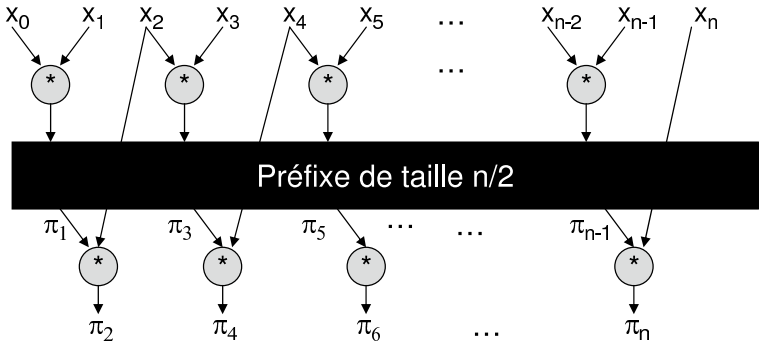

FIG. 3.7 – Algorithme parallèle récursif de Ladner et Fischer

quelque soit le nombre de processeurs utilisés. Cet algorithme est à grain fin, indépendant du nombre de processeurs effectivement disponibles ; il peut être émulé sur p processeurs identiques en temps asymptotique $\frac{2n}{p} + O(\log n)$ pour $p < \frac{n}{\log n}$ mais effectue $2n$ opérations, et est à un facteur de $\frac{p+1}{p}$ de l'optimal. En plus du surcoût en nombre d'opérations, cet algorithme à grain fin génère un surcoût en terme de créations de tâches qui est de l'ordre de $2n\tau_{tche}$ où τ_{tche} est le surcoût de création d'une tâche. Donc le temps d'exécution de cet algorithme est dégradé par ces surcoûts en nombre d'opérations et de créations de tâches, et pour atteindre une performance optimale tout en conservant un nombre optimal d'opérations et un temps d'exécution meilleur, il est nécessaire de recourir à un algorithme adaptatif.

Remarque : En prenant en compte le surcoût de la découpe récursive, la profondeur devient $D(n) = D(\frac{n}{2}) + \Theta(\log n) = \Theta\left(\log^2 n\right)$. Mais une découpe en $\sqrt{n}$ permet d'obtenir un algorithme parallèle de profondeur $\Theta(\log n)$.

3.4 Les techniques d'adaptivité existantes

Plusieurs techniques d'adaptation existent, mais dans des contextes différents en fonction des problèmes étudiés ou des architectures utilisées [1, 30, 10, 82, 28, 86, 53, 48]. Les techniques de tolérances aux fautes [48, 53] étudient le plus souvent l'adaptativité des applications parallèles sur des architectures dynamiques (les ressources peuvent apparaître et disparaître à tout moment de l'exécution). Samir et al [48] proposent un mécanisme basé sur les applications de type graphe de flot de données d'une exécution parallèle dans les environnements dynamiques et hétérogènes. Dans [23] les auteurs donnent une classification élargie des différentes techniques d'adaptation. Dans cette section, nous nous intéressons à l'adaptation au niveau algorithmique, et nous supposons que l'environnement est tolérant aux fautes. Nous présentons deux techniques non exhaustives qui sont les plus couramment utilisées dans l'adaptation algorithmique. Elles sont basées sur des sélections et combinaisons d'algorithmes, et sur l'ordonnancement dynamique par vol de travail.

3.4.1 Les techniques par sélection et combinaison d'algorithmes

Plusieurs algorithmes peuvent exister pour la résolution d'un problème, chacun ayant ses contraintes (*e.g.* type de données, environnement d'exécution) pour le résoudre efficacement. Pour résoudre le problème quelque soit les types de données ou l'environnement d'exécution, des techniques par sélection et combinaison d'algorithmes ont été proposées [28, 67, 24, 72]. Un algorithme basé sur ces techniques est appelé *poly-algorithme* ou algorithme *hybride* [23, 66].

Définition 3.4.1. *[23] Un algorithme est un poly-algorithme quand sa stratégie de décision est basée sur le choix entre aux moins deux algorithmes résolvant le même problème.*

Le choix fait par la stratégie de décision peut se faire au cours de l'exécution, dans ce cas le poly-algorithme est dit "dynamique" ou avant l'exécution, dans ce cas le poly-algorithme est dit "statique" . Lorsque le choix est unique et ne varie plus pendant l'exécution, la stratégie de décision sélectionne un seul algorithme parmi au moins deux algorithmes parallèles, on parle dans ce cas de *sélection* d'algorithme. Lorsque le choix est fait en combinant les solutions d'au moins deux algorithmes pour la résolution du problème, on parle de *combinaison* d'algorithmes. Les techniques de sélection d'algorithmes sont utilisées généralement quand l'objectif de performance est unicritère (*e.g.* minimiser le temps d'exécution seulement), et quant aux techniques de combinaisons d'algorithmes, elles sont utilisées quand l'objectif de performance est multi-critère (*e.g.* minimiser le temps d'exécution final et le nombre total d'opérations à réaliser).

Les techniques de sélection d'algorithmes ont été étudiées pour la première fois en 1976 par John Rice [72]. La bibliothèque STAPL (Standard Template Adaptive Parallel Library) [98] par exemple utilise les techniques de sélection d'algorithmes pour la parallélisation adaptative de certains algorithmes de la Librairie Standard C++ [65, 87]. L'algorithme de tri est un exemple intéressant pour illustrer cette technique : il existe plusieurs algorithmes pour trier qui ont des complexités différentes selon le contenu de la séquence à trier, la taille de la séquence, le type des données, etc. Par exemple si la séquence ne contient que des entiers, le tri radix (*radix sort*) peut être le meilleur choix pour avoir un temps d'exécution minimal.

Les techniques de combinaison d'algorithmes sont aussi très utilisées pour résoudre d'une manière adaptative certains problèmes. Les objectifs de ces techniques étant généralement multi-critères, l'idée est donc de faire collaborer plusieurs algorithmes résolvant le même problème, mais optimisant chacun un critère de performance spécifique. Par exemple, si on veut optimiser le temps d'exécution T_p obtenu par le théorème 3.4.1 d'un programme parallèle ordonnancé par vol de travail, l'objectif est à la fois de minimiser le travail W et la profondeur D. Pour atteindre cet objectif, l'approche générale est de coupler un algorithme séquentiel optimal qui minimise le travail W et un algorithme parallèle à grain fin qui minimise la profondeur D. Il existe deux approches qui permettent de réaliser ce couplage : l'une est *récursive* et l'autre est en *cascade*. Les deux approches sont détaillées ci-dessous :

– L'approche récursive consiste à exécuter un algorithme parallèle jusqu'à un certain grain, puis à exécuter l'algorithme séquentiel. L'inconvénient de cette technique est que pour

avoir un travail optimal il faut augmenter le grain, ce qui augmente la profondeur D donc aussi le nombre de vols. Par exemple pour le calcul récursif des préfixes (algorithme de Ladner et Fischer dans la section 3.3.4) si le grain est égal à $\frac{n}{2}$, alors $W = \frac{3n}{2}$ et $D = n$. Puisque le grain est fixé statiquement, il faut une autre technique qui permet d'adapter le grain automatiquement

– L'approche en cascade [67] rend le travail W de l'algorithme parallèle proche du travail séquentiel W_{seq} et consiste à exécuter d'abord l'algorithme séquentiel optimal pour réduire le nombre d'opérations, puis à utiliser l'algorithme parallèle pour réduire la profondeur D. L'inconvénient de cette technique est l'utilisation d'un algorithme séquentiel à gros grain et d'un algorithme parallèle à grain fin. Aussi la cascade est décidée statiquement en fonction du nombre de processeurs ou en fonction du grain. L'algorithme statique du calcul des préfixes présenté dans la section 3.3.2 illustre ce cas avec un grain égal à $\frac{n+1}{p+1}$. Donc, on peut faire la même observation que l'approche récursive concernant l'adaptation du grain.

On peut constater que dans ces techniques (combinaison et sélection) pour avoir une adaptativité totale, il est nécessaire d'adapter la granularité de l'algorithme choisi (nous rappelons qu'un grain est un bloc de calculs (resp. données) qui est évalué localement de manière séquentielle sur un seul processeur). Un gros grain n'est pas adapté à des processeurs de vitesses différentes, ou à vitesses variables, ou sur des données différentes. Un grain fin peut engendrer des surcoûts de parallélisme (gestion de la pile contenant les appels récursifs, création de tâches). Pour réaliser cette adaptation, nous devons recourir à une technique qui permet de prendre en compte l'adaptation de cette granularité.

3.4.2 Ordonnancement par vol de travail : Dégénérescence séquentielle

3.4.2.1 Ordonnancement par vol de travail

L'ordonnancement par vol de travail (*work-stealing* en anglais) [3, 18, 9] est une technique qui peut permettre d'équilibrer efficacement les charges d'un programme. Il est adapté par exemple pour les algorithmes récursifs (*e.g.* tri).

De manière générale, un algorithme par vol de travail suit le principe glouton (*greedy schedule*) : à tout instant où il existe une tâche prête mais non encore ordonnancée, tous les processeurs sont actifs. Parmi les ordonnancements gloutons figurent les ordonnancements de liste : les tâches non encore ordonnancées sont stockées dans une liste ; lorsqu'un processeur devient inactif, il va chercher une tâche prête dans cette liste si celle-ci est non vide et sinon il s'ajoutent à la liste des processeurs inactifs.

Les ordonnancements par vol de travail suivent ce principe, mais sont de plus distribués : Chaque processeur gère localement la liste des tâches que lui-même a créées dans une pile. La gestion locale de cette pile suit l'ordre séquentiel : la tâche qui est dépilée est toujours celle qui serait exécutée en premier parmi les tâches dans la pile, dans un ordonnancement séquentiel.

Dans la suite, nous considérons un ordre séquentiel de type profondeur d'abord : la tâche la plus prioritaire est alors toujours en dernière position de la pile. Chaque fois qu'un processeur crée une tâche, il l'empile dans sa pile en dernière position. Lorsqu'un processeur termine une tâche, deux cas sont distingués :

- soit sa pile locale contient des tâches prêtes : dans ce cas il prend la plus récente parmi celle-ci et démarre localement son exécution ;
- soit sa pile locale est vide ou ne contient pas de tâches prêtes, auquel cas il passe à l'état "voleur" et cherche à récupérer du travail sur les autres processeurs. Cet état reste inchangé tant qu'il n'a pas trouvé un autre processeur, appelé "victime" possédant au moins une tâche prête. Dans ce cas, il vole au processeur victime sa tâche prête la plus ancienne.

Le choix du processeur victime peut être fait de différentes manières. Dans la suite, nous considérons que ce choix est fait de manière aléatoire (*random workstealing*) et uniforme. L'intérêt de ce choix est qu'il peut être implanté efficacement en distribué sans nécessiter de synchronisation, tout en garantissant une performance quasi optimale avec une grande probabilité.

Le théorème suivant donne une borne sur le temps d'exécution de l'algorithme parallèle de travail W et de chemin critique D ordonnancé par vol de travail (pour les notations voir la section 2.3). Nous rappelons que T_p est la durée d'exécution de l'algorithme sur p processeurs.

Théorème 3.4.1. *[3, 18] Sur une machine avec p processeurs identiques, avec une grande probabilité*[2] *, le nombre de vols réussis est majoré par $O(pD)$ et le temps d'exécution T_p est majoré par*

$$T_p \leq \frac{W}{p} + O(D)$$

Ce théorème est étendu dans [8] au cas de processeurs hétérogènes ou de vitesses différentes, non connues ou variables. Au niveau du vol de travail, la seule modification est lorsque p_v vole du travail à un processeur actif p_w : si p_w est en cours d'exécution mais n'a pas de travail prêt à être volé et si p_v est plus de β fois plus rapide que p_w (par exemple au moins deux fois plus rapide si $\beta = 2$) alors la tâche en cours d'exécution sur p_w est préemptée et migrée sur p_v. Le temps d'exécution T_p est alors donné par le théorème suivant :

Théorème 3.4.2. *[8] Avec une grande probabilité, le nombre de vols réussis est majoré par $O(pD)$ et le temps d'exécution T_p est majoré par*

$$T_p \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{\beta.D}{\Pi_{ave}}\right)$$

.

Ainsi, nous pouvons remarquer que si $D \ll W$ et $W \simeq W_{seq}$, l'espérance du temps est proche de l'optimal. Mais le travail W contient aussi les surcoûts liés au parallélisme (créations de tâches, copie de données, etc.). Et donc pour obtenir un ordonnancement efficace, ce surcoût doit être réduit le plus possible. Pour limiter ce surcoût d'ordonnancement, les techniques

[2] i.e. pour tout $c > 0$ et n assez grand, la probabilité est supérieure à $1 - n^{-c}$.

mises en œuvres [18, 9, 71] consistent à privilégier, dans le cas où tous les processeurs sont actifs, l'exécution séquentielle efficace de l'algorithme parallèle. Un nouveau processus n'est effectivement créé pour l'exécution d'une tâche prête que lorsqu'un processeur devient inactif et effectue une opération de vol : on parle de *dégénérescence séquentielle* [76]. Nous allons détailler dans la section suivante cette technique.

3.4.2.2 Dégénérescence séquentielle

La technique de dégénérescence séquentielle (aussi appelé *Work First Principle*) [35, 76, 71] consiste à minimiser le surcoût de création de tâches parallèles de l'ordonnancement par vol de travail. Le principe est basé sur le fait qu'une application parallèle est composée d'un nombre de tâches très largement supérieur au nombre de processeurs. L'idée est donc de privilégier une exécution séquentielle efficace de nombreux groupes de tâches sur un processeur sans nécessiter de créations de tâches inutiles, et de tolérer un surcoût plus important lors du vol par un processeur inactif. Pour exécuter les groupes de tâches efficacement sur un processeur, les appels de création de tâches immédiatement prêtes (cas des programmes série-parallèle, fork/join) sont traduits en appel local de fonction séquentielle. Ainsi dans un ordonnancement par vol utilisant la dégénérescence le nombre de créations est égal au nombre de vols réussis.

Dans le cas des programmes série-parallèles, le nombre de vols par processeur est au plus D d'après le théorème 3.4.1 ; ainsi, le surcoût dû au parallélisme est borné par $pD\tau_{tche}$ où τ_{tche} est le surcoût de création d'une tâche. Par exemple le surcoût du parallélisme de l'algorithme Ladner-Fischer à grain fin du calcul parallèle des préfixes est de $(2n-3)\tau_{tche}$ et son travail $W = W_{seq} + (2n-3)\tau_{tche}$ par un ordonnancement par vol de travail sans dégénérescence ; et avec dégénérescence séquentielle le surcoût est de $\Theta\left(p\log^2 n\right)$ et son travail $W = W_{seq} + \Theta\left(p\log^2 n\right)$.

Cette technique diminue le surcoût d'ordonnancement de l'algorithme parallèle, mais ne s'attaque pas au surcoût arithmétique lié à la parallélisation, qui peut s'avérer très coûteux en comparaison avec un algorithme séquentiel optimisé. En effet, c'est l'algorithme parallèle qui est exécuté, sauf que si tous les processeurs sont actifs les créations de tâches sont traduites en appel de fonction. La technique n'est efficace que, si la sérialisation de l'algorithme parallèle est un algorithme séquentiel efficace. Pourtant, cette hypothèse est très restrictive, car beaucoup d'algorithmes parallèles (*e.g.* préfixe parallèle) font plus d'opérations qu'un algorithme séquentiel optimal.

3.5 Adaptivité dans les bibliothèques existantes

Plusieurs bibliothèques implémentent des algorithmes adaptatifs, dans cette section, nous nous intéressons à deux bibliothèques : Intel TBB et Athapascan/kaapi (présentées au chapitre 2). Nous nous intéressons à Intel TBB car elle est une bibliothèque récente, et aussi nous allons

comparer nos implémentations adaptatives à celles de TBB. Nous nous intéressons à Athapascan/kaapi car le moteur exécutif de notre interface adaptative utilise Athapascan/kaapi.

3.5.1 Intel TBB

TBB auto_partitioner

TBB auto_partitioner permet d'adapter dynamiquement le choix de la granularité en fonction de l'activité des processeurs, ce qui limite le nombre de vols de l'algorithme d'ordonnancement par vol de travail. Le principe de ce mécanisme dans TBB est le suivant :
Le bloc de calculs (*range TBB*) est d'abord divisé en S_1 sous-blocs, où S_1 est proportionnel au nombre de processus (*threads*) créés par l'ordonnanceur. Ces sous-blocs sont exécutés séquentiellement par les processus créés sans être découpés sauf lors des vols. Si un sous-bloc est victime d'un vol par un processus inactif, TBB auto_partitioner le divise en deux sous-blocs petits pour créer des sous-blocs additionnels.

L'ordonnanceur TBB utilise l'ordonnancement par vol de travail selon le principe "Travail d'abord" (*Work-first principle*) similaire à celui de cilk.

3.5.2 Athapascan/kaapi

La bibliothèque Athapascan/kaapi [37, 38] permet d'adapter automatiquement la charge d'un programme en fonction de l'activité des processeurs grâce à l'utilisation de l'ordonnancement par vol de travail. Elle permet aussi de minimiser les surcoûts de créations des tâches lors de l'exécution d'un programme parallèle grâce l'utilisation de la technique de dégénérescence séquentielle [71]. Grâce à l'utilisation de ses protocoles de tolérance aux pannes Athapascan/kaapi [48, 46] s'adapte automatiquement à la dynamicité des ressources (elles peuvent disparaître et apparaître à tout moment). Athapascan/kaapi est une bibliothèque qui permet le développement des programmes efficaces pour les architectures à mémoire partagée comme distribuée.

3.6 Conclusion

Nous avons présenté dans ce chapitre dans un premier temps la définition générale d'un algorithme adaptatif et nous avons classifié les algorithmes adaptatifs en deux grandes classes : des algorithmes qui dépendent des ressources de l'architecture cible et des algorithmes qui sont indépendants des ressources. Puis dans une deuxième partie nous avons présenté les principaux besoins d'algorithmes adaptatifs dans la programmation parallèle. Nous avons vu que pour pouvoir bénéficier pleinement de la puissance de calcul fournie par les architectures dont le nombre ou la vitesse des processeurs peuvent varier il est nécessaire de faire recours à un algorithme

adaptatif. Nous avons vu que paralléliser un programme peut introduire des surcoûts (arithmétiques, copie, création de tâches) qui peuvent dégrader la performance du programme parallèle. Pour illustrer tous ces besoins, nous avons utilisé le calcul parallèle des préfixes comme un cas d'étude. Nous avons d'abord donné une borne inférieure du temps de ce calcul sur des processeurs à vitesse variable, puis nous avons présenté deux algorithmes parallèles pour ce calcul qui ont permis de mettre en évidence les besoins d'algorithmes parallèles adaptatifs. Ensuite dans la troisième partie nous avons présenté les techniques existantes permettant d'adapter les algorithmes parallèles. Nous avons montré les limites de ces techniques. Notre objectif est de développer des algorithmes parallèles qui atteignent des performances optimales sans faire aucune connaissance du nombre de processeurs de l'architecture cible (*Processor oblivious algorithms* en anglais), c'est à dire que ces algorithmes doivent s'adapter automatiquement et dynamiquement au processeurs disponibles. Dans le chapitre suivant, nous allons proposer un algorithme adaptatif indépendant du nombre de processeurs pour le calcul parallèle des préfixes.

Chapitre 4

Un algorithme adaptatif pour le calcul parallèle des préfixes

Le calcul des préfixes est une opération fondamentale que l'on retrouve dans de nombreuses applications importantes, comme par exemple dans les domaines du calcul matriciel, du traitement de l'image, de la reconnaissance de langages réguliers. Dans ce chapitre, nous donnons un nouvel algorithme parallèle de calcul des préfixes pour des processeurs dont la vitesse ou le nombre peut varier en cours d'exécution. Basé sur le couplage récursif d'un algorithme séquentiel optimal et d'un algorithme parallèle non optimal mais récursif à grain fin, il exploite un ordonnancement dynamique de type vol de travail. Sa performance théorique est analysée sur p processeurs à vitesses variables, de vitesse moyenne Π_{ave}. Bien que cet algorithme est indépendant du nombre de processeurs, son temps d'exécution est équivalent à $\frac{2T_{seq}}{\Pi_{ave}(p+1)}$, ce qui est optimal si les processeurs sont identiques (*e.g.* $\Pi_{ave} = 1$). Expérimentalement, cet algorithme adaptatif est comparé à un algorithme optimal pour un nombre fixé p de processeurs identiques avec ordonnancement statique optimal sur deux machines SMP à 8 et 16 cœurs. Ses performances sont analogues dans le cas où les processeurs sont dédiés au calcul, et il est beaucoup plus rapide dans le cas général où la machine exécute concurremment d'autres processus (cas multi-utilisateur).

4.1 État de l'art

Étant donnés $x_0, x_1, \ldots x_n$, les n préfixes π_k, pour $1 \leq k \leq n$, sont définis par $\pi_k = x_0 \star x_1 \star \ldots \star x_k$ où $\star$ est une loi associative.

Le calcul des préfixes est une opération qui apparaît dans de nombreux algorithmes notamment l'évaluation de polynômes et les additions modulaires [27, 54, 57, 92, 58], le packing, la parallélisation des boucles [62] et figure dans MPI sous le nom de *scan* [81]. Plusieurs applications du calcul des préfixes peuvent être trouvées dans [15, 17].

Le calcul séquentiel itératif peut être effectué par une boucle simple :

$\pi[0] = x[0]$
$Pour\ i = 1$ à n
  $\pi[i] = \pi[i-1] \star x[i]$

Ce calcul séquentiel des préfixes requiert n opérations $\star$. Nous pouvons diviser le calcul parallèle des préfixes en deux catégories : sur un nombre non borné de ressources (processeurs par exemple) ou généralement le nombre de ressources est en $O(n)$ dépendant de la taille n du problème, et sur un nombre p fixé de processeurs où p est indépendant de la taille du problème, mais l'algorithme dépendand du nombre p de processeurs.

Plusieurs auteurs [58, 33, 60, 61, 99] ont étudié le calcul des préfixes sur un nombre non borné de ressources avec une profondeur petite ($O(\log n)$). Dans [58] Ladner et Fischer proposent un algorithme parallèle basé sur une découpe récursive de temps $2\log n$ avec $2n$ opérations pour $n = 2^k$ avec $k > 0$. Fich [33] donne une borne inférieure de la taille en sortie de tout circuit de calcul parallèle des préfixes de n entrées et de profondeur $\log n + k$ avec $0 \leq k \leq \log n$. Fich montre que pour $n = 2^m$, cette borne est de $(2 + \frac{2^{1-k}}{3})2^m - O(m^2)$ pour $1 \leq k \leq m-1$, et qu'elle est de $\frac{10}{3}2^m - O(m^3)$ pour $k = 0$ et $m \geq 0$. On peut alors en déduire que le nombre minimal d'opérations par tout algorithme parallèle du calcul des préfixes de profondeur $\log n$ est au moins $\frac{10}{3}n$, et pour ceux de profondeur $2\log n$ est $2n$, soit plus de deux à trois fois plus élevé que celui de l'algorithme séquentiel. Donc les algorithmes proposés par ces auteurs font au moins deux fois plus d'opérations que l'algorithme séquentiel. Ces algorithmes proposés peuvent être émulés sur p processeurs identiques, mais ne seront pas optimaux en nombre d'opérations. Par exemple l'algorithme de Ladner et Fischer [58] peut être émulé sur p processeurs identiques en temps asymptotique $\frac{2n}{p} + O(\log n)$ pour $p < \frac{n}{\log n}$, mais effectue $2n$ opérations donc n'est pas optimal : comme nous l'avons vu au chapitre 3.

D'autres travaux ont étudié le calcul des préfixes sur un nombre borné de ressources (processeurs) [94, 85, 56, 32, 43]. Kruskal [56] propose un algorithme pour le calcul parallèle des préfixes sur p processeurs de temps $\frac{2n}{p} + \log p$. Cet algorithme prend $\frac{2n}{p(p+1)} + \log p$ plus d'étapes que le temps optimal, donc est loin de l'optimal pour $p \ll n$. Snir [85] donne un algorithme du calcul parallèle des préfixes avec un nombre fixé de processeurs pour les machines à lecteurs concurrentes et écritures exclusives (CREW) telle que la profondeur du circuit résultant pour p processeurs est $\frac{2n}{p+1} + O(1)$, qui est très proche du temps optimal. L'algorithme de Snir est basé est une découpe en $p+1$ blocs de tailles différentes. Pour un problème donné de taille n, l'algorithme de Snir calcule la taille de chaque bloc donnant une partition correcte du problème permettant de minimiser la profondeur de l'ordonnancement résultant et une nouvelle partition est faite à chaque nouvelle taille donnée. Le surcoût dû au calcul du partitionnement correct a une influence sur le temps d'exécution de l'algorithme de Snir. Egecioglu et koc [32] ont étudié le calcul parallèle des préfixes sur un petit nombre de processeurs. Ils montrent que le temps d'exécution de leur algorithme est de $\frac{p(p-1)n}{p(p+1)+2} + \frac{1}{2}p(p-1)$ et son nombre d'opérations effectuées est de $\frac{2(p+1)n}{p(p+1)+2} - 1$. Leur algorithme est loin de l'optimal si p devient grand. Sur p processeurs (p indépendant de n) pour $n > \frac{p(p+1)}{2}$ Nicolau et Wang [94] construisent un algorithme optimal de temps(étapes) $\lceil \frac{2n}{p+1} \rceil$ basé sur une technique qu'ils appellent ordonnancement harmonique ("Harmonic Schedules"). Cette technique permet en fait de minimiser le nombre

total d'opérations finales de préfixes effectuées et de maximiser l'utilisation des processeurs. Le nom harmonique dans l'ordonnancement vient au fait du compromis entre la minimisation du nombre total d'opérations effectuées et l'utilisation maximum des processeurs. Dans cet algorithme le surcoût de synchronisation entre les processeurs est de $O(\frac{2n}{p+1})$. Donc si on prend en compte le surcoût de synchronisation, le temps d'exécution total de leur algorithme est de $\lceil \frac{2n\tau_{op}}{p+1} \rceil + \lceil \frac{2n\tau_{sync}}{p+1} \rceil$ qui n' est optimal que si $\tau_{op} \ll \tau_{sync}$ où τ_{op} est le temps d'une opération $\star$ et τ_{sync} le temps d'une synchronisation (barrière de synchronisation).

Ces différents algorithmes proposés dépendent tous du nombre p de processeurs et d'un ordonnancement statique. L'inconvénient d'un algorithme optimal pour p processeurs fixé est que le nombre d'opérations est au moins $2\frac{p}{p+1}$ fois supérieur au nombre n d'opérations de l'algorithme séquentiel. De plus ces algorithmes, bien qu'optimaux sur p processeurs, ne sont pas performants si l'on dispose d'une machine avec des processeurs différents, ou d'une machine utilisée par plusieurs utilisateurs car la charge des processeurs varie au cours de l'exécution. Pour traiter ce problème, nous proposons dans ce chapitre un nouvel algorithme parallèle adaptatif de calcul de préfixes, dit à grain adaptatif. La section 4.2 présente le nouvel algorithme à grain adaptatif et analyse sa complexité dans le cas de processeurs identiques et de vitesses variables. Dans la section 4.3, nous présentons des comparaisons expérimentales entre cet algorithme et un algorithme optimal sur deux machines à 8 et 16 processeurs, dans les quatre cas suivants : processeurs dédiés, processeurs perturbés par des processus additionnels, processeurs hétérogènes et processeurs distribués ; conformément à l'analyse théorique, l'algorithme adaptatif est le plus rapide.

4.2 Algorithme parallèle à grain adaptatif

Pour faciliter la compréhension de l'algorithme que nous proposons, nous commençons la présentation dans la section 4.2.1 par le couplage d'un algorithme séquentiel optimal qui est toujours exécuté par un seul processus et d'un algorithme parallèle récursif à grain fin exécuté par les autres processus. Ce couplage permet de maintenir un nombre optimal d'opérations tout en diminuant la profondeur au fur à mesure que le nombre de processeurs augmente.

4.2.1 Couplage de deux algorithmes

Notre algorithme parallèle à grain adaptatif est basé sur le couplage d'une part d'un processus séquentiel P_s qui calcule séquentiellement les préfixes et d'autre part d'un algorithme parallèle récursif à grain fin, qui est ordonnancé par vol de travail par d'autres processus P_v. Chacun des processus est placé sur un processeurs. Nous rappelons que l'algorithme prend en entrée un tableau x indicé de 0 à n et donne en sortie un tableau π indicé aussi de 0 à n. Nous supposons que l'accès à tout élément du tableau est direct (accès aléatoire). Nous désignerons par m le nombre de préfixes finals qui reste encore à calculer (la taille du calcul restant). L'algorithme utilisant le couplage est le suivant :

Initialement, le processus P_s démarre le calcul des préfixes sur l'intervalle $[0, n[$. Chaque processus P_s (resp. P_v) possède sa liste distribuée L_s (resp. L_v) des intervalles qui lui ont été volés, initialement vide.

• **Algorithme séquentiel sur un processus** P_s :

1. P_s calcule séquentiellement les préfixes à partir de l'indice 0 (i.e. π_1), jusqu'à atteindre l'indice n ou un indice u_1 tel que l'intervalle $[u_1, u_2]$ d'indices (intervalle en tête de sa liste L_s si elle n'est pas vide) a été volé par un processus P_v.
2. Si L_s n'est pas vide, P_s dépile l'intervalle $[u_1, u_2]$, insère la liste L_v de P_v à la tête de sa liste L_s.
   – Si le calcul sur cet intervalle $[u_1, u_2]$ n'est pas terminé P_s se synchronise avec le processus P_v ; pour cela il attend éventuellement que P_v ait terminé son calcul en cours d'exécution (préemption faible). Il récupère le dernier indice $k \leq u_2$ calculé par P_v, qui a donc déjà calculé $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \star x_{u_1+1}, \ldots, r_k = r_{k-1} \star x_k$. P_s envoie la valeur π_{u_1-1} à P_v et lui demande de finaliser l'intervalle $[u_1, k[$ (voir l'étape 1 de l'algorithme des processus p_v).
   – Sinon P_s récupère le dernier indice $k = u_2$ calculé par P_v, qui a donc déjà calculé $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \star x_{u_1+1}, \ldots, r_{u_2} = r_{u_2-1} \star x_{u_2}$. P_s prépare une tâche de finalisation qui prend en entrée la valeur $\beta = \pi_{u_1-1}$ et l'intervalle $[u_1, u_2[$. Cette tâche de finalisation sera volée par un processus P_v (voir l'étape 2 de l'algorithme des processus p_v).
   – P_s calcule $\pi_k = \pi_{u_1-1} \star r_k$; puis il reprend le calcul séquentiel des préfixes de $k+1$ à n à partir de $k+1$ en revenant à l'étape 1. On parle d'opération de saut ; pour chaque opération de saut, P_s effectue une opération $\star$.
3. P_s s'arrête lorsqu'il a calculé π_n (sa liste L_s est vide en ce moment). Après avoir calculé π_n, il devient un processus voleur et exécute l'algorithme P_v.

• **Algorithme parallèle sur** $p-1$ **processus** P_v :

1. Lorsqu'il est préempté par P_s (voir algorithme P_s), P_v a déjà calculé localement des préfixes partiels $r_{u_1}, \ldots, r_k$ de l'intervalle $[u_1, k]$. Il reçoit alors la valeur du dernier préfixe $\beta = \pi_{u_1-1}$ calculée par P_s. Il finalise alors l'intervalle $[u_1, k[$ (π_k est calculé par P_s) en calculant les produits $\pi_i = \beta \star r_i$ pour $u_1 \leq i < k$. Ces produits sont parallèles ; sur inactivité d'un autre processus voleur, une moitié de ces calculs restant à faire sur P_v dans cet intervalle peut alors être volée.
2. Lorsqu'il est inactif, le processus P_v choisit au hasard un processus jusqu'à trouver un processus actif P_w. Il peut s'agir soit de P_s, soit d'un autre processus voleur. Si la victime est P_s et si P_s a encore des tâches de finalisation prêtes et non volées, une de ces tâches est d'abord volée en priorité ; sinon P_v vole sur l'intervalle restant à calculer et volable sur P_s.
   (a) P_v découpe l'intervalle restant à calculer et volable sur P_w en deux parties ; il extrait la partie droite $[u_1, u_2[$ de l'intervalle et la vole. La partie gauche reste en cours de calcul sur P_w.
   (b) P_v démarre le calcul sur l'intervalle volé $[u_1, u_2[$. Il peut s'agir : soit d'un calcul de préfixes locaux (i.e. $r_{u_1} = x_{u_1}, r_{u_1+1} = r_{u_1} \star x_{u_1+1}, \ldots$) ; soit de la finalisation de calculs de préfixes à partir de valeurs r_k déja calculées (i.e. $\pi_{u_1} = \beta \star r_{u_1}, \pi_{u_1+1} = \beta \star r_{u_1+1}, \ldots$).

Le programme s'arrête lorsque tous les processeurs sont inactifs. L'intérêt de cet algorithme est qu'un processeur devenu lent sera soit préempté par le processus séquentiel, soit volé par un processus parallèle.

4.2.1.1 Amortissement du surcoût de synchronisations

L'algorithme précédemment présenté a un surcoût dû à la gestion de la synchronisation nécessaire lors des accès à un intervalle $[u_i, u_j[$ volables sur un processus actif par des processus inactifs. En effet, à la fois le processus volé et le processus voleur accèdent à un même intervalle $[u_i, u_j[$ lors du vol (Puisque pendant que le processus volé ait entrain d'extraire un indice dans l'intervalle $[u_i, u_j[$ un processus voleur peut accéder à ce même intervalle $[u_i, u_j[$ pour extraire une partie du travail en cours). Le code de l'algorithme séquentiel doit donc être analysé afin d'identifier les sections critiques de code qui doivent être exécutées en exclusion mutuelle vis-vis des modifications de l'intervalle $[u_i, u_j[$. La réalisation effective de l'exclusion peut reposer sur l'utilisation des primitives générales de synchronisation (verrou, sémaphore, compare_and_swap, ...).

Ce surcoût de maintien de cohérence peut s'avérer important vis-vis des opérations de calcul et masquer le gain en nombre d'opérations arithmétiques grâce à l'utilisation d'un algorithme séquentiel optimisé à la place d'un algorithme parallèle. Dans ce cas, il est alors nécessaire de changer l'algorithme séquentiel afin d'augmenter sa granularité et donc diminuer le surcoût de cohérence : c'est-à-dire augmenter le nombre d'opérations effectuées par accès à l'intervalle de calcul. Nous allons désigner par **extract_seq** l'opération qui permet d'extraire la taille du bloc d'instructions qui sera traité séquentiellement, nous devons alors analyser le choix de cette taille pour masquer le surcoût de synchronisation par rapport au nombre d'opérations effectuées. Si le nombre d'opérations extraites par **extract_seq** est très grand, il y aura perte de parallélisme car la fraction extraite sera exécutée séquentiellement : aucun vol ne peut être effectué sur la fraction extraite donc cela diminue le degré de parallélisme. Si ce nombre est petit, les surcoûts de synchronisations peuvent masquer le gain de calcul. Donc il faut chercher une bonne méthode pour calculer ce nombre. Une façon de calculer ce nombre est déterminer d'abord le temps en nombre d'opérations sur un nombre non borné de processeurs de l'algorithme parallèle sur l'intervalle restant (la profondeur de l'algorithme). Nous rappelons que la profondeur de l'algorithme parallèle est la borne inférieure du temps d'exécution de l'algorithme parallèle sur n'importe quel nombre de processeurs. En choisissant ce nombre comme la taille du bloc d'instructions à extraire par l'opération **extract_seq**, ceci n'augmentera pas la profondeur de l'algorithme. Nous allons désigner par **extract_par** l'opération qui permet d'extraire du travail un processus victime par un processeur voleur. Donc les opérations **extract_seq** et **extract_par** sont synchronisées à l'aide des verrous pour garantir la cohérence des informations extraites.

Pour minimiser ce surcoût de synchronisations, nous avons implémenté les fonctions **extract_seq** et **extract_par** comme suit :

- **extract_seq** : consiste à extraire localement par chaque processus un bloc de taille $\alpha \log m$ sur son intervalle et à faire avancer l'indice courant de l'intervalle de $\alpha \log m$. Supposons que initialement l'intervalle restant avant l'extraction est décrit par $[u_i, u_j[$ où $m = u_j - u_i$,

alors après l'extraction le travail restant sera $[u_i + \alpha \log m, u_j[$. Nous avons choisi $\alpha \log m$ comme taille d'extraction, car pour le calcul parallèle des préfixes, la profondeur sur un intervalle $[u_i, u_j[$ est de $\Omega((\log u_j - u_i))$ [33].

- **extract_par** : consiste à extraire la moitié du travail restant de la victime. Supposons que le travail restant de la victime est décrit par l'intervalle $[u_i, u_j[$, alors l'opération d'extraction consiste à extraire l'intervalle $[u_j - \frac{u_j - u_i}{2}, u_j[$ et à laisser l'intervalle $[u_i, u_j - \frac{u_j - u_i}{2}[$ à la victime. Pour augmenter le degré de parallélisme, l'extraction est effectuée à grain fin (tant que la taille du travail restant est supérieure à 2 on peut toujours extraire la moitié du travail restant). Nous avons choisi ce facteur 2 pour permettre la découpe récursive jusqu'à ce grain le plus fin.

4.2.2 Problème lié à la découpe

Dans le chapitre 3, nous avons rappelé un algorithme statique et proposé un nouvel algorithme optimal du calcul parallèle des préfixes en découpant statiquement la taille initialement du tableau en $p + 1$ blocs de même taille (à un élément près), mais ce découpage n'est pas adapté si l'on dispose des processeurs avec des vitesses variables ou hétérogènes ou le temps d'une opération $\star$ est variable (par exemple le calcul de la multiplication matricielle qui dépend des tailles des matrices à multiplier). Mais construire un algorithme parallèle optimal du calcul des préfixes par découpe dynamique n'est pas direct : dans l'algorithme précédent nous avons fait le découpage par moitié ce qui n'est pas optimal.

Par exemple supposons qu'on ait deux processeurs P_0 et P_1 identiques et que l'intervalle du travail à effectuer est $[0, n[$ affecté à P_0. Après le vol de P_1 sur P_0, le travail restant sur P_0 est $[0, \frac{n}{2}[$ et $[\frac{n}{2} + 1, n[$ le nouveau travail sur P_1. Les deux processeurs passent chacun un temps (en supposant que le temps d'une opération est constant et égal à 1) de $\frac{n}{2} - 1$ sur le calcul séquentiel des préfixes et un temps de $\frac{n}{4}$ sur la finalisation de l'intervalle $[\frac{n}{2} + 1, n[$. Donc le temps total obtenu par les deux processeurs est $T_2 = \frac{3n}{4} - 1$ or le temps optimal est de $\frac{2(n-1)}{3}$, et le temps obtenu est alors à un facteur de $\frac{9}{8}$ de l'optimal. Le nombre d'opérations effectuées par les deux processeurs est de $n - 1 + \frac{n}{2} = \frac{3n-2}{2}$, or le nombre optimal d'opérations sur 2 processeurs est de $\frac{4(n-1)}{3}$, et le nombre d'opérations obtenu est alors à un facteur de $\frac{9}{8}$ aussi. La figure 4.1 illustre cet exemple.

L'un de nos objectifs étant de diminuer le nombre d'opérations effectuées par l'algorithme tout en maintenant un temps optimal, donc une mauvaise découpe peut nous éloigner de ces objectifs. On voit dans l'exemple précédent qu'une mauvaise découpe entraîne une augmentation du nombre d'opérations, du nombre de tâches de finalisations, ce qui augmente le temps de calcul global : une mauvaise découpe augmente le surcoût de parallélisme. Une première solution pour pallier à ce problème est de faire la découpe en fonction du nombre de processeurs (ce qui été fait dans [77, 78]) ; mais dans ce cas on sera dépendant du nombre de processeurs, or un de nos objectifs est d'être indépendant du nombre de processeurs (*processor-oblivious* en anglais). La question qu'on se pose est la suivante :

– Quelle méthode faut t'il faire pour avoir une découpe optimale ?

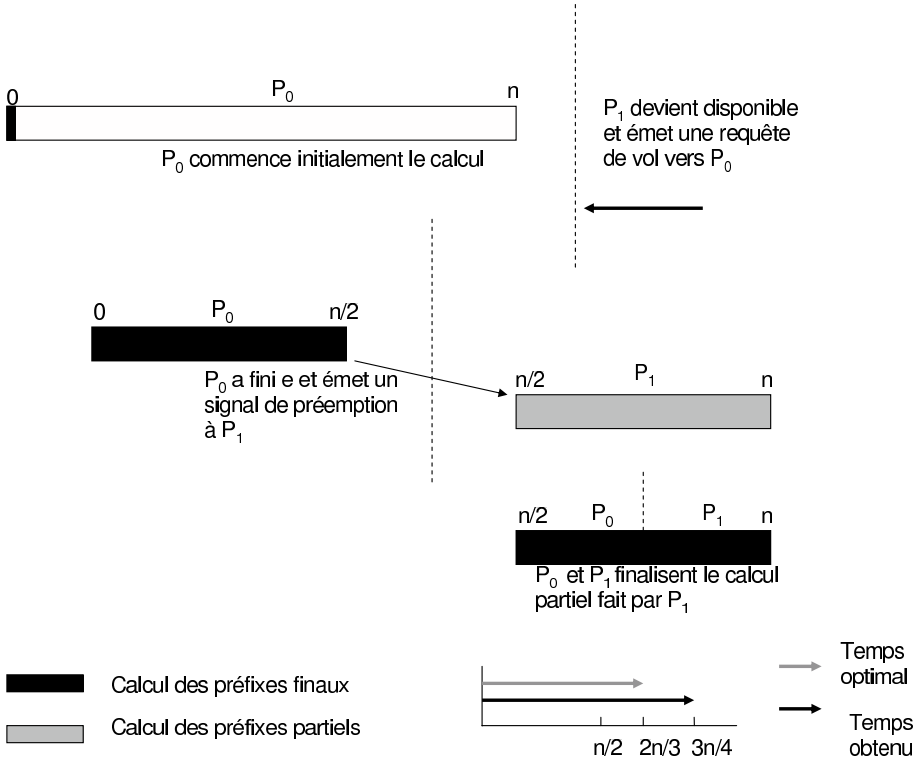

FIG. 4.1 – Exemple d'un découpage non optimal

Avant de répondre à cette question, nous reprenons l'exemple précédent avec les processeurs P_0 et P_1. Soit $n = 6s - 5$ où s est un entier positif et strictement supérieur à 0. Nous découpons le tableau initial de taille n entre trois blocs de tailles respectives $2s$, $3s - 3$ et $s - 2$. La figure 4.2 illustre une découpe indépendamment du nombre de processeurs. P_0 et P_1 appliquent l'algorithme parallèle adaptatif de préfixes présenté sur le premier bloc qui est de taille $2s$, après le vol P_1 et P_2 travaillent chacun sur un intervalle de taille s, P_0 calcule séquentiellement les préfixes finals du premier intervalle qui dure $s - 1$ unités et P_1 calcule séquentiellement les préfixes partiels du deuxième intervalle qui dure $s - 1$ unités de temps aussi. Dès que P_0 finit son calcul il préempte P_1 ; et P_0 récupère le dernier préfixe partiel calculé, et effectue un saut sur les préfixes partiels calculés ; enfin P_0 demande à P_1 de finaliser (compléter) les préfixes partiels. P_0 en effectuant le saut atteint la limite du premier bloc, il effectue d'abord la fusion du dernier préfixe final qu'il a calculé avec le préfixe partiel qu'il a récupéré chez P_1, ensuite il continue séquentiellement son calcul sur le deuxième bloc pendant que P_1 finalise les préfixes partiels du bloc précédent. Ils font tous deux $s - 1$ unités de temps sur leurs intervalles respectifs. Dès que P_1 termine sa finalisation, il part aider P_0 en lui volant la moitié de ce qui lui reste, P_0 calcule des préfixes finaux et P_1 des préfixes partiels, ce qui dure $s - 1$ unités de temps. Si P_0 finit son calcul, il effectue la préemption comme dans le premier bloc, et continue ensuite séquentiellement dans le troisième bloc son calcul pendant que P_1 fait la finalisation des préfixes partiels dans le deuxième bloc qui dure $s - 1$ unités de temps. Le temps total d'exécution du calcul est alors de $4(s-1) = \frac{2(n-1)}{3}$, ce qui donne un temps optimal. Le nombre total d'opérations effectuées par P_0 et P_1 est de $4(s-1)$ pour P_0 et $4(s-1)$ pour P_1, ce qui fait

un temps total de $8(s-1) = \frac{2(n-1)p}{p+1}$ qui est optimal aussi. Donc on peut trouver une découpe optimale indépendamment du nombre de processeurs en découpant la taille initiale du tableau en plusieurs blocs de tailles éventuellement différentes. Mais il reste toujours un problème qui est : comment choisir la taille de ces blocs ? La section suivante répond à cette question.

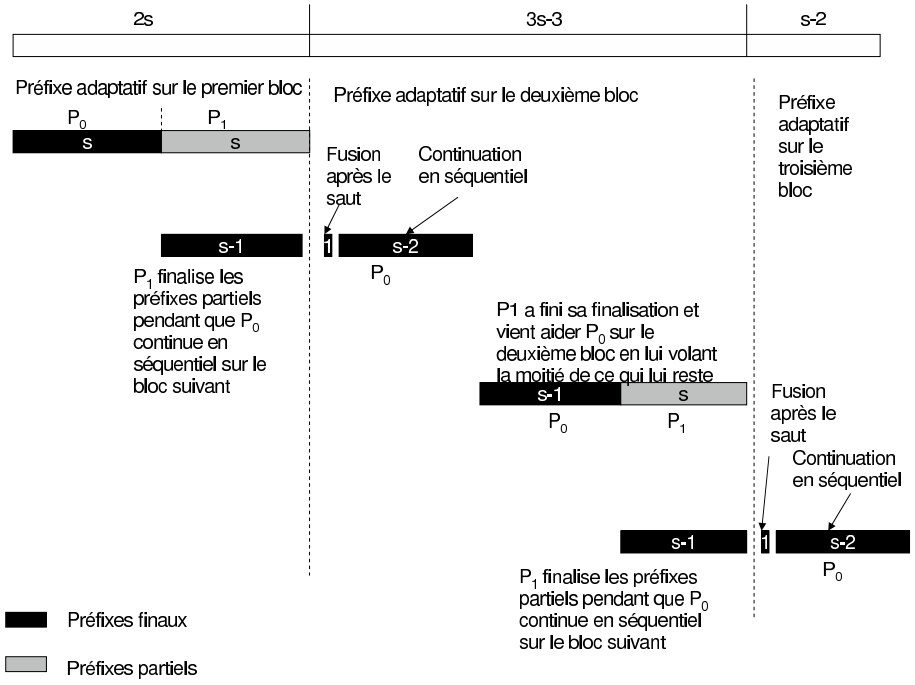

FIG. 4.2 – Exemple d'un découpage optimal

4.2.3 Algorithme optimal avec découpage adaptatif

Pour amortir ce surcoût dû au découpage, nous utilisons une technique étudiée dans [25] qui consiste à diviser dynamiquement l'intervalle global de calculs en des étapes avec synchronisation à chaque étape. Notre algorithme adaptatif sera constitué alors d'un ensemble d'étapes où dans chaque étape un intervalle de calcul de taille inconnue à priori sera exécuté en parallèle. Soit $s_1, s_2, \ldots s_i$ la taille du l'intervalle de calcul à l'étape i. Pour le choix de la taille de l'intervalle, nous utilisons la technique présentée dans [7]. Dans [7] différents choix de taille ont été étudiés, en particulier pour $s = \rho^{\frac{n}{\log n}}$ avec $1 < \rho < 2$. L'étape i de la division de taille s_i ne commence qu'après terminaison de l'étape $i-1$ pour les processus P_v, mais le processus P_s peut continuer le calcul séquentiel optimal dans les étapes en laissant derrière lui les processus P_v finalisés les préfixes partiels des étapes précédentes. Dans [25] cette division dynamique du travail global est appelée **macro-loop** et les étapes de calcul sont appelées *macro-step*. Donc, dans l'algorithme lorsque le processus P_s atteint la fin de la *macro-step* k, il passe à la macro-etape $k+1$, les autres processus voleurs P_v ne peuvent passer à la *macro-step* $k+1$ tant qu'ils n'ont pas fini la finalisation des préfixes partiels de la *macro-step* k. Notez que seulement tous les processus voleurs se synchronisent dans la **macro-loop**. Le processus P_s ne participant pas

à la finalisation, sauf éventuellement pendant la dernière étape de la macro-loop. Nous définissons $s_{k+1} = \frac{n_k}{\epsilon(n_k)}$ avec $\epsilon(n) = o(n)$ et $\epsilon(n) \to +\infty$ quand $n \to +\infty$, et nous supposons que s_1 est initialement égale à une valeur constante. Typiquement, nous considérons dans la suite $\epsilon(n) = \log n$.

Le théorème suivant montre que l'algorithme de préfixe à grain adaptatif indépendant du nombre de processeurs et leurs vitesses respectives est asymptotiquement optimal.

Théorème 4.2.1. *Le temps T_p sur p processeurs du calcul de $n+1$ préfixes par l'algorithme adaptatif avec macro-loop vérifie avec une grande probabilité :*

$$T_p \leq \frac{2n}{\Pi_{ave}(p+1)} + O\left(\frac{n}{\epsilon(n)} + \log^3 n\right) \sim_{n\to+\infty} \frac{2n}{\Pi_{ave}(p+1)}$$

Démonstration. On analyse l'exécution en la découpant en deux phases, la phase ϕ_1 jusqu'à ce que le processus P_s qui exécute la partie séquentielle de l'algorithme adaptatif termine avec le calcul du dernier préfixe π_n et la phase ϕ_2 jusqu'à la fin du calcul. Par définition de la *macro-step*, la taille de la dernière *macro-step* est $O\left(\frac{n}{\epsilon(n)}\right)$.

Soient Π_{seq} la vitesse moyenne du processus P_s, Π_{ϕ_1} la vitesse moyenne des processus pendant la phase ϕ_1 et Π_{ϕ_2} la vitesse moyenne des processus pendant la phase ϕ_2. Pour simplifier nous supposons que $\Pi_{seq} = \Pi_{\phi_1} = \Pi_{\phi_2} = \Pi_{ave}$. Soit n_{seq} le nombre de préfixes calculés de manière séquentielle par P_s dans la phase ϕ_1. Soit n_{final} le nombre de préfixes finaux calculés par les processus P_v dans la phase ϕ_1. Soit j le nombre de sauts effectués par le processus P_s dans la phase ϕ_1 et i_k ($k \in \{1,2\}$) le nombre de tops d'inactivité dans la phase ϕ_k. Soient $T_p(\phi_1)$ et $T_p(\phi_2)$ le temps parallèle de la phase ϕ_1 et ϕ_2.

Durant la phase ϕ_1 de temps $T_p(\phi_1)$: si nous considérons le processus P_s qui exécute toujours, on a

$$T_p(\phi_1) = \frac{n_{seq} + j}{\Pi_{ave}} \tag{4.1}$$

Si nous considérons les processus P_v qui exécutent un algorithme parallèle, avec découpe récursive sur vol de travail (profondeur $O(\log n)$) , la profondeur finale de cet algorithme parallèle est de $D = \epsilon(n).\log n = \log^2 n$ due à la macro-loop (nombre d'étapes dans la **macro-loop** étant égal à $\log n$), on a

$$T_p(\phi_1) = \frac{n - n_{seq} + n_{final} + i_1}{\Pi_{ave}(p-1)} \tag{4.2}$$

En utilisant le théorème 3.4.1, on a $j = O\left((p-1).\log^3 n\right)$ (car à chaque préemption P_s attend au plus $\Omega(\log n)$: lorsque le processus à préempter est en cours de calcul qui ayant extrait à l'aide de l'opération **extract_seq** au plus $\Omega(\log n)$ comme taille du calcul à faire séquentiellement, et on a aussi le nombre total de vols borné par $O\left((p-1).\log^2 n\right)$), et

$i_1 = O\left((p-1).\log^2 n\right)$. Nécessairement on a $n_{final} \leq n - n_{seq}$. Donc, en multipliant l'équation 4.1 par 2 et l'équation 4.2 par $p-1$ on en déduit que

$$T_p(\phi_1) \leq \frac{2n}{\Pi_{ave}(p+1)} + O(\log^3 n) \tag{4.3}$$

Durant la phase ϕ_2 de temps $T_p(\phi_2)$, il ne reste que les préfixes partiels de la dernière étape de la macro-loop à finaliser, soit $n - n_{seq} - n_{final} = O\left(\frac{n}{\epsilon(n)}\right)$ calculs à effectuer, on a donc :

$$T_p(\phi_2) = \frac{n - n_{seq} - n_{final} + i_2}{p\Pi_{ave}} \tag{4.4}$$

En appliquant le théorème 3.4.1, on a $i_2 = O\left(\log\left(n - n_{seq} - n_{final}\right)\right) = O\left(\log \frac{n}{\epsilon(n)}\right) = O(\log n)$. Donc

$$T_p(\phi_2) \leq \frac{1}{p\Pi_{ave}}.O\left(\frac{n}{\epsilon(n)}\right) + O(\log n) \tag{4.5}$$

En addition l'équation 4.3 et 4.5 on obtient : $T_p \leq \frac{2n}{\Pi_{ave}(p+1)} + O\left(\frac{n}{\epsilon(n)} + \log^3 n\right)$.

❑

4.3 Expérimentations

Nous présentons dans cette section une évaluation des performances de notre algorithme adaptatif du calcul parallèle des préfixes.

4.3.1 Préliminaires

Nous avons fait nos expérimentations sur deux machines différentes qui sont :

– La machine *idbull*. Cette machine est une machine à mémoire partagée(SMP) composée de 2 processeurs quadri-coeurs (soit 8 coeurs au total). Elle dispose des processeurs Intel Itanium-2 cadencés à 1,5Ghz et 31 GB de mémoire. Sur cette machine, nous avons utilisé le noyau Lunix 2.6.7, le compilateur gcc 4.2.3 et l'option de compilation -02.
– La machine *idkoiff*. Cette machine est composée de huit processeurs Dual Core AMD Opteron 875, soit 16 coeurs au total à 2,2Ghz, dispose de huit bancs de 4Go de mémoire. Sur cette machine, nous avons utilisé le noyau Lunix 2.6.23, le compilateur gcc 4.2.3 et

l'option de compilation -02.

L'algorithme parallèle adaptatif (implantation "préfixe adaptatif") a été implanté sur Kaapi [46] qui intègre l'ordonnancement par vol de travail. Nous avons aussi implanté l'algorithme statique (implantation "préfixe statique") présenté précédemment (section 3.3.2) qui est théoriquement optimal sur des processeurs identiques avec mémoire uniforme.

Nous divisons nos expérimentations en quatre parties qui sont :

- Expérimentation sur des processeurs identiques. Nous présentons dans cette partie, une comparaison de l'accélération obtenue par l'implantation "prefixe adaptatif" avec l'accélération théorique optimale, et aussi nous nous comparons à l'implantation "préfixe statique".
- Expérimentation sur des processeurs perturbés. Dans cette partie nous présentons une évaluation des performances obtenues par l'implantation "préfixe adaptatif" dans le cas où les processeurs sont de vitesses variables par rapport à l'application, du fait de leur utilisation par d'autres processus (des charges additionnelles).
- Expérimentation sur des processeurs hétérogènes. Dans cette partie, nous présentons les performances obtenues par notre algorithme sur des architectures hétérogènes (fréquences différentes) simulées à l'aide d'un simulateur de charge CPU développé par Christophe Cérin de l'université de Paris 13 dans le cadre du projet SAFESCALE.
- Expérimentation en distribué. Dans cette partie, nous présentons des résultats obtenus sur des processeurs distribués.

4.3.2 Évaluations

Expérimentations sur des processeurs identiques

Les premières expérimentations ont pour but de montrer que l'implantation "préfixe adaptatif" se comporte bien sur des processeurs identiques. Nous comparons l'accélération obtenue expérimentalement par l'implantation "préfixe adaptatif" à l'accélération théorique optimale (théorème 4.2.1 avec $\Pi_{ave} = 1$). Aussi nous comparons l'implantation "préfixe adaptatif" à l'implantation "préfixe statique".

On considères deux cas pour les expérimentations : le premier cas concerne un temps par opération $\star$ élevé et le deuxième cas concerne un temps par opération $\star$ faible.

Première classe d'expériences

Pour le premier cas, les expérimentations consistent au calcul des préfixes de 30000 éléments en faisant varier p le nombre de processeurs utilisés (de 1 à 8 pour *idbull* et de 1 à 16 pour *idkoiff*). Le temps séquentiel optimal de référence est de $19,31$s pour *idbull* et de $47,35$s

pour *idkoiff*. La taille s_0 de la première étape de la **macro-loop** a été fixée à 30 et la constante α fixée à 1. Les seuils ont été choisis aux valeurs qui ont donné les meilleures performances expérimentales.

Le tableau 4.1 donne les temps d'exécution obtenus par l'implantation "préfixe adaptatif" sur p processeurs (de 1 à 8 pour *idbull* et 1 à 16 pour *idkoiff*). Pour chaque expérience, 10 mesures ont été effectuées et seuls sont reportés les temps de l'exécution la plus rapide, la plus lente et le temps moyen des 10 exécutions.

Le tableau 4.1 montre les temps d'exécution lorsqu'il n'y a pas d'autres calculs en cours sur les processeurs. On remarque que les mesures de temps sont stables (écart entre temps minimum et maximum inférieur à 4%). On vérifie l'optimalité de l'algorithme adaptatif (théorème 4.2.1 avec $\Pi_{ave} = 1$) qui est à moins de 2% sur *idbull* et de 7% sur *idkoiff* de la borne inférieure.

	idbull				*idkoiff*					
	p=1	p=2	p=4	p=8	p= 1	p=2	p=5	p=10	p=12	p=16
Minimum	19,31	12,92	7,78	4,33	47,35	31,64	15,85	8,70	7,38	5,78
Maximum	19,35	12,92	7,84	4,42	47,35	31,64	16,18	8,76	7,57	6,03
Moyenne	19,33	12,92	7,81	4,35	47,35	31,64	15,97	8,72	7,47	5,87
Borne inférieure	19,31	12,87	7,724	4,29	47,35	31,57	15,78	8,61	7,28	5,57

TAB. 4.1 – Temps d'exécution de l'implantation "préfixe adaptatif" sur le tableau de taille $n = 3.10^4$ sur les machines *idbull* et *idkoiff*.

Les figures 4.3 et 4.4 comparent les accélérations obtenues par l'implantation "préfixe adaptatif" sur les machines *idbull* et *idkoiff* à l'accélération théorique optimale ($\frac{p+1}{2}$) sur le tableau de taille $n = 3.10^4$. Sur la machine *idbull*, nous observons que les accélérations obtenues par l'implantation "préfixe adaptatif" sont identiques aux accélérations théoriques optimales sur 1 à 8 processeurs (figure 4.3). Sur la machine *idkoiff* nous observons que les accélérations obtenues par l'implantation "préfixe adaptatif" sont identiques aux accélérations théoriques optimales sur 1 à 10 processeurs ; et sur 12 à 16 elles sont légèrement inférieures aux accélérations théoriques.

Deuxième classe d'expériences

Les figures 4.5 et 4.6 comparent les temps d'exécution de l'implantation "préfixe adaptatif" à ceux de l'implantation "préfixe statique" qui est basée sur une découpe en $p+1$ parties. Pour le deuxième cas les expérimentations consistent au calcul des préfixes de 10^8 doubles (l'opération $\star$ est l'addition de deux doubles) en faisant varier p le nombre de processeurs utilisés (de 1 à 8 pour *idbull* et de 1 à 16 pour *idkoiff*). Le temps séquentiel optimal de référence est de $3,14$s pour *idbull* et de $1,55$s pour *idkoiff*. La taille s_0 de la première étape de la **macro-loop** a été fixée à 300 et la constante α fixée à 100. Nous remarquons que sur les deux machines, l'implantation "préfixe adaptatif" se comporte mieux que l'implantation "préfixe statique". Cette différence s'explique par le fait qu'il y a des petites charges dans le système, ce qui perturbe un peu l'implantation "préfixe statique", et l'implantation "préfixe adaptatif" s'adapte en fonction de ces charges. Nous pouvons remarquer que la meilleure accélération obtenue par ces

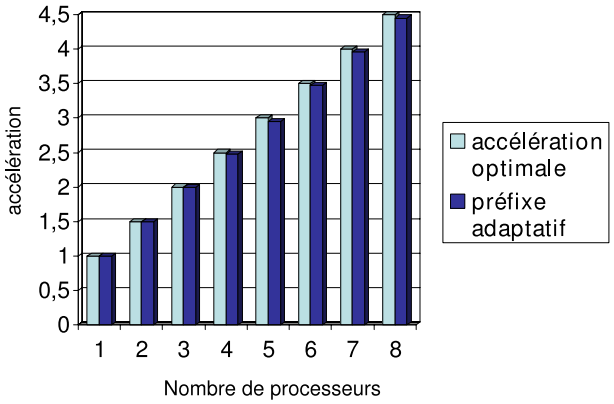

FIG. 4.3 – Comparaison sur *idbull* de l'accélération obtenue par l'implantation "préfixe adaptatif" ($n = 3.10^4$) avec l'accélération théorique optimal ($\frac{p+1}{2}$).

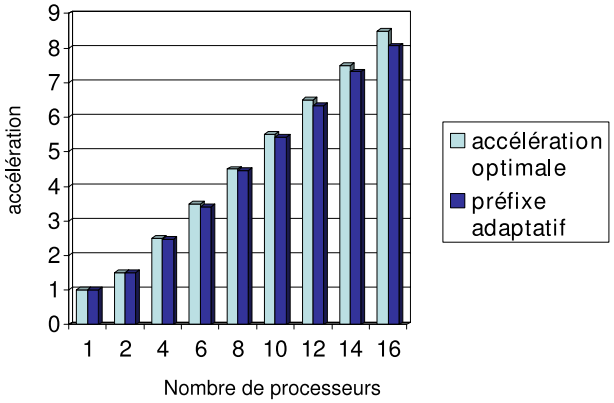

FIG. 4.4 – Comparaison sur *idkoiff* de l'accélération obtenue par l'implantation "préfixe adaptatif" ($n = 3.10^4$) avec l'accélération théorique optimal ($\frac{p+1}{2}$).

expérimentations ne dépasse pas 4. Ceci est dû à la contention d'accès mémoire sur ces machines NUMA où chaque bi-processeurs partage la même mémoire. Mais si nous augmentons le temps de l'opération $\star$, l'accélération augmente, ce qui été montrée dans les expérimentations précédentes.

Expérimentations en contexte multi-utilisateurs

Dans le tableau 4.2 des processus de charges additionnels sont injectés pour perturber l'occupation de la machine et simuler le comportement d'une machine réelle, perturbée par d'autres utilisateurs. Par souci de reproductibilité, chaque expérience sur $p \leq 16$ processeurs est perturbée par $16 - p + 1$ processus artificiels de durée supérieure à 19s (le temps séquentiel de référence est de 19s). On peut vérifier dans le tableau 4.2 qu'en moyenne l'implantation "préfixe adaptatif" est au moins 13% plus rapide.

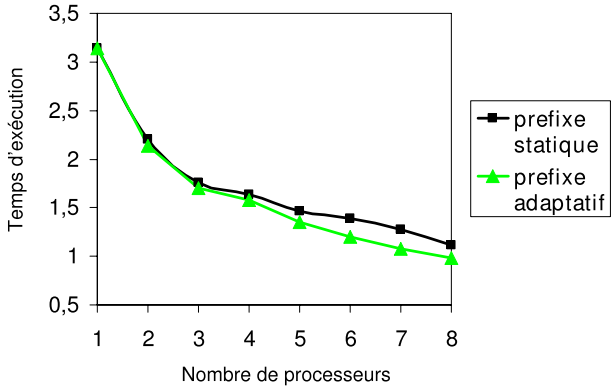

FIG. 4.5 – temps d'exécution pour $n = 10^8$ sur la machine *idbull*

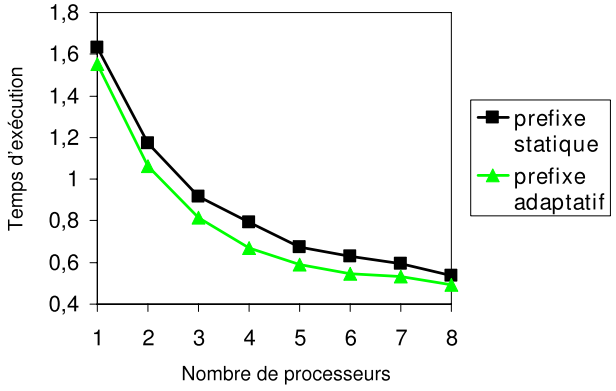

FIG. 4.6 – temps d'exécution pour $n = 10^8$ sur la machine *idkoiff*

En conclusion, l'algorithme à grain adaptatif apporte une performance garantie lorsque la machine est partagée entre plusieurs utilisateurs, en s'adaptant automatiquement aux ressources disponibles au cours de l'exécution. De plus sa performance reste proche de l'optimale même dans le cas idéal où les processeurs sont tous dédiés à l'application. Il apparaît donc être plus performant que l'algorithme séquentiel ou qu'un algorithme parallèle à grain fixé.

Expérimentations sur des processeurs hétérogènes

Pour simuler l'hétérogénéité des cœurs, nous avons utilisé un simulateur de charge CPU développé par Christophe Cérin dans le cadre du projet SAFESCALE. Des tests similaires ont été réalisés sur un autre simulateur de charge développé au laboratoire [68]. Pour faciliter la simulation, un processus a été fixé sur chaque cœur et ne sera pas migré tout long de l'exécution sur un autre cœur. La figure 4.7 compare les accélérations obtenues par l'implantation "préfixe adaptatif" à l'accélération théorique optimale ($\frac{P.\Pi_{ave}+\Pi_{max}}{2}$), et à l'accélération de l'implantation "préfixe statique" basé sur une découpe en $p+1$ parties de la bibliothèque SWARM [5]. Ici nous avons donné la même vitesse à tous les processeurs, sauf un qui va deux fois moins vite que les autres. Nous observons que l'accélération obtenue par l'implantation "préfixe adaptatif"

	Statique						Adaptatif					
	p=2	p=4	p=8	p=12	p=14	p=16	p=2	p=4	p=8	p=12	p=14	p=16
Minimum	16,10	10,48	5,91	4,03	3,48	3,13	16,11	9,27	4,80	3,31	2,90	2,70
Maximum	22,22	13,60	7,77	4,11	3,60	3,17	18,17	9,99	5,04	3,71	3,02	2,77
Moyenne	19,06	11,42	6,91	4,08	3,55	3,14	17,52	9,58	4,93	3,50	2,96	2,76

TAB. 4.2 – Comparaison des temps des implantations sur p processeurs perturbés. Sur les 10 exécutions de chacun des tests, l'implantation "préfixe adaptatif" est la plus rapide.

est proche de l'optimale, ce qui montre qu'il s'adapte bien à l'hétérogénéité des coeurs. On remarque que l'accélération obtenue par l'implantation "préfixe statique" est loin de l'optimale et elle est égale à $\frac{p+1}{4}$, qui est égale à l'accélération du processeur le plus lent.

La figure 4.8 compare les trois temps (minimal, moyen, maximal) obtenus par l'implantation "préfixe adaptatif", au temps théorique $\frac{2T_{seq}}{(p+1)\Pi_{ave}}$, et aux temps obtenus par l'implantation "préfixe statique" implanté avec la librairie SWARM. Les deux implantations (adaptative et statique) ont été testées sur des cœurs hétérogènes (obtenus à l'aide du simulateur) dont les caractéristiques sont données dans le tableau 4.3. Le temps séquentiel de référence est de $15,92s$ qui a été mesuré sur un seul cœur de fréquence maximale égale à $2200Mhz$ et avec une charge CPU égale à 100%. La figure montre que conformément à la théorie les temps obtenus par notre algorithme sont très proches de $\frac{2T_{seq}}{(p+1)\Pi_{ave}}$, et elle montre aussi que ces temps sont très meilleurs à celui de l'algorithme statique. On peut observer sur cette figure que les temps obtenus par l'implantation "préfixe adaptatif" sont stables. Sur la figure les processeurs sont utilisés à partir de l'indice croissant de leur numéro indiqué dans le tableau 4.3, par exemple si le nombre de cœurs est égal à i, cela signifie que les processeurs $P_1, \ldots P_i$ ont été utilisés.

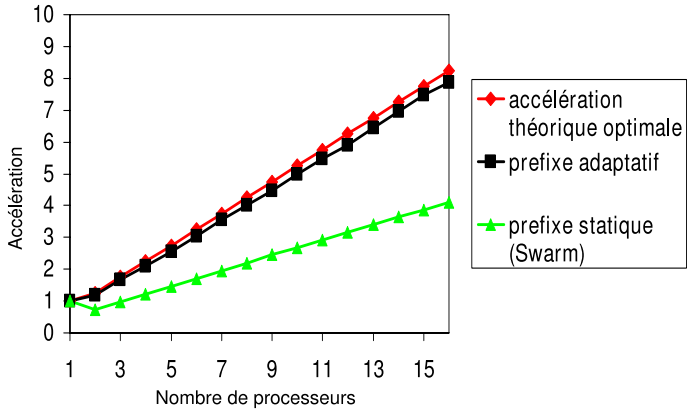

FIG. 4.7 – Comparaison de l'implantation "préfixe statique" en SWARM et l'implantation "préfixe adaptatif" en Athapasacan/Kaapi sur un tableau de taille $n = 10000$ sur des processeurs de vitesses différentes (parmi p processeurs, $p-1$ processeurs vont a la vitesse maximale et un processeur va deux fois moins vite que les autres)

Expérimentations en distribué

Lors d'une exécution distribuée, les surcoûts de copie des données du tableau en entrée

Numéro	Fréquence	Numéro	Fréquence	Nombre de cœurs	cœurs utilisés	Nombre de cœurs	cœurs utilisés
P_1	$1760Mhz$	P_6	$440Mhz$	1	P_1	6	$P_1 \dots P_6$
P_2	$550Mhz$	P_7	$1320Mhz$	2	$P_1 \dots P_2$	7	$P_1 \dots P_7$
P_3	$1100Mhz$	P_8	$880Mhz$	3	$P_1 \dots P_3$	8	$P_1 \dots P_8$
P_4	$1650Mhz$	P_9	$550Mhz$	4	$P_1 \dots P_4$	9	$P_1 \dots P_9$
P_5	$660Mhz$	P_{10}	$1760Mhz$	5	$P_1 \dots P_5$	10	$P_1 \dots P_{10}$

TAB. 4.3 – Fréquences des différents cœurs utilisés et la façon dont ils ont été utilisés.

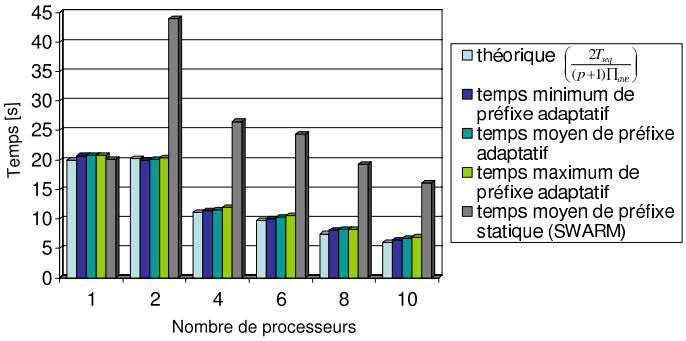

FIG. 4.8 – Comparaison de l'implantation "préfixe statique" en SWARM et l'implantation "préfixe adaptatif" en Athapasacn/Kaapi sur un tableau de taille $n = 10000$ sur des cœurs hétérogènes.

peuvent conduire à des mauvaises performances. Pour limiter ce surcoût, les données ont été copiées dans un fichier, et ce fichier a été projeté en mémoire (*mmap*).

Pour réaliser cette exécution distribuée, 8 machines *idbull* ont été utilisées à l'aide de la commande *karun* de Kaapi. Dans ce cas distribué, l'extraction d'une partie du travail est réalisée par l'exécution, sur interruption, d'un service à distance chez le processeur volé ; la réalisation de ce mécanisme a été faite en utilisant les fonctions permettant de réaliser des communications par échange de message ("message actif") développées dans Kaapi [39]. la figure 4.9 compare l'accélération obtenue par notre algorithme en distribué sur 8 processeurs à l'accélération optimale. Nous avons utilisé un fichier contenant 10000 pages dont chaque page contient 8192 doubles. Le grain de la fonction **extract_seq** a été fixé à la taille d'une page, c'est à dire 8192 doubles. Nous pouvons remarquer sur la figure que l'implantation "préfixe adaptatif" se comporte aussi bien.

4.4 Conclusion

Motivés par l'utilisation de machines multi-processeurs partagées entre plusieurs utilisateurs, dans ce chapitre nous avons introduit un nouvel algorithme parallèle pour le calcul des préfixes qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles. Nous avons montré que son travail était asymptotiquement optimal. Il est équivalent à celui de l'algorithme séquentiel lorsqu'un seul processeur est disponible et à celui d'un al-

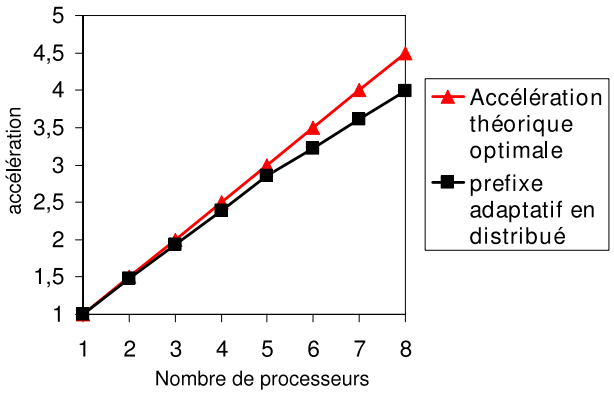

FIG. 4.9 – L'accélération sur 1 à 8 machines *idbull* en distribué sur un fichier de 10000 pages avec chaque page contenant 8192 doubles

gorithme parallèle optimal lorsque p processeurs identiques sont disponibles. Dans le cas de p processeurs de vitesses variables, son temps est équivalent à celui d'un algorithme optimal sur p processeurs identiques de vitesse égale à la moyenne des vitesses. Ces résultats théoriques sont validés par les expérimentations menées sur des machines de 8 et 16 processeurs. Plus généralement, notre algorithme adaptatif est basé sur le couplage récursif et dynamique de deux algorithmes un algorithme séquentiel, optimal en nombre d'opérations, et l'autre parallèle avec un degré maximal de parallélisme. Dans le chapitre suivant, nous proposons des algorithmes adaptatifs de la fusion de deux listes triées, de partition et ces algorithmes seront utilisés pour paralléliser des algorithmes de tri introspectif (une variante du tri rapide) et de tri par fusion.

Chapitre 5

Algorithmes adaptatifs de tri parallèle

Dans ce chapitre, nous présentons deux algorithmes parallèles de tri pour des architectures multicœurs à mémoire partagée dont le nombre ou la vitesse des processeurs physiques alloués à une application donnée peuvent varier en cours d'exécution. Le premier repose sur une fusion adaptative parallèle, le deuxième sur une partition parallèle adaptative. Les performances théoriques sont prouvées asymptotiquement optimales par rapport au temps séquentiel. Les performances expérimentales sont analysées sur deux machines différentes à 8 et 16 cœurs.

5.1 Introduction

Le tri d'un ensemble de données est l'un des problèmes fondamentaux les plus étudiés en informatique. Il est souvent dit que 25% à 50% du travail réalisé par un ordinateur est accompli par des algorithmes de tri [21]. L'une de ces raisons est qu'un ensemble de données trié est plus facile à manipuler qu'un ensemble non trié ; par exemple la recherche d'une valeur dans une séquence triée est généralement plus rapide que dans une séquence non triée. Ainsi, de part son importance pratique considérable, il est intégré dans de nombreuses bibliothèques séquentielles et parallèles.

Le tri parallèle a été étudié expérimentalement par beaucoup d'auteurs sur différentes architectures [16, 80, 20, 50, 31] que nous ne pouvons pas tous mentionner dans ce chapitre. Nous nous intéressons aux algorithmes parallèles basés sur le tri rapide [19] et le tri par fusion. Dans [42], les auteurs proposent une parallélisation du tri rapide qui repose sur une parallélisation statique de la partition par bloc basée sur la méthode "fetch and add". Tsigas et Zhang [91] ont étudié expérimentalement une version similaire à celle proposée dans [42] ; leurs expérimentations donnent de bons résultats. Le temps d'exécution de leur algorithme avec en entrée un tableau de taille n sur p processeurs identiques est en $O\left(\frac{n \log n}{p}\right)$ en moyenne et en $O\left(\frac{n^2}{p}\right)$ dans le pire des cas. Outre un pire cas en $O(n^2)$ en travail, l'algorithme parallèle proposé par Tsigas et Zhang n'est pas performant si l'on dispose d'une machine avec des processeurs dif-

férents, ou d'une machine utilisée par plusieurs utilisateurs car la charge des processeurs varie en cours d'exécution. Dans [84], les auteurs ont étudié expérimentalement une version parallèle du tri par fusion qui repose sur une fusion parallèle. Leur algorithme de fusion parallèle est fait en fonction du nombre de processeurs, ce qui n'est pas adapté aussi sur des architectures multicœurs.

Pour réaliser des exécutions efficaces de ces deux algorithmes sur des architectures multicœurs, nous utilisons la technique adaptative présentée dans le chapitre 6. Nous présentons dans la section 5.3 un algorithme adaptatif de la fusion parallèle de deux listes triées. Puis à partir de cette fusion parallèle adaptative, nous présentons un algorithme adaptatif de tri parallèle par fusion dans la section 5.3. Cet algorithme parallèle adaptatif est une version améliorée de l'algorithme classique présenté dans [89]. Dans la section 5.4, nous proposons un algorithme adaptatif de la partition parallèle. Puis à partir de cette partition adaptative, nous proposons un algorithme parallèle adaptatif du tri rapide introspectif [64] dans la section 5.5.

5.2 La fusion adaptative de listes triées

Dans toute la suite, nous utilisons des tableaux à accès direct pour effectuer la fusion. L'intervalle de travail restant en cours sur un tableau T est représenté par $w = [d, f[$ où d et f sont deux indices de T, nous désignons par $m = f - d$ son nombre d'éléments. On suppose que les indices sont numérotés à partir de 0 ; ainsi le l'intervalle initial de travail d'un tableau T de taille n (nombre d'éléments) est représenté par $[0, n[$ où T contient les éléments $T[0], \ldots, T[n-1]$.

L'algorithme de tri par fusion consiste à prendre deux tableaux triés, notés T_1 et T_2 disjoints du tableau T ; et à interclasser les éléments de T_1 et T_2 pour les stocker triés dans un tableau T.

Nous désignons par m_1 et m_2 le nombre d'éléments des tableaux restant à fusionner, on note $w_1 = [i, i + m_1[$ et $w_1 = [j, j + m_2[$ les intervalles d'indices des sous-tableaux correspondants dans T_1 et T_2. La figure 5.1 montre un exemple de la fusion de deux tableaux triés.

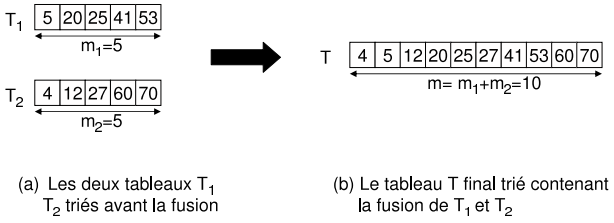

FIG. 5.1 – T est la fusion de T_1 et T_2

Pour construire la fusion adaptative parallèle, comme pour le calcul adaptatif des préfixes, nous utilisons les opérations **extract_seq** et **extract_par** qui permettent d'extraire les tailles des calculs à effectuer localement et en parallèle. Pour minimiser le surcoût de synchronisations ces

trois fonctions ont été implémentées comme suit :

- **extract_seq** : consiste à extraire $\alpha \log m_1$ sur w_1 ou $\alpha \log m_2$ sur w_2. Trois cas peuvent apparaître lors de l'opération d'extraction :
  - cas 1 : $w_1 \neq \emptyset$ et $w_2 \neq \emptyset$, l'opération consiste à extraire $\alpha \log m_1$ sur w_1 et $\alpha \log m_2$ sur w_2. La figure 5.2 illustre ce cas.
  - cas 2 : $w_1 \neq \emptyset$ (resp. $w_2 \neq \emptyset$) et $w_2 \neq \emptyset$ (resp. $w_1 \neq \emptyset$), l'opération consiste à extraire $\alpha \log m_2$ (resp. $\alpha \log m_1$) sur w_2 (resp. w_1). La figure 5.3 illustre ce cas.
  - cas 3 : $w_1 = \emptyset$ et $w_2 = \emptyset$ l'opération retourne faux.

- **extract_par** : consiste à prendre l'élément e au milieu du tableau qui a la plus grande taille, ce qui divisera celui-ci en deux parties et à chercher, par recherche dichotomique en $\log(min(m_1, m_2))$ comparaisons, l'indice i_e du premier élément supérieur à e dans le tableau qui a la plus petite taille, ce qui divisera celui-ci aussi en deux parties. Un cas particulier doit être pris en compte : lors de la recherche dichotomique lorsque le premier élément supérieur à e est le premier élément du tableau de plus petite taille et que la victime est en cours d'exécution, le voleur attend qu'elle termine l'exécution du travail en cours (**local_run**) donc au plus $\alpha \log(max(m_1, m_2))$ comparaisons ; puis le voleur cherche de nouveau le premier élément supérieur à e sur le travail non terminé par le **local_run** ; ce qui prend au plus $\log \alpha + \log \log(max(m_1, m_2))$. Ensuite le voleur vole les dernières parties des deux tableaux et les deux premières parties restent chez la victime. La victime récupère aussi la position du tableau de sortie qui doit correspondre à la position du début des éléments qu'elle doit fusionner. L'extraction se fait donc toujours en temps logarithmique. La figure 5.4 illustre cette extraction de travail.

Nous appelons par **local_run** la fonction qui permet d'exécuter sans aucun surcoût de parallélisme l'intervalle extrait par la fonction **extract_seq**. Elle été implémenté comme suit :

- **local_run** : applique l'algorithme séquentiel optimal sur les deux parties extraites par **extract_seq**. L'algorithme 5.2.1 représente le **local_run**.

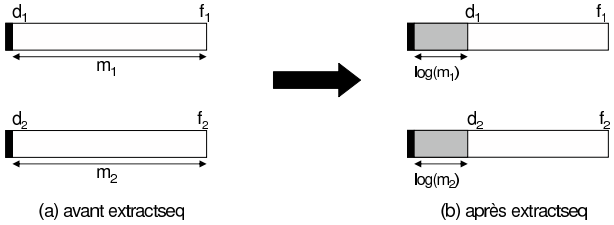

FIG. 5.2 – Illustration de la fonction **extract_seq** : cas 1

Le théorème suivant montre l'optimalité asymptotique de l'algorithme adaptatif de la fusion en prenant en compte le surcoût d'ordonnancement. Nous supposons que n est la somme des tailles des deux tableaux à fusionner.

Théorème 5.2.1. *Soit $W_s(n)$ le travail séquentiel de l'algorithme de fusion et soit $T_p(n)$ son temps d'exécution sur p processeurs de vitesse moyenne Π_{ave}. Alors, pour n suffisamment grand,*

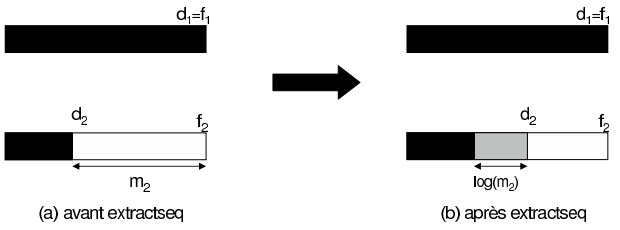

FIG. 5.3 – Illustration de la fonction **extract_seq** : cas 2

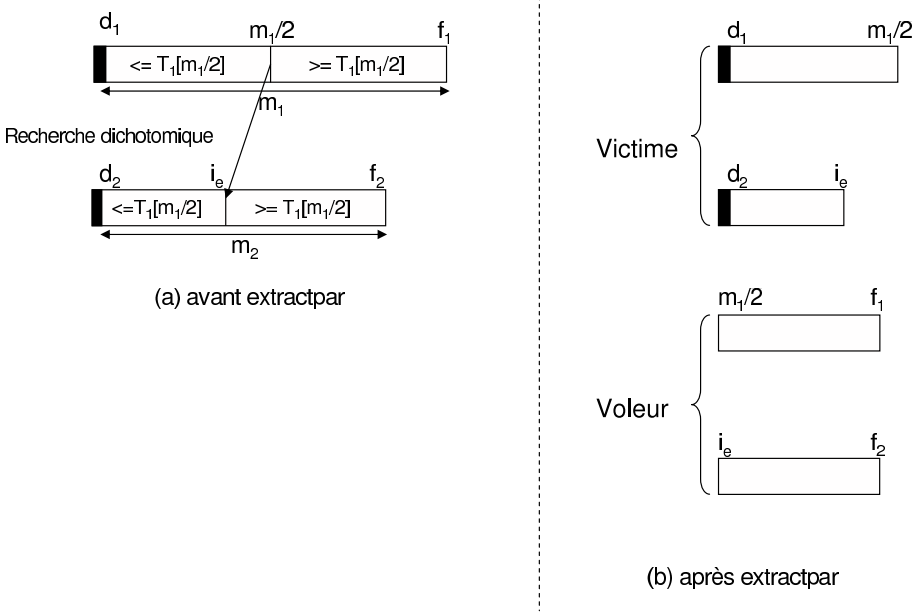

FIG. 5.4 – Illustration de la fonction **extract_par**

on a :

$$T_p(n) = \frac{W_s(n)}{p.\Pi_{ave}} \left(1 + O\left(\frac{1}{\log n}\right)\right) + O\left(\frac{\log^2 n}{\Pi_{ave}}\right).$$

Ce qui est asymptotiquement optimal.

Démonstration. Lors du vol, l'opération **extract_par** (la recherche binaire sur le tableau de plus petite taille) fait au plus $O(\log n)$ comparaisons. Le temps de chaque vol est alors $O(\log n)$. Après le vol, les parties extraites par le voleur et celles sur la victime seront exécutées en parallèle ; donc la profondeur de l'algorithme est égale à la profondeur de ces parties extraites additionnée de $O(\log n)$. Dans le pire des cas, la moitié du tableau de plus grande taille et tout le tableau de plus petite taille restent chez la victime ou seront volés par le voleur, dans ce cas la victime ou le voleur interclasse au plus $\frac{3n}{4}$ éléments. Donc la profondeur $D(n)$ de l'algorithme est $D(n) = D(\frac{3n}{4}) + O(\log n) = O\left(\log^2 n\right)$. Puisque chaque voleur exécute

l'algorithme séquentiel optimal et que le nombre de **extract_seq** est au plus $O\left(\frac{n}{\log n}\right)$, alors le travail séquentiel effectué par l'algorithme sur p processeurs est au plus $W(n) = W_s(n) + O\left(\frac{n}{\log n}\right)$ qui est asymptotiquement égal au travail $W_s(n) = O(n)$ de la fusion séquentielle optimale. Grâce au théorème 3.4.2, on a donc $T_p \leq \frac{W_s(n)}{p.\Pi_{ave}}\left(1 + O\left(\frac{1}{\log n}\right)\right) + O\left(\frac{\log^2 n}{\Pi_{ave}}\right) \simeq \frac{W_s(n)}{p.\Pi_{ave}}$ qui est asymptotiquement optimal. ❑

Algorithme 5.2.2

```
1   local_run() {
2       Eltype *first1 = _first1 + _local_seq_debut1;
3       Eltype *last1 = _first1 + _local_seq_fin1;
4       Eltype *first2 = _first2 + _local_seq_debut2;
5       Eltype *last2 = _first2 + _local_seq_fin2;
6       Eltype *output = _output + _taille_elts_interclasses;
7       if (first1 >= last1) {
8           memcpy(output, first2, sizeof(Eltype)*(last2 - first2));
9           _taille_elts_interclasses+ = (last2 - first2);
10          _local_seq_debut2 = _local_seq_fin2;
11      }
12      else if (first2 >= last2) {
13          memcpy(output, first1, sizeof(Eltype)*(last1 - first1));
14          _taille_elts_interclasses+ = (last1 - first1);
15          _local_seq_debut1 = _local_seq_fin1;
16      }
17      else {
18          while((first1! = last1) && (first2! = last2)) {
19              if( *first2 < *first1) *output + + = *first2 + +;
20              else *output + + = *first1 + +;
22          }
23      _local_seq_debut1 = (first1 - _first1);
24      _local_seq_debut2 = (first2 - _first2);
25      _taille_elts_interclasses = _local_seq_debut1 + _local_seq_debut2;
26      }
27  };
```

Algorithme 5.2.1: Algorithme de **local_run** spécialisé pour la fusion. Les types $\_first1$, $\_first2$, $\_last1$, $\_last2$ représentent les débuts et fins des deux tableaux à fusionner ; le type $output$ représente le tableau final qui doit contenir le résultat de la fusion ; les types $\_local\_seq\_debut1$, $\_local\_seq\_debut2$ représentent les positions courantes des deux tableaux en entrée ; et le type $\_taille\_elts\_interclasses$ représente le nombre d'éléments déjà interclassés dans le tableau final.

5.3 Le tri par fusion parallèle

Le tri par fusion consiste à découper le tableau initial de taille n en deux parties de taille $n/2$ qui peuvent être triées en parallèle puis à fusionner ces deux parties. La figure 5.5 montre une illustration sur un exemple du tri par fusion.

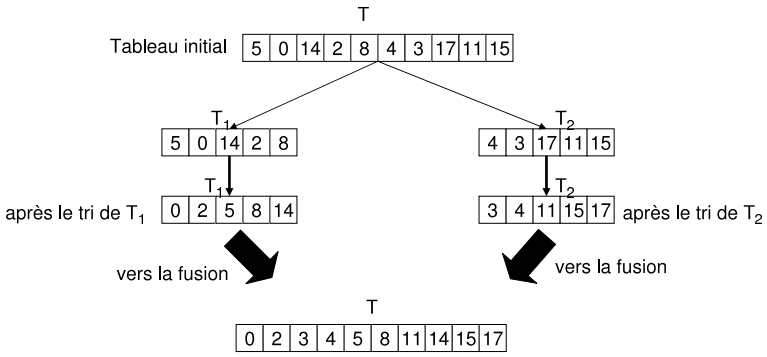

FIG. 5.5 – Un exemple de tri par fusion

Telles que définies ici, les fusions ne sont pas faites en place : un espace mémoire auxiliaire de taille n est alloué en début et laisser en fin pour réaliser les fusions [1].

Nous avons utilisé l'ordonnancement par vol de travail (voir chapitre 3, section 3.4.2.1) et l'algorithme de fusion parallèle adaptative proposé à la section précédente pour réaliser l'algorithme adaptatif du tri parallèle par fusion. Lors de la découpe du tableau en deux parties, les tâches à réaliser sur ces deux parties seront volées dynamiquement par vol de travail. Une fois l'exécution de ces tâches terminée, l'algorithme de fusion parallèle adaptatif est appelé sur les deux parties triées. La découpe récursive du tableau est limitée au grain $\alpha \log n$; la constante α permet d'amortir le coût de synchronisation supposé constant [25].

Le travail de cet algorithme est $W^{tri\_fusion}(n) = 2.W^{tri\_fusion}(\frac{n}{2}) + W^{fusion}_{seq}(n)(1+o(1)) \simeq W^{tri\_fusion}_{seq}(n)$ et la profondeur $D^{tri\_fusion}(n) = D^{tri\_fusion}(\frac{n}{2}) + O\left(\log^2 n\right) = O\left(\log^3 n\right)$. Le temps d'exécution de l'algorithme de tri par fusion parallèle est donc équivalent à $\frac{W_s(n)}{p.\Pi_{ave}} + O\left(\frac{\log^3 n}{\Pi_{ave}}\right)$ donc asymptotiquement optimal par rapport à l'algorithme séquentiel.

5.4 La partition parallèle adaptative

Soit un tableau $T[0 \ldots n-1]$, et un élément quelconque appelé $pivot$ pris dans le tableau, et un prédicat $pred$ donné. Le but de l'algorithme de partition est d'arriver à la situation suivante :

[1] Il existe des algorithmes de fusion en place plus complexes, basés sur une découpe en $\sqrt{n}$ [45]

1. $pred(T[i], pivot)$ pour $i \in [0, k[$
2. $\neg pred(T[j], pivot)$ pour $j \in [k, n[$

Le principe de l'algorithme séquentiel est le suivant :

1. Parcourir T à partir de l'indice 0 et par indices croissants, jusqu'à trouver le premier élément tel que $pred(T[i], pivot)$ est vrai.
2. Parcourir T à partir de l'indice $n-1$ et par indices décroissants, jusqu'à trouver le premier élément tel que $\neg pred(T[j], pivot)$ est vrai .
3. Permuter $T[i]$ et $T[j]$.
4. Répéter ce processus jusqu'à ce que $j \leq i$.

Exemple 5.4.1. *Soit* $T = [10, 5, 2, 8, 20, 6, 32, 3, 7]$ *et* $pivot = 8 = T[3]$. *Supposons que le prédicat est la comparaison (*$(pred(T[i], pivot)) \equiv (T[i] < pivot)$ *). Alors*

$$T_{final} = [\textbf{\textit{7,5,2,3,6}}, 20, 32, 8, 10]$$

Une approche naïve de la parallélisation de cet algorithme consiste en quatre phases :

1. On découpe le tableau T en p parties égales (à un élément près), et chaque processeur applique l'algorithme séquentiel de la partition dans la partie qui lui été allouée.
2. Chaque processeur calcule dans sa partie le nombre d'éléments qui se trouvent à gauche du $pivot$ et le nombre d'éléments qui se trouvent à droite du $pivot$. Soit L_i et R_i ces deux nombres calculés par le processeur de numéro i en supposant que les processeurs sont numérotés de 1 à p. On suppose que $L_0 = 0$ (resp. $R_0 = 0$).
3. On applique l'algorithme du calcul des préfixes sur ces nombres permettant de calculer les tailles des calculs à faire dans l'étape 4. Cet algorithme renvoie $L'_0 = L_0, L'_1 = L_0 + L_1, \ldots, L'_p = L_0 + \ldots + L_p$ (resp. $R'_0 = R_0, R'_1 = R_0 + R_1, \ldots, R'_p = R_0 + \ldots + R_p$).
4. Le processeur P_i copie les éléments partitionnés qui se trouvent dans l'intervalle $[L_{i-1}, L_i[$ (resp. $[R_{i-1}, R_i[$) du tableau T dans un tableau intermédiaire T' dans l'intervalle $[L'_{i-1}, L'_i[$ (resp. $[R'_{i-1}, R'_i[$). Le tableau T' contient le tableau final partitionné.

Cet algorithme effectue un travail $W = W_{seq} + n\tau_{copie}$ où τ_{copie} est le coût de l'opération de copie. Cet algorithme parallèle fait plus d'opérations que l'algorithme séquentiel et en plus il ne partitionne pas les éléments sur place (car utilise un tableau intermédiaire pour stocker les éléments partitionnés). En plus du surcoût en nombre d'opérations, il n'est pas adapté pour les architectures dont le nombre ou la vitesse des processeurs physiques alloués à une application donnée peuvent varier en cours d'exécution.

Une autre approche naïve consiste à découper récursivement le tableau jusqu'à des blocs de taille $\log n$, puis de partitionner chaque bloc. Mais il faut ensuite réarranger les blocs pour obtenir une partition ordonnée, ce réarrangement introduit un surcoût en $\Theta(n)$.

Afin de rendre le surcoût en nombre d'opérations négligeable et adapter l'algorithme à la variation du nombre ou de la vitesse des processeurs physiques, dans cette section nous proposons une parallélisation originale et en place de la partition qui suit le schéma de couplage adaptatif.

Nous considérons le tableau T comme des juxtapositions de plusieurs blocs de tailles éventuellement différentes. Nous disons qu'un bloc a été traité lorsque tous ses éléments ont été visités par l'algorithme de partition. Nous désignons par B_g (resp. B_d) le bloc non traité situé dans la partie extrême gauche (resp. droite) du tableau T privé des blocs traités. Soit **local_partition**(B_g, B_d) la fonction séquentielle qui parcourt les deux blocs B_g et B_d jusqu'à ce qu'au moins tous les éléments d'un de ces deux blocs aient été visités. Elle range les éléments qui ne vérifient pas le prédicat (les éléments qui sont supérieurs par exemple) dans B_d et ceux qui vérifient le prédicat (les éléments qui sont inférieurs) dans B_g. Notre algorithme parallèle adaptatif pour la partition est alors le suivant.

Nous désignons par P_s le processus qui initie l'exécution de la partition et par P_v les autres processus qui font du vol de travail. P_s suit l'algorithme séquentiel de partition, en bénéficiant éventuellement des opérations anticipées par les autres processus voleurs ; ces opérations correspondent à des intervalles volés qui sont chaînés dans une liste L_v gérée par P_s.

Initialement, P_s démarre la partition séquentielle du tableau T sur l'intervalle $[0, n[$. Il extrait deux blocs non traités de taille $\alpha.\log n$ (opération **extract_seq**) : B_g à l'extrémité gauche et B_d à l'extrémité droite de l'intervalle qui lui reste à partitionner. La constante α est choisie pour rendre négligeable le coût de synchronisation [25].

• **Algorithme séquentiel pour un processus** P_s :

1. P_s travaille localement sur deux blocs de taille $\alpha.\log n$ en extrayant deux blocs non traités (B_g et B_d) à l'extrémité gauche et droite de l'intervalle qui lui reste à faire.
2. P_s exécute **local_partition**(B_g, B_d) jusqu'à ce que l'un des blocs au moins soit traité.
3. Soit alors m le nombre d'éléments de l'intervalle que P_s doit encore partitionner.
   - Si B_g a été traité et qu'il reste des blocs non traités, P_s extrait le bloc de taille $\alpha \log m$ juste à droite de B_g et repart à 2.
   - Si B_d a été traité et qu'il reste des blocs non traités, P_s extrait le bloc de taille $\alpha \log m$ juste à gauche de B_d et repart à 2.
   - Si B_g et B_d ont été traités et qu'il reste des blocs non traités, P_s revient à l'étape 1 en extrayant deux blocs B_g et B_d de taille $\alpha \log m$.
   - Sinon P_s va à l'étape 4.
4. Si L_v n'est pas vide, P_s dépile le premier intervalle volé dans la liste L_v. Si ce vol est terminé, il récupère l'information sur les blocs traités et repart à 4. Sinon, P_s se synchronise avec le processus P_v qui l'a volé le premier ; pour cela il attend éventuellement que P_v ait terminé son exécution en cours de **local_partition** (préemption faible).
   - Si P_v n'a pas de blocs non traités, P_s récupère l'information sur les blocs traités et repart à 4.
   - Si P_v a un seul bloc non traité, P_s récupère celui-ci et le met dans sa liste B_F des blocs à finaliser et repart à 4.

– Sinon si P_s a fini le travail de son intervalle gauche (resp. droit), il récupère la moitié du travail restant à faire par P_v à l'extrême gauche (resp. extrême droite) et repart à l'étape 1.

5. Réarrangement : Si il ne reste plus que des intervalles volés non traités à gauche (resp. à droite), P_s ne peut plus extraire de blocs à droite (resp. à gauche), les processus sont alors inactifs. Les processus déplacent les blocs de la liste B_F et les blocs non traités à des positions cohérentes par rapport au pivot. Si tous les blocs étaient traités, P_s s'arrête et l'algorithme se termine. Sinon, il reste au sein du tableau un unique intervalle restant à partitionner : P_s repart alors à l'étape 1 sur cet intervalle.

• **Algorithme parallèle pour les processus** P_v . Chaque processus possède deux intervalles, l'un à gauche l'autre à droite.

– Lorsqu'il est inactif, P_v choisit au hasard un processeur jusqu'à trouver un processus actif P_w sur lequel il reste des blocs à traiter. Il peut s'agir soit de P_s, soit d'un autre processus voleur. Soit q_g (resp. q_d) le nombre d'éléments restant à partitionner dans l'intervalle gauche (resp. droit) sur P_w.

1. P_v découpe chacun des deux intervalles restants à traiter sur P_w en deux parties de tailles respectives $q_g/2$ et $q_d/2$; il vole la partie droite I_g de l'intervalle gauche et la partie à gauche I_d de l'intervalle droit. P_v insère alors l'information sur l'intervalle volé dans la liste L_v juste après celle de sa victime P_w.
2. Soit m le nombre d'éléments restant à partitionner sur P_v (initialement, $m = q_g/2 + q_d/2$) ; P_v extrait de I_g (resp. I_d) le bloc B_g le plus à gauche (resp. B_d à droite) non traité et de taille $\alpha \log m$.
3. P_v applique **local_partition**(B_g, B_d) jusqu'à ce que l'un des blocs au moins soit traité.
4. Si P_w a envoyé un signal de synchronisation à P_v (préemption) ; P_v attend que P_w ait terminé la découpe de l'intervalle qu'il lui reste à partitionner (cf étape 4 de P_s) et repart à 2.
5. Sinon, il continue sa partition en extrayant de nouveaux blocs B_g et/ou B_d (identiquement aux étapes 1, 2, 3 et 4 de P_s).

On remarque que si il y a un seul processeur, l'algorithme exécute une partition séquentielle ; il effectue un travail $W_{seq}(n)$ (nombre de comparaisons et permutations) identique. Dans la suite le surcoût de réarrangement, qui limité au bloc B_F, n'est pas considéré ; le travail arithmétique sur p processeurs est aussi $W_p(n) = W_{seq}(n)$. Le théorème suivant montre alors l'optimalité asymptotique de cet algorithme de partition en prenant en compte le surcoût d'ordonnancement.

Théorème 5.4.1. *Soit $W_{seq}(n)$ le travail séquentiel de la partition. En négligeant le surcoût de réarrangement alors, sur p processeurs de vitesse moyenne Π_{ave} et pour n suffisamment grand, le temps $T_p(n)$ est asymptotiquement optimal et vérifie*

$$T_p(n) = \frac{W_s}{p.\Pi_{ave}} + O\left(\frac{\log^2 n}{\Pi_{ave}}\right).$$

Démonstration. L'exécution est structurée par le processus P_s en étapes successives ; d'abord partitionnement partiel du tableau. Puis lorsque tous les vols sont terminés, les blocs de la liste B_F et les blocs

non traités sont réarrangés (déplacement des éléments) pour former l'intervalle suivant à partitionner de taille n' égale au nombre d'éléments non classés dans B_F et des blocs non traités. L'étape suivante correspond alors au partitionnement (récursif séquentiel) de ces n' éléments restants à partitionner. A chaque vol correspond au plus un bloc ajouté dans B_F ; la taille de chaque bloc dans B_F est majorée par $O(\log n)$.

Soit $D^{(1)}$ la profondeur de la première étape de partition sur un nombre non borné de processeurs identiques. De part la découpe récursive par moitié lors de chaque vol et la taille logarithmique de chaque bloc extrait pour **local_partition**, avec une infinité de processeurs, chaque processeur exécute une seule fois **local_partition** et traite au moins un bloc, au plus deux. On a donc $D^{(1)} = O(\log n)$ et par suite la profondeur de chacune des étapes est majorée par $O(\log n)$. De plus, le nombre de blocs dans B_F est au plus la moitié du nombre de blocs total : donc $n' \leq n/2$ et le nombre total d'étapes est donc majoré par $\log n$.

Considérons maintenant l'exécution sur p processeurs de la première étape, qui est de profondeur $O(\log n)$. Par le théorème 3.4.2, le nombre de vols durant cette étape est donc $O(p \log n)$. Comme la profondeur de chacune des étapes est majorée par $O(\log n)$, on en déduit qu'il y'a au plus $O(p \log^2 n)$ vols. Finalement, le théorème 3.4.2 permet de déduire le temps d'exécution $T_p(n)$ sur les p processeurs de vitesse Π_{ave} : $T_p(n) = \frac{W_s(n)}{p.\Pi_{ave}} + O\left(\frac{p \log^2 n}{\Pi_{ave}}\right)$. Pour $p = o\left(\frac{\sqrt{n}}{\log n}\right)$, le temps d'exécution est alors équivalent à $\frac{W_s(n)}{p.\Pi_{ave}}$ ce qui est optimal.

❑

En pratique, p est fixé ; l'algorithme adaptatif de partition est alors asymptotiquement optimal. Dans la section suivante, il est utilisé pour paralléliser le tri introspectif.

Dans le reste de ce chapitre, pour simplifier la présentation, nous supposons que le prédicat à vérifier est : *si l'élément considéré est strictement inférieur au pivot choisi*

5.5 Le tri rapide introspectif parallèle

Le tri introspectif est une amélioration proposée par David Musser [64] de l'algorithme de tri rapide (*Quiksort*). L'idée est de combiner le tri rapide au tri par tas (*heapsort*) pour que la complexité de l'algorithme résultant reste $O(n \log n)$. L'algorithme réalise un tri rapide classique, au cours duquel le nombre total d'appels récursifs est compté et lorsqu'on réalise $\lceil \log n \rceil$ appels récursifs en séquence, le tri par tas remplace le tri rapide sur le reste de la séquence à trier.

Le tri introspectif (comme le tri rapide) est basé sur une partition en place : un élément pivot e (pseudo-médiane) est choisi qui est utilisé pour réarranger en temps $\Theta(n)$ le tableau en deux parties, le premier contenant les éléments inférieurs à e, le deuxième ceux supérieurs.

La parallélisation adaptative de la partition est directement utilisée pour effectuer chacune des partitions du tri introspectif. Après chaque partition adaptative, le sous-tableau contenant

les éléments inférieurs d'une part et ceux supérieurs d'autre part peuvent être triés en parallèle.

L'algorithme 5.5.1 décrit l'algorithme parallèle adaptatif de tri en Athapascan/Kaapi [37]. L'écriture est similaire à celle de l'implantation de la STL, les deux seules différences étant l'appel à la partition parallèle adaptative (ligne 11) et la parallélisation potentielle des éléments supérieurs au pivot (ligne 12).

Deux seuils sont utilisés dans la boucle principale. Le paramètre $depth\_limit$, clef de l'algorithme séquentiel introspectif est initialisé à $\log n$ dans l'appel principal et permet de limiter le travail du tri introspectif à $O(n \log n)$. Si $depth\_limit == 0$, on fait appel à l'algorithme du tri par tas (ligne 4) qui a toujours une complexité en $O(n \log n)$. Le seuil $grain$ permet de limiter la parallélisation récursive parallèle à des tableaux de taille supérieure à $\alpha \log n$.

Sur la ligne 9 de l'algorithme, le pivot choisi est la médiane des trois valeurs (le premier élément, l'élément du milieu et le dernier élément du tableau en cours de tri). Sur la ligne 11 tous les processeurs inactifs exécutent l'algorithme adaptatif parallèle de la partition à partir du pivot fourni. Puis sur la ligne 12, une tâche est créée pour le tri en parallèle des éléments supérieurs au pivot.

Algorithme 5.5.3

```
1   atha_intro_sort_adapt (a1 : :remote<InputeIterator> first,
                           a1 : :remote<InputeIterator> last, size_t depth_limit ) {
2       int grain = α × depth_limit ;
3       while( last − first > grain) {
4           if(depth_limit == 0) return heap_sort(first, last, last) ;
5           depth_limit = depth_limit − 1 ;
6           typedef typename std : :iterator_traits<InputIterator> : :difference_type diff_t ;
7           typedef typename std : :iterator_traits<InputIterator> : :value_type value_t ;
8           const diff_t sz = last − first ;
9           value_t median = value_t( std : :__median(first, first + sz/2, first + sz − 1)) ;
10          a1 : :remote<InputeIterator> split ;
11          split = adaptive_parallele_partition(first, last, less_than_median<value_t>(median)) ;
12          a1 : :Fork< atha_intro_sort_adapt <InputIterator> >()(split, last, depth_limit) ;
13          last = split ;
14      } ;
15      std : :sort(first, last) ; // tri séquentiel si (last − first < grain)
16  } ;
```

Algorithme 5.5.1: Algorithme parallèle de tri introspectif

5.6 Expérimentations

Nos expérimentations ont été faites sur deux machines à mémoire partagée NUMA : une Intel Itanium-2 à 1.5GHz avec 31GB de mémoire composée de deux noeuds quadri-coeurs ; une AMD Opteron composée de 8 noeuds bi-coeurs. Les algorithmes ont été implantés avec l'interface adaptative générique ; la constante α a été fixée à $\alpha = 100$ sur les deux machines après calibrage expérimental.

Les premières expérimentations consistent à trier un tableau de 10^8 doubles (les données sont tirées aléatoirement) en faisant varier le nombre de processeurs utilisés (de 1 à 8 pour itanium et 1 à 16 pour AMD). Les tableaux 5.1 et 5.2 donnent les temps d'exécution obtenus par les deux algorithmes (tri par fusion et quicksort) sur les machines Itanium et AMD. Nous avons réalisé dix exécutions et pour chaque test nous avons pris le minimum, le maximum et la moyenne ; les résultats sont très stables.

Nous remarquons dans le tableau 5.1 que les deux algorithmes se comportent très bien sur 1 à 8 processeurs, et qu'ils se comportent moins bien sur 9 à 16 processeurs. Ceci est dû à la contention d'accès à la mémoire sur cette machine NUMA où chaque bi-processeurs partage la même mémoire. D'ailleurs, lorsque l'on augmente artificiellement le grain de la comparaison à un temps arithmétique de 1μs (en choisissant alors $\alpha = 1$), l'accélération obtenue est linéaire : jusqu'à 15,2 pour 16 processeurs.

On remarque aussi que l'algorithme de tri introspectif (non stable) est meilleur que le tri par fusion. Dans le tableau 5.2, nous remarquons que les deux algorithmes se comportent très bien, avec de très bonnes accélérations.

Dans le tableau 5.3, des processus de charges additionnelles sont injectés pour perturber l'occupation de la machine et simuler le comportement d'une machine réelle, perturbée par d'autres utilisateurs. Par souci de reproductibilité, chaque expérience sur $p \leq 8$ processeurs est perturbée par $8 - p + 1$ processus artificiels sur la machine itanium-2 et par $16 - p + 1$ sur la machine AMD. Ainsi, l'un des exécuteurs a sa vitesse divisée par 2, ce qui conduit à $p.\Pi_{ave} \leq (p - 0,5).\Pi_{max}$ où π_{max} est la vitesse d'un coeur dédié. Si T_s est le temps séquentiel, le temps parallèle optimal théorique serait donc $\tilde{T_p} = \frac{T_s}{p-0,5}$; la dernière ligne du tableau reporte ce temps pour $T_s = 22,45$s. Conformément à la théorie, on constate les performances stables du tri adaptatif sur cette machine perturbée ; de plus les temps obtenus sont relativement proches à moins de 30% de l'estimation théorique $\tilde{T_p}$.

	Tri rapide parallèle adaptatif				Tri par fusion parallèle			
	p=1	p=4	p=8	p=16	p=1	p=4	p=10	p=13
Minimum	22,45	5,51	3,05	2,60	24,50	6,44	4,15	3,71
Maximum	22,56	5,58	3,30	2,82	27,15	6,90	4,42	3,76
Moyenne	22,51	5,54	3,14	2,65	25,56	6,68	4,18	3,72

TAB. 5.1 – Temps d'exécution tri rapide versus tri par fusion parallèle sur AMD opteron 16 coeurs.

	Tri rapide parallèle adaptatif			Tri par fusion parallèle		
	p=1	p=4	p= 8	p=1	p=4	p= 8
Minimum	60,2562	13,3525	7,06782	75,6164	16,5904	8,817766
Maximum	62,087	13,4384	7,77093	75,7446	17,0557	8,92339
Moyenne	60,8665	13,4098	7,1193	75,7019	16,9006	8,8881

TAB. 5.2 – Temps d'exécution tri rapide versus tri par fusion sur IA64 8 coeurs.

	Tri rapide parallèle adaptatif perturbé								
	p=1	p=2	p=4	p=6	p=7	p=8	p=10	p= 12	p=16
Minimum	22,4562	14,2576	6,63771	4,58461	3,83735	3,44868	2,74097	2,74097	2,73606
Maximum	43,9689	17,6414	7,6414	4,92626	4,06878	3,68123	3,25123	3,02834	2,95637
Moyenne	43,6168	15,075	7,31	4,71937	3,91663	3,52036	3,12081	2,84938	2,62112
$\tilde{T}_p$ optimal	[22,45 ;44,9]	14,96	6,41	4,08	3,45	2,99	2,36	1,95	1,45

TAB. 5.3 – Temps d'exécution du tri rapide adaptatif parallèle perturbé sur la machine AMD opteron et comparaison à la borne inférieure.

La figure 5.6 montre la performance de notre algorithme de fusion adaptative parallèle par rapport à l'algorithme de fusion parallèle basé sur une découpe statique en p parties de la bibliothèque MCSTL [84]. Elle montre l'accélération obtenue par notre algorithme sur 8 processeurs de la machine AMD opteron en faisant varier la taille (n) de la séquence de données à trier. Dans les deux cas l'accélération augmente avec n et est limité lorsque n est petit. Cependant, pour $n > 10^5$ on obtient par notre algorithme une accélération supérieure à 1 par rapport au temps séquentiel, tant disque pour la fusion parallèle de MCSTL, l'accélération dévient supérieure à 1 lorsque $n > 316228$. On observe sur cette figure que notre algorithme est au moins 25% supérieur pour n assez grand à la fusion parallèle statique. Ceci s'explique par le mécanisme d'adaptation qui garantit qu'un processeur suit l'exécution séquentielle et qui adapte automatiquement la charge en fonction de la disponibilité des processeurs.

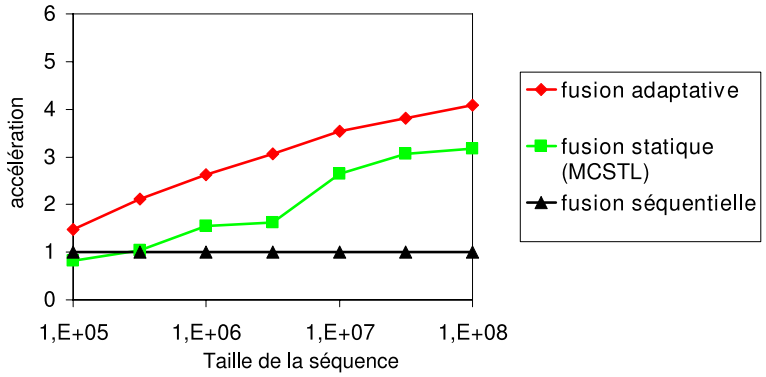

FIG. 5.6 – Comparaison de la fusion adaptative avec la fusion statique en fonction de la taille de la séquence sur 8 processeurs de la machine AMD Opteron

La figure 5.7 montre la performance de notre algorithme de tri par fusion adaptative parallèle

par rapport à l'algorithme de tri par fusion parallèle. Elle montre l'accélération obtenue par notre algorithme sur 8 processeurs de la machine AMD opteron en faisant varier la taille (n) de la séquence de données à trier. Notre algorithme de tri par fusion parallèle repose sur la fusion parallèle adaptative tant disque l'algorithme de tri par fusion parallèle de la bibliothèque MCSTL repose sur la fusion parallèle statique. On observe que notre algorithme est au moins 25% plus rapide pour n assez grand. Ceci s'explique de la même manière que l'explication donnée pour la figure 5.6.

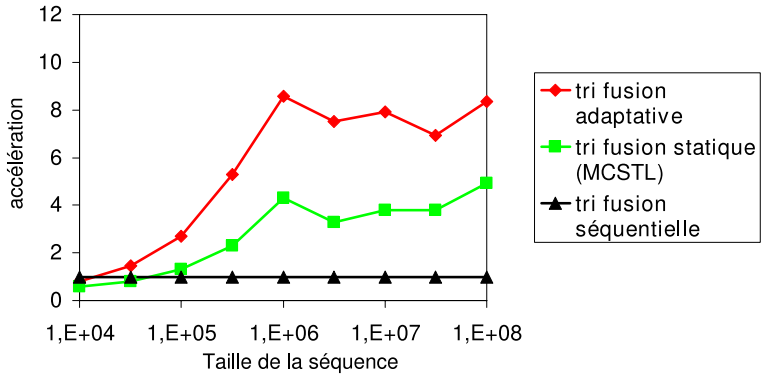

FIG. 5.7 – Comparaison du tri par fusion adaptative avec le tri par fusion statique en fonction de la taille de la séquence de la machine AMD Opteron

La figure 5.8 montre la performance sur un tableau de taille $n = 10^8$ de notre algorithme de tri par fusion adaptative parallèle par rapport à l'algorithme de tri par fusion parallèle. On observe que notre algorithme est au moins 25% plus rapide pour n assez grand. Ceci s'explique aussi de la même manière que l'explication donnée pour la figure 5.6.

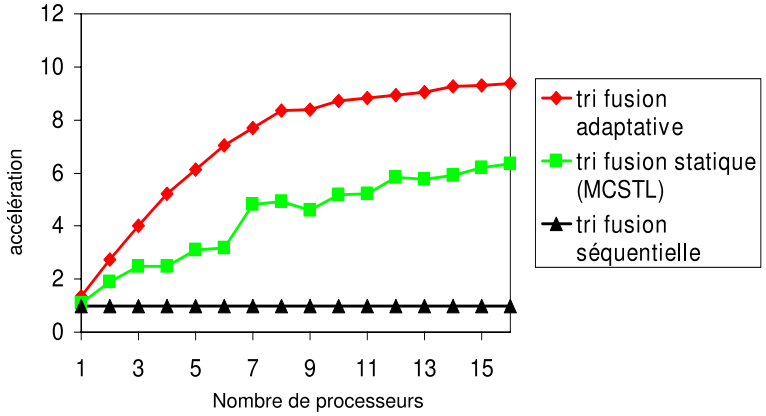

FIG. 5.8 – Comparaison du tri par fusion adaptative avec le tri par fusion statique en fonction du nombre de processeurs de la machine AMD Opteron

5.7 Conclusions

Dans ce chapitre, nous avons proposé une parallélisation adaptative des algorithmes de tri par fusion et de tri introspectif, les performances de ce dernier étant bien meilleures en séquentiel. Nous avons donné des analyses théoriques de leurs temps d'exécution.

Grâce au couplage d'un algorithme séquentiel avec un algorithme parallèle à grain fin ordonnancé par vol de travail, des garanties théoriques de performance sont obtenues par rapport au temps séquentiel sur des machines à mémoire partagée même lorsqu'elles sont utilisées en concurrence par d'autres applications.

Les expérimentations menées montrent le bon comportement des algorithmes même sur des machines dont la charge des processeurs est perturbée ce qui est particulièrement intéressant en contexte multi-utilisateurs. Les perspectives sont d'une part de compléter les études expérimentales sur la tolérance aux variations de charge en contexte perturbé : un point difficile concerne la reproductibilité des expérimentations. D'autre part, une autre perspective est de compléter ce travail par une analyse des défauts de cache [59], avec la comparaison à la parallélisation adaptative d'algorithmes de tri inconscients à la hiérarchie mémoire (cache-oblivious).

Dans les deux chapitres précédents, nous avons présenté des algorithmes adaptatifs pour le calcul des préfixes, la fusion de listes triées, partition, et tris, nous allons présenter dans le chapitre suivant, un schéma générique permettant de construire des algorithmes adaptatifs.

Chapitre 6

Schéma générique des algorithmes parallèles adaptatifs

Les techniques présentées dans le chapitre 3 ne prennent pas en compte l'adaptation automatique de la granularité et la minimisation du surcoût arithmétique. Dans ce chapitre, nous allons présenter une extension de la technique adaptative proposée par Daoudi&al [26] qui permet de contrôler la granularité du parallélisme en cours d'exécution. Elle est basée sur le couplage d'un algorithme parallèle à grain fin minimisant la profondeur qui est ordonnancé par vol de travail et d'un algorithme minimisant le nombre d'opérations (travail). La génération du parallélisme n'est effectuée qu'en cas d'inactivité d'un processeur. Elle permet d'obtenir une garantie de performances en temps écoulé par rapport à une exécution de l'algorithme séquentiel dans les mêmes conditions. Puis à partir de cette technique, nous proposons une interface générique permettant le développement des algorithmes parallèles adaptatifs.

6.1 Hypothèses

Nous utilisons le couplage de deux algorithmes, l'un séquentiel f_{seq}, et l'autre parallèle f_{par}, pour résoudre le même problème. Nous supposons que l'algorithme séquentiel est tel qu'à tout instant de l'exécution, la séquence des opérations qui termine l'algorithme peut être réalisée par un autre algorithme, parallèle à grain fin (*e.g.* de type découpe récursive). L'opération qui consiste à extraire une partie du calcul séquentiel pour la traiter (par un autre algorithme séquentiel du même type) est appelée "extraction de parallélisme" (**extract_par**) ; après traitement, les résultats d'une part de la partie qui a subi l'extraction et d'autre part du travail extrait sont ensuite fusionnés (opération **join** ou **finalize**).

Plus précisément, étant donné un algorithme séquentiel (resp. parallèle), le résultat r de son évaluation sur une entrée x est notée $r = f_{seq}(x)$ (resp. $f_{par}(x)$). Nous faisons l'hypothèque qu'une entrée x possède une structure de liste avec un opérateur de concaténation noté $\sharp$ et qu'il

existe un opérateur (non nécessairement associatif) de fusion $\oplus$ des résultats.

A tout moment de l'évaluation de $f_{seq}(x)$, il est possible de découper x en $x_1 \sharp x_2$. Le résultat calculé par l'algorithme parallèle est alors $f_{par}(x) = f_{seq}(x_1) \oplus f_{seq}(x_2)$. Nous supposons que les deux résultats $f_{seq}(x)$ et $f_{par}(x)$ sont équivalentes s'ils sont proches au sens d'une métrique donnée.

Dans le cadre restreint des homomorphismes de liste [12, 14, 40, 44, 13], cette hypothèse s'écrit alors : $f_{seq}(x \sharp y) = f_{seq} \oplus f_{seq}(y)$. Cependant, il existe des techniques qui permettent de construire un algorithme parallèle pour des problèmes qui ne sont initialement pas des homomorphismes de liste. Par exemple dans [22], le problème du calcul du segment d'entiers de somme maximale est ainsi transformé en l'évaluation d'un homomorphisme suivi d'un filtrage, et ce au prix d'une augmentation du nombre d'opérations.

A partir de cette hypothèse, nous donnons dans la section suivant les éléments qui seront utilisés dans notre schéma générique adaptatif.

6.2 Fonctions de base virtuelle du schéma

Avant de présenter l'algorithme du schéma générique adaptatif, nous définissons dans cette section les éléments qui doivent être spécialisés pour son utilisation ; autrement dit les structures et fonctions virtuelles qu'il manipule. Le schéma que nous proposons fait abstraction de tout type d'architecture parallèle, il est basé sur des structures de données et fonctions abstraites qui peuvent être instanciées sur un problème spécifique. Nous détaillons ci-dessous chacune des structures de données ou fonctions du schéma proposé :

- Un descripteur de travail **WorkAdapt** : c'est une structure de donnée qui représente un bloc de travail (un flot d'instructions à exécuter par exemple). Cette structure peut contenir par exemple le type de calcul qui doit être réalisé, l'information pour accéder aux données d'entrées, l'endroit où stocker les résultats, l'information sur la progression du calcul en cours (instructions courantes par exemple), des verrous permettant d'éviter des conflits lors de l'extraction ou lors de l'écriture des résultats (cohérence), etc.

- Un descripteur de travail volé **stolenwork** : c'est une structure de donnée qui représente le travail volé. Elle peut contenir comme la structure **WorkAdapt** le type de calcul qui doit être réalisé, l'information pour accéder aux données d'entrées, l'endroit où stocker les résultats, l'information sur la progression du calcul en cours (instructions courantes par exemple), des verrous permettant d'éviter des conflits lors de l'extraction ou lors de l'écriture des résultats (cohérence), etc.

- Une fonction **extract_seq** : elle consiste à extraire localement une petite fraction du travail séquentiel restant pour la traiter par un algorithme séquentiel optimal en évitant tous les surcoûts liés au parallélisme (synchronisations, arithmétique, ...). Elle retourne vrai si

l'extraction est possible (c'est à dire s'il reste encore du calcul séquentiel à faire) sinon elle retourne faux.

- Une fonction **extract_par** : elle consiste à extraire une partie du travail séquentiel en cours pour la traiter en parallèle (par un autre algorithme séquentiel éventuellement de type différent). Elle est utilisée pour les vols de travail. Elle retourne vrai si l'extraction est possible (dans ce cas le vol a été une réussite) sinon faux (dans ce cas le vol a été un échec).

- Une fonction **local_run** : elle exécute le meilleur algorithme séquentiel sur le bloc de travail extrait par la fonction **extract_seq** sans aucun surcoût de synchronisation ou arithmétique.

- Une opération de fusion **join** : elle consiste à fusionner les résultats de la partie qui a subi l'extraction et du travail extrait volé (stolen work).

- Une opération **finalize** : elle consiste à éventuellement mettre à jour (finaliser) le travail réalisé dans la partie volée qui été extraite.

- Une opération de saut **jump** : elle permet de faire un saut sur le travail déjà réalisé dans la partie extraite qui a été volée et de récupérer le travail restant encore à faire dans cette partie pour l'initialiser comme nouveau travail et de l'exécuter séquentiellement.

- Une boucle **extract_next_macro_work** : elle permet d'amortir le surcoût lié au parallélisme. Un pas (*macro-step*) dans cette boucle est un ensemble de calculs pouvant être faits en parallèle.

6.3 Schéma adaptatif générique

Dans toute la suite, nous utilisons l'ordonnancement dynamique par vol de travail basé sur le principe de travail d'abord [35] pour ordonnancer l'exécution de l'algorithme parallèle sur des processeurs physiques. Cet ordonnancement est basé sur le couplage dynamique de deux algorithmes spécifiés pour chaque fonction du programme, l'un séquentiel, l'autre parallèle récursif à grain fin ordonnancé par vol de travail [38, 26].

Au niveau de l'application, chaque processeur physique exécute un processus unique, appelé **exécuteur**, qui exécute localement un programme séquentiel. A tout moment de l'exécution d'un programme, nous supposons qu'il est possible d'extraire par un autre exécuteur (appelé voleur) une partie du travail en cours qui est à la charge d'un exécuteur actif (appelé victime ou volé) par l'opération **extract_par** sans interrompre la victime. Chaque exécuteur maintient localement une liste chaînée L_v de pointeurs de ses travaux volés (un pointeur n'est inséré dans la liste que s'il y a eu du vol). Cette liste distribuée permet de diminuer le surcoût de parallélisme (création de tâches, diminution des surcoûts dû à la récursivité). Nous supposons que le travail à réaliser par chaque exécuteur est un flot d'instructions séquentielles et qu'il est délimité par l'instruction de début du flot I_d et l'instruction de fin du flot $I_\perp$.

Initialement un exécuteur P_s commence l'exécution du programme à effectuer en créant tout d'abord le descripteur de travail à effectuer $w = [I_d, I_\perp[$ (où w représente le travail à réaliser, I_d l'instruction de début de w et $I_\perp$ l'instruction de fin), en donnant la possibilité d'extraire du parallélisme sur le travail qui reste à effectuer. Lorsqu'un exécuteur P_v devient inactif, il devient voleur et cherche à participer au travail restant à faire sur P_s. Pour cela il exécute l'opération **extract_par** sur w qui lui renvoie le travail $[I_k, I_\perp]$ avec $k > d$ et construit un nouveau travail à effectuer qui est décrit par $w' = [I'_{k'}, I'_{\perp'}[$. Lors de l'opération **extract_par**, un pointeur sur le descripteur du travail volé par P_v est inséré dans la liste chaînée des descripteurs de pointeurs de travaux volés sur le travail en cours sur P_s. La figure 6.1 illustre cette opération d'extraction, initialement la liste est vide. La figure 6.2 montre comment les descripteurs des travaux volés sont chaînés entre eux dans la liste.

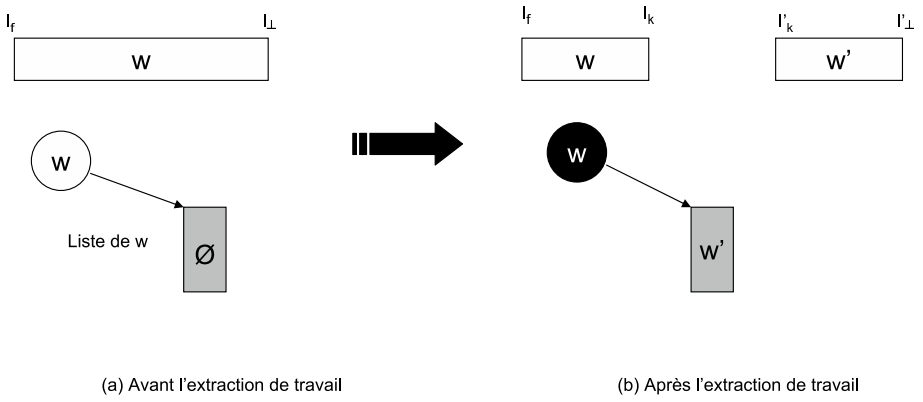

FIG. 6.1 – Modification de travail lors de l'opération **extract_par**. Le descripteur de travail dans un cercle blanc indique qu'aucun vol n'a été effectué sur lui, par contre quand il est dans un cercle noir, cela indique qu'il a été victime au moins d'un vol.

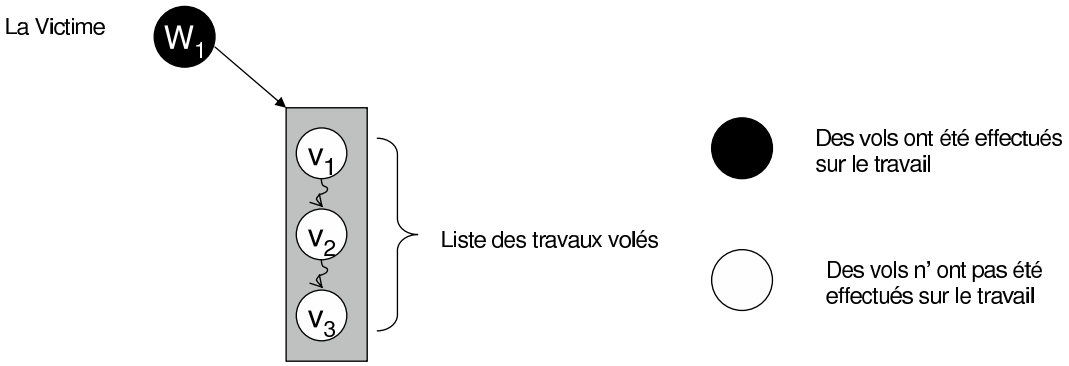

FIG. 6.2 – La liste des travaux volés. Les descripteurs v_1, v_2 et v_3 sont les descripteurs des travaux qui ont été volés sur le descripteur w_1.

Le voleur P_v commence l'exécution du nouveau travail décrit par w' pendant que la victime P_s continue l'exécution séquentielle sur w réduit à $[I_d, I_k[$ par l'opération **extract_par**. P_v se comporte comme P_s en créant une liste pour ses propres voleurs. Notez que w' est différent de

l'intervalle $[I_k, I_\perp[$, puisqu'il comprend le travail parallèle, nécessaire à réaliser qui aurait été un travail séquentiel dans l'intervalle initial.

L'algorithme 1 généralise l'exécution de l'algorithme générique adaptatif par les exécuteurs victimes et voleurs. Tout d'abord, lorsqu'un exécuteur devient inactif, il devient voleur et cherche à participer au travail restant à faire sur un autre exécuteur actif. Pour cela, il choisit aléatoirement un autre exécuteur (victime) qui est actif et effectue une opération d'extraction d'une fraction du travail restant à faire grâce à l'opération **extract_par** (ligne 1). Puis ensuite, l'exécuteur entre dans une boucle (lignes 9-11) dont chaque pas dans cette boucle consiste à extraire une partie du travail local par l'opération **extract_seq** et à exécuter celle-ci par l'algorithme séquentiel optimal appelé par la fonction **local_run**. Dans toute la suite nous appelons cette boucle interne la boucle **nano-loop**. Lorsque l'exécuteur sort de cette boucle, donc a terminé le travail qu'il a à faire, deux cas se présentent :

- il a été préempté par sa victime : alors il reçoit d'elle un signal de fusion des résultats (ligne 12), une fois ce signal reçu, il envoie à sa victime un signal (ligne 13) permettant à ce dernier de faire un saut sur le travail déjà réalisé et d'aller continuer sur le travail restant à faire et il finalise éventuellement ce travail déjà fait grâce à celui effectué par sa victime (ligne 14).
- sa liste contient des travaux volés (ligne 16) : il préempte alors le premier voleur de sa liste, enchaîne la liste du voleur à la tête de sa liste (ligne 17). Si le voleur n'a pas terminé (ligne 19), il récupère les résultats calculés par celui-ci dont il a besoin, lui envoie un signal de fusion (ligne 20), attend qu'il ait reçu le signal (lignes $21-23$). Puis il effectue un saut sur le travail déjà réalisé par le voleur (ligne 24), ensuite fait la fusion (ligne 24) de ses résultats avec ceux du voleur lui permettant de continuer en séquentiel sur le travail restant. Si le travail enlevé de la tête de sa liste est terminé (le voleur a déjà fini ce qu'il a volé avant qu'il soit préempté), si éventuellement ce travail doit subir un autre traitement pour être terminé complètement (finalisation), l'exécuteur prépare une tâche de finalisation sur ce travail qui sera prête à être volé par un autre exécuteur(ligne 28), puis il fusionne le résultat calculé par son voleur avec son résultat (ligne 28).

La boucle externe (lignes $7-31$) qui consiste à exécuter localement le travail et à préempter éventuellement les calculs, est appelée la boucle **micro-loop**. Sur la figure 6.3 par exemple, le descripteur de travail initial était égal à $[I_1, I_\perp[$, après plusieurs vols ($v_1, v_2, v_3, v_{11}, v_{12}, v_{31}$ et v_{32}) ce descripteur a été réduit à w_1 , et l'ordre séquentiel des descripteurs est $w_1, v_1, v_{11}, v_{12}, v_2, v_3, v_{31}$ et v_{32}. Dans l'algorithme les accolades indiquent les sections critiques.

Généralement, le travail extrait w' de P_v diffère du descripteur volé $[I_k, I_\perp[$. De plus, une opération de fusion sur $[I_d, I_k[$ et $[I'_{k'}, I'_{\perp'}[$ peut être appelée pour compléter le résultat r de $[I_1, I_\perp[$. Cette opération de fusion peut être vide pour certains algorithmes. Cependant dans le cas général, elle consiste en deux calculs non bloquants permettant de compléter $[I_d, I_k[$ (resp. $[I'_{k'}, I'_{\perp'}[$) basés sur des opérations **join** (resp. **finalize**) exécutées par l'exécuteur victime (resp. l'exécuteur voleur). La figure 6.4 montre ces opérations de fusion. Chaque opération correspond à un flot séquentiel d'instructions, qui peut être exécuté en séquentiel de manière non bloquante si aucun vol n'a eu lieu. Les deux opérations de fusion sont exécutées en parallèle. Les opérations de fusion **join** et **finalize** qui seront exécutées dépendent des derniers résultats calculés par l'exécuteur voleur (P_v) et victime (P_s). Lorsque l'exécuteur victime P_s complète

l'instruction I_{k-1}, deux cas apparaissent. Soit le travail volé est terminé et le processeur victime P_s prépare une tâche de finalisation sur le travail volé terminé et qui sera prête à être volé par un autre exécuteur, puis ensuite P_s fusionne son résultat avec celui de P_v en exécutant l'opération **join**. Soit le dernier voleur P_v de P_s a complété l'instruction I'_{l-1} avec $k' < l < \perp'$ et est en train d'exécuter l'instruction I'_l , l'instruction de début de $w' = [I'_l, I'_{\perp'}[$. La victime P_s préempte alors le voleur P_v après l'exécution de I'_l ; après une instruction de synchronisation, P_s envoie son résultat r_s à P_v et récupère le résultat r_v de P_v. P_s effectue un saut sur le travail $w' = [I'_l, I'_{l-1}[$ partiellement réalisé par P_v et exécute l'opération **join** pour compléter $w = [I_l, I_\perp[$ à l'aide du résultat r_v et P_v exécute **finalize** pour finaliser $[I'_{k'}, I'_{\perp'}[$ à l'aide du résultat r_s (voire la figure 6.4 qui illustre ces opérations).

Chaque victime P_s gère sa liste de pointeurs des descripteurs localement ; lorsque P_s termine l'exécution de l'opération **join** , il accède à la tête de sa liste, enlève le descripteur de travail correspondant, envoie un signal de fusion (Signal merge) à l'exécuteur exécutant ce descripteur. Ce dernier recevant le signal complète sa dernière instruction, enchaîne sa liste des descripteurs volés à la tête de celle de sa victime et vide sa liste ; ensuite il commence l'exécution de **finalize**. La figure 6.5 montre l'enchaînement de la liste des descripteurs volés de P_v à la tête de la liste des descripteurs volés de P_s lors de l'opération de préemption. P_s n'est interrompu que si il exécute une opération de préemption sur un exécuteur voleur : il attend alors au plus la terminaison du **lacal_run** courant en cours d'exécution sur P_v.

```
1  Init  w ← { extract_par(v) }
2  //  v est le travail de la victime, w est le travail volé
3  // v initialise w.SignalMerge à false
4  // v initialise w.SignalJump à false
5  w.thiefList ← ∅
6  // Micro-loop
7  while ( { w ≠ ∅ } ) or ( { w.thiefList ≠ ∅ } ) do
8      // Nano-loop :
9      while ( { w.extract_seq() } ) do
10         ← w. local_run() ;
11     end
12     if { w.SignalMerge = true } then
13         { w.SignalJump ← true }
14         w.workToFinalize.finalize() ;
15     end
16     if { w.thiefList ≠ ∅ } then
17         w_s ← thiefList.head
18         { w.thiefList.insertHead( w_s.thiefList ) ; w_s.thiefList ← ∅ }
19         if ({ w_s ≠ ∅ }) then
20             { w_s.SignalMerge ← true }
21             while ( { w_s.SignalJump ≠ true } ) do
22                 yield
23             end
24             w.jump() ; w.join(w_s.result) ;
25         end
26         else
27             w.jump() ;
28             fork(w_s.workToFinalize) ; w.join(w_s.result) ;
29         end
30     end
31 end
32 { w.SignalJump ← true }
33 goto 1
```

Algorithm 1: L'algorithme du schéma adaptatif.

6.4 Analyse théorique

L'algorithme précédemment présenté a un surcoût dû à la gestion de la synchronisation nécessaire lors des accès à un descripteur de travail $[I_1, I_k[$. En effet, à la fois l'exécuteur victime et voleur peuvent accéder simultanément à un même descripteur de travail $[I_1, I_k[$ lors du vol (Puisque pendant que l'exécuteur victime ait en train d'extraire un bloc d'instructions localement dans le descripteur $[I_1, I_k[$ un exécuteur voleur peut accéder à ce même descripteur au même moment $[I_1, I_k[$ pour extraire une partie du travail en cours et la traiter en parallèle).

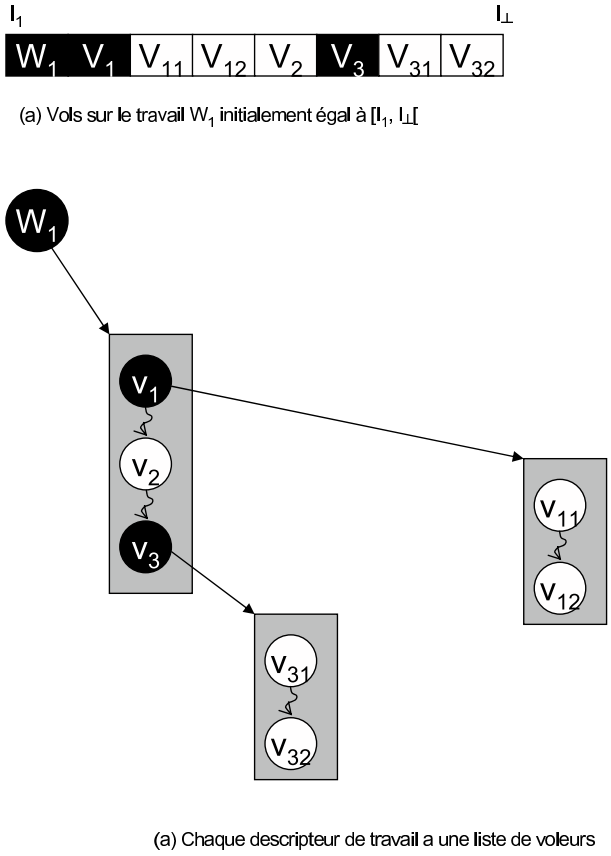

FIG. 6.3 – Illustration des listes distribuées. Les descripteurs v_1, v_2 et v_3 ont été volés sur w_1 ; les descripteurs v_{11} et v_{12} ont été volés sur v_1 ; et les descripteurs v_{31} et v_{32} ont été volés sur v_3.

Le code de l'algorithme séquentiel doit donc être analysé afin d'identifier les sections critiques de code qui doivent être exécutées en exclusion mutuelle vis-vis des modifications du descripteur de travail $[I_1, I_k[$. La réalisation effective de l'exclusion peut reposer sur l'utilisation des primitives générales de synchronisation (verrou, sémaphore, compare_and_swap, ...).

Ce surcoût de maintien de cohérence peut s'avérer important vis-vis des opérations de calcul et masquer le gain en nombre d'opérations arithmétiques grâce à l'utilisation d'un algorithme séquentiel optimisé à la place d'un algorithme parallèle. Dans ce cas, il est alors nécessaire de changer l'algorithme séquentiel afin d'augmenter sa granularité et donc diminuer le surcoût de cohérence : c'est-à-dire augmenter le nombre d'opérations effectuées par accès au descripteur de travail. Puisque c'est l'opération **extract_seq** qui permet d'extraire la taille du bloc d'instructions qui sera traité séquentiellement, nous devons alors analyser le choix de cette taille pour masquer le surcoût de synchronisation par rapport au nombre d'opérations effectuées. Si le nombre d'opérations extraites par **extract_seq** est très grand, il y aura perte de parallélisme car la fraction extraite sera exécutée séquentiellement : aucun vol ne peut être effectué sur la fraction extraite donc cela diminue le degré de parallélisme. Si ce nombre est petit, les surcoûts de synchronisations peuvent masquer le gain de calcul. Donc il faut chercher une bonne méthode pour calculer ce nombre. Une façon de calculer ce nombre est de déterminer d'abord le

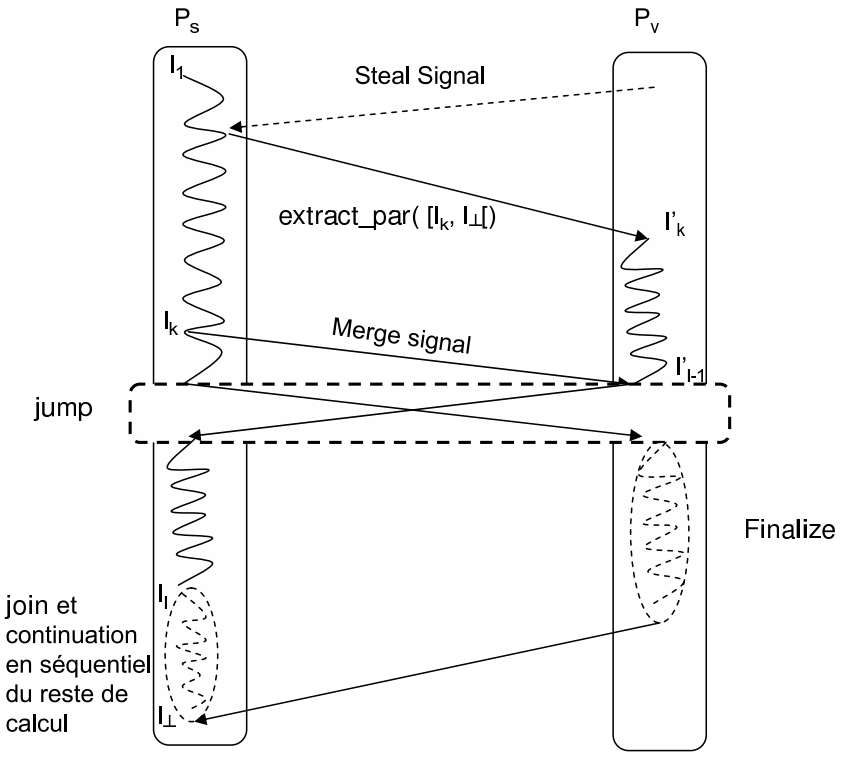

FIG. 6.4 – L'exécuteur P_s exécute l'algorithme séquentiel sur son descripteur de travail. L'exécuteur P_v appelle `extract_par` et travaille sur le descripteur de travail $[I'_k, I'_\perp[$

temps en nombre d'opérations sur un nombre non borné de processeurs de l'algorithme parallèle sur l'intervalle restant (la profondeur de l'algorithme). Nous rappelons que la profondeur de l'algorithme parallèle est la borne inférieure du temps d'exécution de l'algorithme parallèle sur n'importe quel nombre de processeurs. En choisissant ce nombre comme la taille du bloc d'instructions à extraire par l'opération **extract_seq**, ceci n'augmentera pas la profondeur de l'algorithme. Le théorème 6.4.1 suivant montre que ce choix peut permettre à l'algorithme adaptatif d'atteindre un travail séquentiel asymptotiquement optimal tout en n'augmentant pas la profondeur de l'algorithme sur un nombre infini de processeurs. Le théorème 6.4.1 donne une borne du nombre d'opération **extract_seq** effectué sur un seul exécuteur qui fait un travail séquentiel.

Théorème 6.4.1. *Soit W_{seq} le nombre d'opération séquentiel du travail restant à faire par un exécuteur. A chaque opération **extract_seq** on suppose que D instructions sont extraites où D est la profondeur de l'algorithme parallèle sur le travail restant. Si les deux hypothèses suivantes sont vérifiées :*

1. $D \geq \log W_{seq}$,
2. $\forall \epsilon > 0, \lim_{W_{seq} \to \infty} \frac{D}{W_{seq}^{\epsilon}} = 0$,

*alors pour tout $\delta > 0$, le nombre d'appels à **extract_seq** est asymptotiquement borné par $(1 + \delta)\frac{W_{seq}}{\log_2 W_{seq}}$ quand W_{seq} tend vers l'infini.*

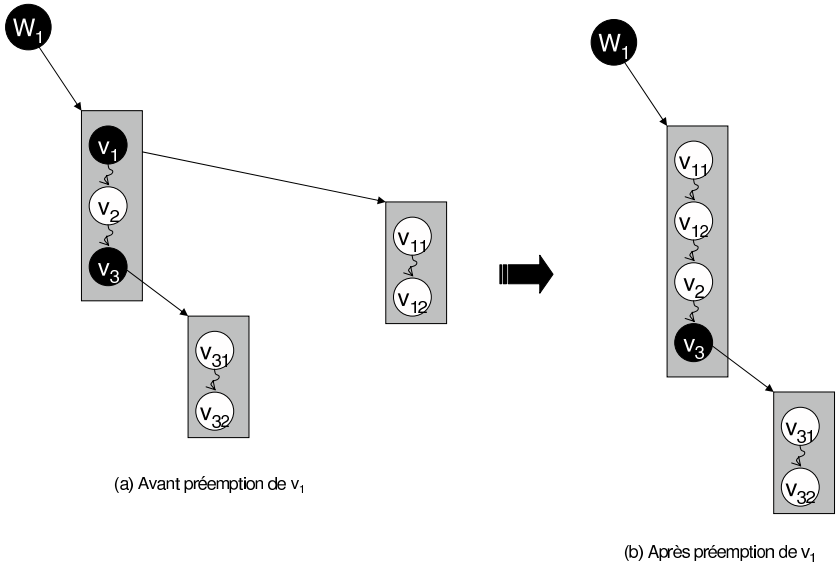

FIG. 6.5 – La liste des Descripteurs de travail de P_s avant et après l'opération de préemption. w_1 est le descripteur de travail de P_s ; P_s préempte l'exécuteur P_v exécutant v_1 ; après la préemption la liste des descripteurs de P_v est enchaînée à la tête de la liste des descripteurs de P_s.

Démonstration. D'après l' hypothèse 2, on a $\forall \delta, \exists w_0$, tel que $\forall W_{seq} \geq w_0$, on a $D < \frac{\delta}{2} W_{seq}^{\frac{\delta}{2+\delta}}$ (1). En multipliant (1) par $W_{seq}^{\frac{2}{2+\delta}}$, on obtient $DW_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2} W_{seq}^{\frac{\delta}{2+\delta}} \frac{2}{2+\delta}$, ce qui donne $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2} \frac{W_{seq}}{D}$ (2). Pour le calcul du nombre total d'appels à **extract_seq**, nous allons diviser la preuve en deux parties : dans la première partie on considère le nombre d' **extract_seq** quand le travail restant est suffisamment gros et dans la deuxième partie on considère le nombre d' **extract_seq** quand il reste peu de travail à effectuer :

1. Pour un travail w restant suffisamment gros c'est à dire tel que $w > W_{seq}^{\frac{2}{2+\delta}}$ opérations : il y a au moins D opérations extraites à chaque appel à **extract_seq** , d'après l'hypothèse 1, $D \geq \log\left(W_{seq}^{\frac{2}{2+\delta}}\right) = \frac{2}{2+\delta} \log W_{seq}$(3). En prenant l'inverse de (3) on aura $\frac{1}{D} \leq \frac{2+\delta}{2} \frac{1}{\log W_{seq}}$(4), puis en multipliant (4) par W_{seq}, on obtient $\frac{W_{seq}}{D} \leq (1 + \frac{\delta}{2}) \frac{W_{seq}}{\log W_{seq}}$. Donc, il y a au plus $\frac{W_{seq}}{D} \leq (1 + \frac{\delta}{2}) \frac{W_{seq}}{\log W_{seq}}$ appels à **extract_seq**.

2. Pour peu de travail restant (w_0) à faire (grain fin), il y a au plus $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2} \frac{W_{seq}}{D}$ opérations qui seront exécutées ($\forall W_{seq} \geq w_0$), et donc au plus $W_{seq}^{\frac{2}{2+\delta}}$ appels à **extract_seq** (si chaque appel retourne seulement une opérations). Puisque $\forall W_{seq} \geq w_0$, on a $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2} \frac{W_{seq}}{D}$ (d'après (2)), et d'après l'hypothèse 1, $D \geq \log W_{seq}$, d'où $W_{seq}^{\frac{2}{2+\delta}} < \frac{\delta}{2} \frac{W_{seq}}{\log W_{seq}}$. Donc, le nombre d'appels à **extract_seq** dans cette phase est moins que $\frac{\delta}{2} \frac{W_{seq}}{\log W_{seq}}$.
En additionnant ces deux bornes, on obtient que le nombre total d'appels à **extract_seq** est au plus $\left(1 + \frac{\delta}{2} + \frac{\delta}{2}\right) \frac{W_{seq}}{\log W_{seq}}$.

❑

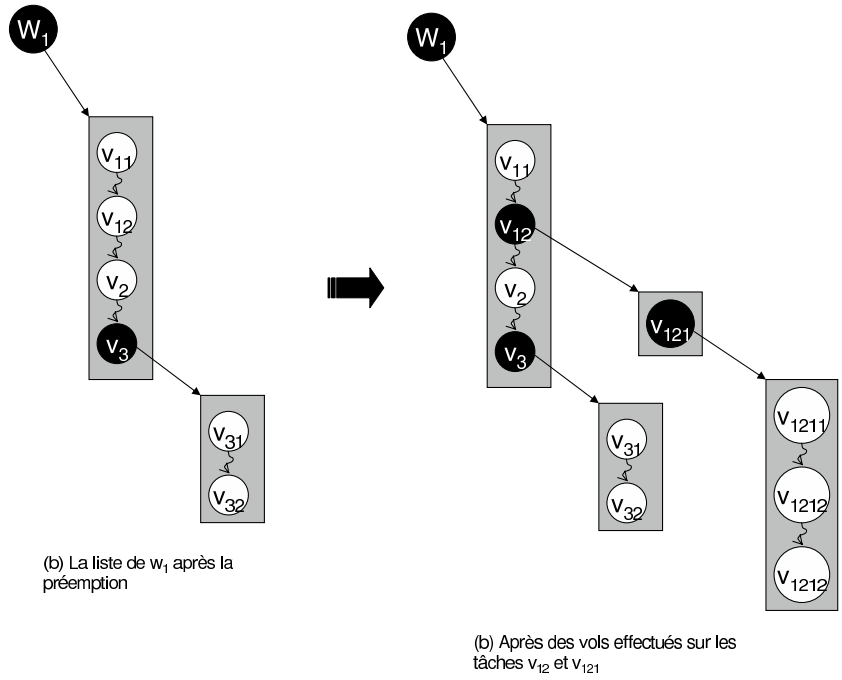

FIG. 6.6 – Comportement dynamique de la liste

D'où, le surcoût engendré par l'opération **extract_seq** dans la boucle **nano-loop** est asymptotiquement borné par $\frac{W_{seq}}{\log W_{seq}}$. Cette borne est similaire à celle de la partition récursive par vol de travail quand la récursivité est faite en divisant par deux le travail à chaque étape jusqu'à un seuil $\Omega(D)$, mais avec le prix d'augmenter la profondeur par un facteur $\rho > 1$. La première hypothèse semble raisonnable, puisque $\log W_{seq}$ est la borne inférieure de la profondeur potentielle si W_{seq} est optimal. La seconde hypothèse signifie que le travail global est beaucoup plus grand que le travail sur le chemin critique, qui est réaliste pour beaucoup d'algorithmes.

Amortissement du surcoût arithmétique par marcro-loop :

Pour certains algorithmes, W_{seq} n'est pas connu d'avance (par exemmple l'algorithme `find_if` de la STL) ou encore le nombre d'opérations en parallèle peut être largement supérieur à W_{seq} (par exemple le calcul des préfixes). Pour tenir compte de ceci, un troisième niveau de contrôle appelé **macro-loop** [25] est ajouté. La macro-loop est constituée d'un ensemble d'étapes où dans chaque étape un flot d'instructions de taille inconnue à priori sera exécuté en parallèle. Soit $s_1, s_2, \ldots s_i$ la taille du flot d'instructions à l'étape i. Pour le choix de la taille du bloc, nous utilisons la technique présentée dans [7]. Dans [7] différents choix de taille ont été étudiés, en particulier pour $s = \rho^{\frac{n}{\log n}}$ avec $1 < \rho < 2$. L'étape i de la macro-loop de taille s_i ne commence qu'après terminaison de l'étape $i-1$. Pour obtenir $\sum_{i=1,m} s_i \simeq W_{seq}$, dans la suite nous posons $s_i = \frac{\sum_{j=1,i-1} s_j}{\log \sum_{j=1,i-1} s_j}$ (avec s_1 initialement égale à une valeur constante). La macro-loop préserve asymptotiquement le travail W_{seq} tout en augmentant au plus la profondeur potentielle par un facteur au plus $\log W_{seq}$. L'algorithme final du schéma adaptatif s'obtient en ajoutant la macro-loop au deux autres boucles (micro-loop et nano-loop). Nous analysons dans le théorème

suivant le temps d'exécution de l'algorithme adaptatif exécutant un algorithme quelconque.

Théorème 6.4.2. *Si l'exécution parallèle d'un algorithme sur un nombre infini de processeurs fait $(1+\sigma)W_{seq}$ opérations avec $\sigma \geq 0$, alors avec une grande probabilité, le temps d'exécution de cet algorithme sur p processeurs avec l'algorithme [macro-micro-nano] est :*

$$T_p = \frac{\sigma+1}{\sigma+p}W_{seq} + \mathcal{O}\left(D\log^2 W_{seq}\right), \textit{ quand } W_{seq} \to \infty.$$

Démonstration. En utilisant le schéma algorithmique sans la macro-loop, d'après le théorème 1, le nombre d'instructions exécutées séquentiellement augmente seulement par la nano-loop de $\mathcal{O}\left(\frac{\log W_{seq}}{W_{seq}}\right)$, soit un nombre total d'opérations égal à $W_{seq} + \mathcal{O}\left(\frac{\log W_{seq}}{W_{seq}}\right)$. En utilisant la macro-loop, ce nombre total est augmenté de $\mathcal{O}\left(\log W_{seq}\right)$ car le nombre d'étapes dans la macro-loop est au plus $\mathcal{O}\left(\log W_{seq}\right)$. Donc, le nombre total d'opérations exécutées par un seul processeur est au plus égal à $W_{seq}\left(1 + \mathcal{O}\left(\frac{\log W_{seq}}{W_{seq}}\right) + \mathcal{O}\left(\frac{1}{\log W_{seq}}\right)\right) = W_{seq}$ asymptotiquement. Si seulement un processeur P_s exécute l'algorithme, il n'est jamais préempté et exécutera simplement l'algorithme séquentiel : il fait donc W_{seq} opérations asymptotiquement. Si p processeurs ($p \geq 2$) exécutent l'algorithme, au plus $p-1$ font des opérations **extract_par**, faisant au plus $W_v = (1+\sigma)(W_{seq} - W_s)$ opérations (car ils exécutent un algorithme parallèle à grain fin qui fait plus d'opérations) et P_s fait W_s opérations. Le nombre d'**extract_par** (nombre de vols) dans une étape (il y a au plus $\mathcal{O}\left(\log W_{seq}\right)$ étapes) de la macro-loop est de $\mathcal{O}\left(\log W_{seq}\right)$ (profondeur obtenue par découpe récursive) avec une grande probabilité, donc le nombre total d'**extract_par** dans la macro-loop est $\mathcal{O}\left(\log^2 W_{seq}\right)$ avec une grande probabilité ; alors le nombre total de préemption par la victime est de $\mathcal{O}\left(D\log^2 W_{seq}\right)$. Puisque le processus P_s n'est jamais inactif, sauf durant la préemption des voleurs ou éventuellement durant la dernière étape de la macro-loop de taille au plus $\frac{W_{seq}}{\log W_{seq}}$: $T_p = W_s + \mathcal{O}\left(D\log^2 W_{seq}\right)$ (1). De plus, dû à l'ordonnancement par vol de travail, avec une grande probabilité : $T_p = \frac{W_v}{p-1} + \mathcal{O}\left(D\log^2 W_{seq}\right)$ (2). En multipliant l'équation (1) par $\sigma+1$ et en l'addition à l'équation (2) multipliée par $p-1$, on obtient $T_p = \frac{\sigma+1}{\sigma+p}W_{seq} + \mathcal{O}\left(D\log^2 W_{seq}\right)$. ❑

La section suivante détaille l'implémentation du schéma en C++ et Kaapi.

6.5 Interface C++ sur Kaapi-Framework

Pour développer des algorithmes parallèles à grain adaptatatif, nous avons développé un Framework basé sur le schéma générique présenté précédemment. Ce Framework a été développé en C++ et utilise Athapascan/Kaapi pour exploiter l'ordonnancement par vol de travail fourni par celui-ci. Dans l'implémentation de ce FrameWork, nous proposons trois classes génériques qui peuvent être étendues pour développer des algorithmes parallèles à grain adaptatif. Ces trois classes sont : la classe **WorkAdapt**, la classe **JumpWork**, et la classe **FinalizeWork**. Les spécifications de ces classes sont données dans les sections suivantes.

La classe **WorkAdapt** permet la construction d'un descripteur de travail adaptatif. Elle fournit 11 fonctions qui sont décrites dans le tableau 6.1. Un utilisateur souhaitant développer un algorithme parallèle doit instancier (spécialiser) les fonctions dont il aura éventuellement besoin. Ces fonctions seront implémentées sans aucun verrou de synchronisation car ils sont gérés par le moteur exécutif du FrameWork. La classe **WorkAdapt** utilise la classe **JumpWork**. L'utilisateur doit étendre la classe **JumpWork** pour définir sa propre classe qui permet de faire le saut et la classe **FinalizeWork** pour définir sa propre classe de finalisation. Par exemple si nous considérons qu'un WorkAdapt est décrit par un intervalle $w = [f, l[$ où f est un indice de début et l un indice de fin, et si on suppose que w a été préempté à l'indice k, alors on peut définir l'intervalle $[k, l[$ comme un **JumpWork**. Toutes ces fonctions ne seront pas complétées par l'utilisateur, il les remplira selon ses besoins.

Fonction	Spécification
bool extract_seq()	permet d'extraire une partie du travail localement, retourne faux s'il n'y a plus de travail à extraire.
bool extract_par()	permet de voler une partie du travail sur le travail restant à faire, retourne faux s'il n'y a plus de travail à extraire.
void local_run()	permet d'exécuter localement l'algorithme séquentiel optimal.
void join(const WorkAdapt* stolenwork)	Permet de faire la fusion du résultat du voleur avec le résultat de la victime.
void jump(JumpWork* &)	Permet d'effectuer le saut sur le travail qui a été fait par le voleur.
void after_jump(const JumpWork*)	Permet d'affecter le travail restant à faire par le voleur qui a été préempté à la victime qui sera son nouveau travail.
bool extract_nextmacro_work()	Permet d'extraire la taille de la macro-loop
bool get_finalize_work(FinalizeWork*& fw)	Permet de récupérer le travail à finaliser par le voleur qui a été préempté.
void get_main_result()	Permet de récupérer le résultat local de la victime.
void get_thieft_result()	Permet de récupérer le résultat local du voleur.
bool must_jump()	Permet d'autoriser tous les exécuteurs à faire un saut sur leurs voleurs respectifs.

TAB. 6.1 – Interface fournie par la classe WorkAdapt

Fonction	Spécification
bool extract_seq()	permet d'extraire une partie du travail localement, retourne faux s'il n'y a plus de travail à extraire.
bool extract_par()	permet de voler une partie du travail sur le travail restant à faire, retounre faux s'il n'y a plus de travail à extraire.
void local_run()	permet d'exécuter localement l'algorithme séquentiel optimal.

TAB. 6.2 – Interface fournie par la classe FinalizeWork

6.5.1 Premier exemple d'utilisation : transform

Nous allons illustrer l'utilisation de la classe WorkAdapt sur trois exemples : l'algorithme **transform** de la librairie standard C++ (STL), l'algorithme du produit itéré et le calcul de fibonacci. Nous commençons par l'exemple de **transform**. L'algorithme **transform** sur une séquence consiste à appliquer une même fonction sur chaque élément de la séquence. Dans la librairie standard C++ (STL) une séquence peut être définie par des itérateurs de début et de

fin. Un itérateur est un objet en C++ permettant de parcourir les éléments d'une séquence ou d'un ensemble d'objets de même type. En C++ l'opérateur ++ permet sur un itérateur de passer à l'élément suivant et l'opérateur * retourne l'élément référencé par l'itérateur. L'algorithme séquentiel **transform** est décrit dans l'algorithme 6.5.1.

Algorithme 6.5.4

```
1   template<class InputIterator, class OutputIterator, class function>
2   void transform (InputIterator first, InputIterator last, OutputIterator out, function fct) {
3       while(first !=last) *out++ = fct(*first++) ;
4   }
```

Algorithme 6.5.1: Algorithme séquentiel de **transfom**

Nous allons programmer de manière adaptative cet algorithme en utilisant l'interface adaptative que nous avons développée. Nous supposons que la séquence utilise des itérateurs à accès direct. D'une manière classique cet algorithme est facilement parallélisable sur p processeurs, car la séquence peut être divisée en p parties et chaque processeur exécute indépendamment sa partie sans avoir besoin de communiquer ses résultats avec d'autres processeurs. A partir de cette analyse, nous pouvons utiliser seulement trois fonctions de l'interface proposée pour rendre cet algorithme adaptatif qui sont : **extract_seq**, **extract_par** et **local_run**. Pour le calcul de la taille du bloc à extraire par la fonction **extract_seq**, nous devons calculer la profondeur de cet algorithme sur une infinité de processeurs d'après les analyses théoriques de la section 6.4. Nous représentons par n le nombre d'éléments de la séquence c'est à la différence entre l'itérateur de début et de fin. La séquence peut être divisée de manière récursive par découpe en moitié jusqu'à un seuil égal à 1, et à partir de cette taille chaque processeur applique la fonction sur un élément. La taille initiale de la séquence étant de n donc la hauteur de l'arbre représentant les découpes est $\log n$ ce qui représente aussi la profondeur du calcul. Dans la pratique une constante α est fixée devant la profondeur $\log n$ pour amortir les surcoûts de synchronisation. Le seuil d'arrêt de la fonction **extract_par** peut être fixé à 1 ou 2 (grain fin) qui augmente le degré de parallélisme, ce choix n'augmentera pas le surcoût de parallélisme car le seuil $\alpha \log n$ de la fonction **extract_seq** permet d'amortir ce surcoût. Dans cet exemple, notre descripteur initial de travail sera représenté par les deux itérateurs de début et de fin. Nous donnons ci-dessous les implémentations des trois fonctions de l'interface adaptative.

Nous allons appeler par **MyTransfomWork** la classe implémentant ces fonctions. Dans la classe, les attributs _first et _last représentent les itérateurs de début et de fin du travail à réaliser. Les attributs _beg, _end représentent les positions courantes de début et fin du travail en cours de traitement. Les attributs _beg_local et _end_local permettent de déterminer l'intervalle du travail à réaliser localement par la fonction **local_run**. L'attribut _alpha représente la constante α, et l'attribut _pargrain représente le grain d'extraction du parallélisme.

```
template<class InputIterator, class OutputIterator, class function>
 class MyTransfomWork : public WorkAdapt {
     InputIterator _fisrt, _last ;
```

```
    OutputIterator _out;
    function _fct;
    int _beg, _end;
    int _beg_local, _end_local;
    int _alpha;
    int _pargrain;


    public :
    MyTransfomWork(InputIterator first, InputIterator last, OutputIterator out,
        int beg, function fct, int alpha=1, int pargrain=2) : WorkAdapt(), _first(first),
        _last(last), _out(out), _beg(beg), _fct(fct), _alpha(alpha), _pargrain(pargrain)
    { _end = _last-_first; }

    bool extract_seq() {
        if(_beg < _end) {
        _beg_local = _beg; _beg += log2(_end-_beg);
            if(_beg > _end) _beg = _end;
            _end_local = _beg;
        return true;
        } else {
    return false;
    }

    bool extract_par(WorkAdapt*& stolenwork) {
        if((_end-_beg) > _pargain) {
            int mid = _end - (_end-_beg)/2;
            int thieft_end = _end;
            _end = mid;
            stolenwork = new TransformWork<InputIterator, OutputIterator, function>(_first,
                    _first+thieft_end, _out, mid, _alpha, _pargrain);
            return true;
        }
        else {
        return false;
        }
    }

    bool local_run() {
        std : :transform(_first+_beg_local, _first+_end_local, _out+_beg_local, _fct);
    }

}
```

Après avoir spécialisé ces fonctions, nous lançons le moteur adaptatif sur notre classe contenant les fonctions remplies (algorithme 6.5.2 ligne 5). Le lancement du moteur adaptatif se fait

à l'aide de la fonction **adapt : :run** qui permet de faire tourner le schéma adaptatif présenté dans la section 6.3.

Algorithme 6.5.5

```
1  template<class InputIterator, class OutputIterator ,class fct>
2  void my_transform_adapt(InputIterator first, InputIterator last, OutputIterator out) {
3      MyTransformWork<InputIterator, OutputIterator, fct>* my_work ;
4      my_work = new MyTransformWork<InputIterator, OutputIterator, fct>(first, last, out, 0) ;
5      adapt : :run(my_work) ;
6  }
```

Algorithme 6.5.2: Lancement d'un algorithme de transform adaptatif

6.5.2 Deuxième exemple d'utilisation : produit itéré

Le deuxième exemple que nous allons illustrer est l'exemple du produit itéré. Le calcul du produit itéré est le suivant :

- **Entrée :** un tableau de valeurs contenant n éléments d'un ensemble muni d'une loi interne associative $\star$.
- **Sortie :** res : le résultat du produit itéré tel que : $res = \star_{i=0}^{n-1} f(i)$ où f est une fonction qui retourne les éléments du tableau.

L'algorithme séquentiel du produit itéré est trivial et est :
for (int $i = 1$, res $= f(0)$; $i < n$; $i++$) res $\star = f(i)$;

La seule différence de cet algorithme avec l'algorithme de **transform** est qu'il y'a fusion des résultats calculés par les processeurs dans le premier. Donc pour paralléliser de manière adaptative nous utiliserons d'autres fonctions en plus des trois autres utilisées dans l'algorithme adaptatif **transform** qui sont **join**, **get_result**, **jump** et **after_jump**. Comme dans l'algorithme **transfom** la profondeur de l'algorithme du produit itéré est aussi de $\log n$. Nous donnons ci-dessous seulement le code des fonctions **join**, **get_result**, **local_run()**, **jump** et **after_jump** car les implémentations pour les deux autres sont similaires à celles de **transform**.

Nous appelons notre classe **MyProdIterWork**. L'attribut _local_res contient le résultat localement calculé et les autres attributs sont les mêmes que ceux de **transform**. Ci-dessous l'implémentation de cette classe :

```
template<class InputIterator, class BinOp>
class MyProdIterWork : public WorkAdapt {

    . . .
    . . .
    std : :iterator_traits<InputIterator> : :value_type _local_res
```

```
    bool local_run() {
        InputIterator tmp = proditer_seq(_first+_beg_local, _first+_end_local, _binop);
        _local_res = BinOp()(_local_res, tmp);
    }

    void joint(const WorkAdapt* jw) {
        ProdIterWork<InputIterator, BinOp>* src;
        src = new dynamic_cast<const ProdIterWork<InputIterator, BinOp>* >(jw);
        InputIterator thieft_res = src->get_result();
        _local_res = BinOp()(_local_res, thieft_res);
    }

    void get_result() {
        return _local_res;
    }

    void jump(NextJumpWork* & j) {
        int end = _end;
        _end = _beg; // arret prementuré du calcul;
        j = new MyNextJumpWork(_beg, end);
    }

    void after_jump(NextJumpWork* j) {
        MyNextJumpWork* src = new dynamic_cast<const MyNextJumpWork* >(j);
        _beg = j->get_beg();
        _end = j->get_end();
    }

}
```

L'instanciation pour construire l'algorithme est similaire à transform (algorithme 6.5.2).

6.5.3 troisième exemple d'utilisation : calcul de fibonacci

Le calcul du $n^{ème}$ entier F_n de la suite de Fibonacci (voir définition chapitre 2 dans la section 2.6.3) est un cas différent des deux autres exemples. En fait l'entrée de l'algorithme de Fibonacci est un entier (n) tandis que pour les deux exemples l'entrée est un tableau d'éléments. Pour adapter cet algorithme, nous allons d'abord construire un algorithme séquentiel itératif en utilisant une file à double tête (*deque*). L'algorithme séquentiel est le suivant :

```
if(n <= 2) res = 1;
 else {
    deque.push(n − 1); //on empile n − 1 dans la file;
```

```
        deque.push(n − 2) ; //on empile n − 2 dans la file ;
        while ( !deque.empty()) {
            int k = deque.front() ; //on dépile k dans la file ;
            if(k <= 2) res+ = 1 ;
            else {
                deque.push(k − 1) ; //on empile k − 1 dans la file ;
                deque.push(k − 2) ; //on empile k − 2 dans la file ;
            }
        }
    }
```

où res est le résultat calculé du $n^{ème}$ entier F_n.

Dans l'algorithme adaptatif l'opération **extract_seq** consiste à extraire un élément en tête de la file, et une taille égale à $\alpha * (nom\_elts\_restants\_dans\_la\_file)$. L'opération **local_run** consiste à appliquer l'algorithme séquentiel décrit précédemment en décrémentant à chaque fois la taille extraite par l'opération **extract_seq**, son exécution s'arrête dès que cette taille dévient égale à zéro ou que la pile devient vide. L'opération **extract_par** consiste à extraire un élément en bas de la file.

6.6 Conclusion

Dans ce chapitre, nous avons spécifié un schéma générique qui permet de développer des programmes parallèles adaptatifs ayant des garanties d'efficacité sur une grande classe d'architectures parallèles, en particulier dans le cas pratique usuel où les processeurs sont de vitesse variable par rapport à l'application, du fait de leur utilisation par d'autres processeurs (système ou autres applications). Ce schéma est basé sur un couplage, récursif et dynamique entre plusieurs algorithmes résolvant un même problème. Nous avons utilisé une liste distribuée de tâches volées qui permet de minimiser le surcoût de création de tâches et de limiter le surcoût lié à la récursivité. Cette liste remplace la pile à double tête (*deque*) utilisé dans l'ordonnancement par vol classique, dans [90] nous avons appelé cet ordonnancement, l'ordonnancement par vol sans pile (*deque-free*). Pour garantir des performances optimales, nous avons utilisé trois niveaux de boucle : la boucle **nano-loop** qui permet de minimiser le surcoût de synchronisation ; la boucle **micro-loop** qui permet de faire le couplage entre un algorithme séquentiel qui minimise le nombre d'opérations et un algorithme parallèle qui diminue la profondeur de l'algorithme parallèle ; la boucle **macro-loop** qui permet de borner le surcoût arithmétique et de garantir une bonne performance pour les algorithmes avec terminaison anticipée. Nous avons donné une analyse théorique montrant les garanties de performance du schéma générique. Pour pouvoir utiliser le schéma expérimentalement, nous avons développé une interface générique dont le moteur exécutif est basé sur Kaapi/Athapascan [39] qui permet de faire le vol de travail. Un programmeur désirant développer des algorithmes adaptatifs n'a qu'à remplir certaines fonctions de l'interface. Dans le chapitre qui suit, nous allons utiliser ce schéma générique pour adapter plusieurs algorithmes de la librairie standard C++ [65, 87].

Chapitre 7

Application du schéma générique à la librairie standard STL

Dans ce chapitre, nous allons présenter une application de notre schéma adaptatif aux algorithmes de la librairie standard C++. Nous commencerons d'abord par présenter dans la section 7.1 un aperçu de la librairie standard C++ [65, 87]. Dans la section 7.2, nous donnons un état de l'art des bibliothèques parallèles de la STL (*Standard Template Library*) existantes. Dans la section 7.3, nous présentons une classification des algorithmes de la STL. Dans la section 7.4, nous détaillerons les algorithmes que nous avons implémentés de manière adaptative. Enfin, nous terminerons par des expérimentations que nous avons menées en comparant nos algorithmes adaptatifs à des algorithmes de la librairie standard C++ implémentés dans d'autres bibliothèque parallèles.

7.1 Un aperçu de la librarie standard C++

La librairie générique standard C++ fournit un ensemble de classes appelées **conteneurs**, permettant de représenter les structures de données ou collections de données tels que les vecteurs, les listes, les piles, les ensembles ou les tableaux associatifs. Elle fournit aussi des **algorithmes** qui permettent de manipuler le contenu de ces collections de données et des **itérateurs** qui permettent de les parcourir et d'accéder à leurs données. Tous les algorithmes fournis sont génériques c'est à dire ils peuvent fonctionner avec tous les types de données (des types prédéfinis par le langage ou des types définis par l'utilisateur).

Les **patrons** de classes (en anglais *Templates*) offrent la possibilité d'écrire du code générique en C++. Tous les algorithmes de la librairie standard C++ sont constitués de patrons de classes. L'implémentation de la STL est essentiellement basée sur des **patrons**.

Les trois composants de la STL **conteneurs**, **itérateurs**, **algorithmes** sont étroitement liés et, la plupart du temps, ils interviennent simultanément dans un programme utilisant des conte-

neurs.

Les conteneurs permettent de contenir d'autres objets ou éléments. Les patrons de conteneurs sont paramétrés par le type de leurs éléments. Par exemple, on pourra construire une liste d'entiers, un vecteur de flottants ou une liste de points par les déclarations suivantes :
list<int> l_entier / liste vide d'éléments de type int */*
vector<double> v_flot(10) / vecteur de taille 10 d'éléments de type double */*
list<point> l_point / liste vide d'éléments de type point */*

Le tableau 7.1 présente les conteneurs de la STL et leur description.

Conteneur	Description
vector	Un tableau dynamique offre des opérations efficaces sur la fin de sa séquence d'éléments, l'accès à un élément du tableau est direct.
deque	Une file à double-accès. Elle est optimisée de telle sorte que les opérations s'appliquant à ses deux extrémités soient efficaces.
list	Une liste chaînée. Elle est optimisée pour l'insertion et la suppression d'éléments, par contre l'accès à un élement est lent.
map	Une séquence de paires (clé, valeur) offrant une recherche rapide au moyen de la clé. Chaque clé est associée à une seule valeur.
multimap	C'est un map dans lequel la duplication des clés est autorisée.
set	Peut être considéré comme un map dans lequel seules les clés sont définies (pas de valeurs).
multiset	C'est un set dans lequel la duplication des clés est autorisée.

TAB. 7.1 – Les conteneurs de la STL

Un itérateur est un objet défini généralement par la classe conteneur concernée qui généralise la notion de pointeur en C. A un instant donné, un itérateur possède une valeur qui désigne un élément donné d'un conteneur ; on dira souvent qu'un itérateur pointe sur un élément d'un conteneur. Un itérateur peut pointer sur l'élément suivant du même conteneur en lui appliquant l'operateur++ (incrémentation). Deux itérateurs sur un même conteneur peuvent être comparés par égalité ou inégalité. L'élément pointé par un itérateur peut être obtenu en lui appliquant l'operateur* (déréférenciation). Tous les conteneurs fournissent un type itérateur associé portant le nom *iterator*.

Tous les conteneurs fournissent des valeurs particulières de type *iterator*, sous forme de fonctions membres (**begin()** et **end()**) qui permettent leurs parcours.

- **begin()** retourne un itérateur sur le premier élément du conteneur.
- **end()** retourne un itérateur qui représente la fin du conteneur. Elle consiste à pointer, non pas sur le dernier élément d'un conteneur, mais juste après.

Les deux itérateurs retournés par ces fonctions forment un intervalle semi-ouvert du type $[first, last[$ où $first$ est l'itérateur retourné par **begin()** et $last$ par **end()**. Plus généralement, on peut définir ce qu'on nomme un intervalle d'itérateurs en précisant les bornes sous forme de deux valeurs itérateur. On dit également que les éléments désignés par cet intervalle forment

une séquence. Cette notion d'intervalles d'itérateur est très utilisée par les algorithmes et par certaines fonctions membres.

Toutes les classes conteneurs pour lesquelles *iterator* est au moins bidirectionnel (on peut donc lui appliquer ++ et --) disposent d'un second itérateur noté *reverse_iterator*. Il permet d'explorer le conteneur suivant l'ordre inverse. Il existe également des valeurs particulières de type *reverse_iterator* fournies par les fonctions membres *rbegin()* et *rend()* ; on peut dire que *rbegin()* pointe sur le dernier élément du conteneur, tandis que *rend()* pointe juste avant le premier.

Les différents types d'itérateurs que l'on nomme généralement catégories d'itérateurs peuvent être classés de façon hiérarchique : le tableau 7.2 montre le diagramme d'héritage de classe de ces différentes catégories d'itérateurs.

Catégorie d'itérateur	Description
Input iterator	Permet d'effectuer la lecture et le déplacement avant.
Output iterator	Permet d'effectuer l'écriture et le déplacement avant.
Forward iterator	Permet d'effectuer la lecture, l'écriture et le déplacement avant.
Bidirectional iterator	Permet d'effectuer la lecture, l'écriture et le déplacement avant et arrière.
Random access iterator	Permet d'effectuer la lecture, l'écriture et accès aléatoire (direct).

TAB. 7.2 – Les catégories d'itérateurs

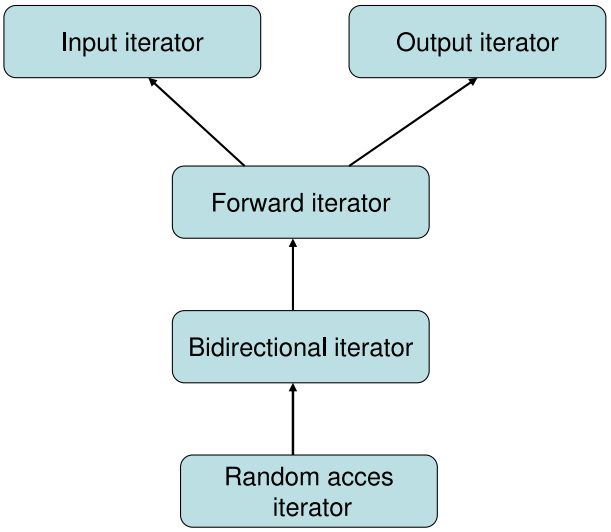

FIG. 7.1 – Le diagramme d'héritage de classe des catégories d'itérateurs

Pour qu'on puisse appliquer des opérations de base à un conteneur tels que la copie, le tri, ou trouver le premier élément de son contenu ayant une valeur donnée, la bibliothèque standard a fourni des algorithmes permettant de répondre à ces besoins. Ces algorithmes sont fournis sous forme de patrons de fonctions, paramétrés par le type des itérateurs qui leurs sont fournis

en argument. Un même algorithme peut être appliqué à des conteneurs différents. La section suivante donne un état de l'art sur la parallélisation des algorithmes de la librairie standard.

7.2 Parallélisation des algorithmes de la STL : état de l'art

La parallélisation des algorithmes de la librairie STL n'est pas une nouvelle idée, elle a donné lieu à plusieurs travaux. HPC++ Parallel Standard Template Library (PSTL) [52, 51] est l'une des premières bibliothèques parallèles de la librairie STL : elle fournit des conteneurs de classes distribués et des iterateurs parallèles permettant la programmation parallèle de quelques algorithmes de la STL sur des architectures à mémoire partagée et distribuée.

La librairie STAPL (*Standard Template Adaptive Parallel Library*) fournit aussi des conteneurs de classes distribués qui permettent l'écriture des programmes sur des machines à mémoire partagée et distribuée. La différence entre STAPL et HPC++ PSTL est que STAPL est basé sur une approche adaptative. L'approche adaptative dans STAPL [98, 88, 2] est basée sur une stratégie de combinaison et de sélection d'algorithmes (voir chapitre 3). C'est à dire, lors de l'installation de la bibliothèque, les paramètres de configuration de la machine cible sont collectés, et les différents algorithmes candidats à la résolution du problème sont testés avec ces paramètres afin de sélectionner le meilleur algorithme pour la résolution du problème.

Récemment, des bibliothèques parallèles de la librairie STL pour les architectures multi-cœurs sont apparues. MPTL (*Multi-Processing Template Library*) [6] est une bibliothèque utilisant Pthread pour la parallélisation et implémente plusieurs algorithmes de la librairie STL. Cependant MPTL n'implémente pas les algorithmes parallèles complexes comme `partial_sum` , `unique_copy` , `remove_copy_if` , `partition` , `merge`. L'algorithme `find` implémenté dans MPTL est une implémentation naïve qui ne garantit pas de bonne performance si la position du premier élément à chercher se trouve au début du tableau où l'élément doit être cherché. L'équilibrage de charge dans MPTL est basé sur la méthode Maître-Esclave et est à grain fixé.

La stratégie de parallélisation appliquée dans la bibliothèque RPA (*Range Partition Adaptor*) [4] est basée sur une technique qui permet de convertir une structure à une dimension (un bloc, en anglais *range*) en une structure à deux dimensions (une collection de sous blocs, en anglais *subranges*) où chaque sous bloc est exécuté séquentiellement par un processus. Des fusions éventuelles des résultats des ces sous blocs peuvent être effectuées possiblement en parallèle. Cette bibliothèque n'implémente que quelques algorithmes simples de la librairie STL comme `for_each` , `copy` , `count`.

MCSTL (*Multi-Core Standard Template Library*) [84] très récemment intégré dans la librairie *gnu libstdc++* sous le nom de *Gnu libstdc++ parallel mode* [83] est une bibliothèque basée sur OpenMP implémentant plusieurs algorithmes parallèles de la librairie STL pour les architectures multi-cœurs. Pour plusieurs algorithmes, MCSTL propose une parallélisation statique et dynamique. La parallélisation dynamique dans MCSTL est utilisée pour adapter l'algorithme à la charge de l'environnement d'exécution, elle est basée sur l'ordonnancement par vol de travail qui place les tâches à grain fixé sur des processeurs inactifs. Cependant quelques algorithmes implémentés ne bénéficient pas encore de cette parallélisation dynamique comme : `partial_sum` , `unique_copy` , `remove_copy_if` , `merge`.

La bibliothèque TBB [70, 95] (voir section 2.6) offre des conteneurs parallèles (concurrent_queue) et des algorithmes parallèles pour **parallel_for**, **parallel_while**, **parallel_reduce**, et **parallel_scan**, qui peuvent être utilisés pour programmer plusieurs algorithmes en parallèle de la librairie STL. Par exemple l'algorithme de **parallel_scan** peut être utilisé pour programmer l'algorithme `partial_sum`, il est basé sur une approche récursive similaire à l'algorithme de Ladner-Fischer (voir section 3.3.4) qui fait $2W_{seq}$ opérations.

7.3 Classification des algorithmes de la STL

Par paralléliser les algorithmes de la STL, nous pouvons les classer en 5 catégories de classes. Le tableau 7.3 présente les cinq catégories de classe.

Classe	Algorithmes implémentés
Algorithmes sans fusion de résultats	`copy`, `copy_bacward`, `fill`, `fill_n`, `for_each`, `generate`, `generate_n`, `replace`, `replace_if`, `replace_copy`, `replace_copy_if`, `swap_ranges`, `transform`
Algorithmes avec fusion de résultats	`count`, `count_if`, `accumulate`, `inner_product`, `partial_difference`
Algorithmes avec terminaison anticipée	`find`, `find_if`, `find_end`, `find_first_of`, `adjacent_find`, `search`, `search_n`
Algorithmes avec surcoûts de parallélisme	`partial_sum`, `remove_copy_if`, `unique_copy`
Algorithme de partition et de tri	`merge`, `stable_sort`, `partition`, `sort`

TAB. 7.3 – Classification des algorithmes de la STL en fonction de leur parallélisation.

Les algorithmes sans fusion de résultats comprennent tous les algorithmes de la STL où chaque processeur peut exécuter indépendamment l'intervalle de données qui lui a été affecté. Après l'exécution, le processeur n'a pas besoin de faire d'autres traitements (communications ou échanges d'éléments avec un autre processeur par exemple). Donc les surcoûts de communications ou échanges de données sont nuls. Une parallélisation classique de cette classe d'algorithmes sur p processeurs consiste à diviser la séquence initiale de taille n en p intervalles de taille $\frac{n}{p}$, et chaque processeur exécute localement dans l'intervalle $[i * \frac{n}{p}, (i+1) * \frac{n}{p}[$ avec $0 \leq i \leq p$ où i est l'identification du processeur. L'inconvénient de cette parallélisation est que le temps d'exécution final de l'algorithme est le temps d'exécution du processeur le plus lent.

Les algorithmes avec fusion de résultats comprennent tous les algorithmes de la STL où chaque processeur exécute l'intervalle qui lui a été affecté et sauvegarde son résultat calculé. Après l'exécution, le processeur envoie le résultat qu'il a calculé à un autre processeur ou reçoit le résultat calculé par un autre processeur. Une parallélisation classique de cette classe d'algorithmes consiste en deux phases : la première phase consiste à exécuter par chaque processeur l'algorithme séquentiel de la partie de la séquence qui lui a été allouée ; la seconde phase consiste à fusionner séquentiellement les résultats obtenus de tous les processeurs par un seul processeur. Les résultats de tous les processeurs peuvent être fusionnés en parallèle par

les p processeurs, mais les surcoûts de création de tâches ou copies de données peuvent être pénalisants par rapport à la parallélisation classique.

Les algorithmes avec terminaison anticipée terminent dès qu'un élément ou un ensemble d'éléments de la séquence en entrée vérifie un prédicat donné même si la fin de la séquence n'est pas atteinte. La parallélisation de cette classe d'algorithmes est difficile, puisque son temps d'exécution est non prédictible. Une parallélisation classique de cette classe d'algorithmes consiste en deux phases : la première phase consiste à exécuter par chaque processeur l'algorithme séquentiel de la partie de la séquence qui lui a été allouée ; la seconde phase consiste à fusionner séquentiellement les résultats obtenus de tous les processeurs par un seul processeur. L'inconvénient de cette parallélisation est que si le prédicat donné est vérifié au début de la séquence alors l'algorithme sur un seul processeur sera beaucoup plus efficace que l'algorithme sur p processeurs. Par exemple, supposons que la séquence en entrée est désignée par $[0, n[$, et que le prédicat est vérifié à la première position. Supposons que le temps d'une opération dure une unité de temps, alors le temps de l'algorithme sur un seul processeur est $T_1 = 1$ et le temps sur p processeurs est $T_p = \frac{n}{p} > T_1$. Un autre inconvénient aussi est que si les processeurs sont de vitesses différentes, le temps d'exécution final sera le temps d'exécution du processeur le plus lent.

Toute parallélisation des algorithmes avec surcoûts de parallélisme sur un nombre de processeurs supérieur à un, introduit une augmentation du nombre d'opérations effectuées par rapport à l'algorithme séquentiel optimal. Une analyse théorique de l'algorithme de **partial_sum** (calcul des préfixes) a été donné dans le chapitre 3 et montre bien que le nombre d'opérations effectuées dépend du nombre de processeurs pour un algorithme optimal sur p processeurs. Les algorithmes parallèles **unique_copy** et **remove_copy** sont similaires à l'algorithme de calcul des préfixes et de la compression [11]. Toute parallélisation récursive de ces algorithmes effectue un nombre d'opérations 2 fois supérieur par rapport à l'algorithme séquentiel optimal.

La bibliothèque standard STL fournit deux algorithmes génériques différents de tri d'un tableau (ensemble disposant d'un itérateur avec accès aléatoire à un élément) : `sort` (tri introspectif basé sur une partition quicksort) et `stable_sort` (basé sur un tri par fusion), ce dernier garantissant l'ordre relatif de deux éléments de même valeur par rapport à la relation d'ordre choisie.

7.4 Applications

Dans cette section, nous détaillons l'implémentation des algorithmes de la librairie standard (STL) que nous avons parallélisés avec notre interface présentée au chapitre 6. Nous donnons pour chaque classe décrite dans le tableau 7.3 les fonctions de l'interface qui ont été utilisées et comment ces fonctions ont été implémentées. Comme ça été dit dans la section 7.1, une séquence de données peut être représentée en C++ par un intervalle semi-ouvert dont les bornes sont des itérateurs. Concernant les itérateurs utilisés, nous précisons que tous les algorithmes implémentés utilisent des itérateurs à accès aléatoires. Nous représentons le descripteur de tra-

vail à l'aide des itérateurs similaire à la représentation de la séquence de données. Avant toute exécution, le descripteur de travail initial représente la séquence globale en entrée. Pour simplifier la présentation, nous désignons ce descripteur de travail par $w = [f, l[$ où f représente l'itérateur qui pointe sur le premier élément de la séquence et l l'itérateur qui pointe sur le dernier élément de la séquence. Pour les analyses théoriques, nous désignons par n la taille de l'intervalle représentant la séquence qui n'est autre que la différence entre l'itérateur de début et de fin ($n = l - f$).

1) Algorithmes sans fusion : puisque dans cette catégorie de classe (cf tableau 7.3) chaque processeur exécute indépendamment son calcul, l'implémentation de trois fonctions de l'interface proposée dans le chapitre 6 suffira pour rendre adaptatifs les algorithmes de cette classe. Ces trois fonctions que nous avons implémentées sont : **extract_seq**, **extract_par** et **local_run**, leurs descriptions sont détaillées ci-dessous :

- **extract_seq** : tous les algorithmes de cette classe peuvent être exécutés sur un nombre de processeurs non borné en temps $\log n$. Donc chaque opération **extract_seq** extrait l'intervalle $[f', f' + \beta \log (l' - f')[$ sur le travail restant à faire séquentiellement représenté par $w' = [f', l'[$ où β est un paramètre multiplicatif qui permet de masquer significativement en pratique les surcoûts de synchronisation par rapport aux surcoûts arithmétiques. En pratique, nous avons observé que pour les calculs dont la durée d'une opération unitaire est supérieure à $0.1ms$, $\beta = 1$ convient et pour ceux dont la durée d'une opération unitaire est de l'ordre $1ns$, β entre 100 et 150 convient.
- **extract_par** : extrait la moitié du travail restant qui est à la charge du processeur actif (appelé victime). Pour cela elle modifie le travail restant du processeur victime et construit un nouveau travail à effectuer en utilisant les informations minimales sur le descripteur de travail de la victime (itérateur courant de début et de fin). Plus précisément si avant l'extraction le travail de la victime était représenté par $w = [f, l[$, alors après l'extraction le nouveau travail modifié de la victime sera représenté par $[f, mid[$ et le travail qui sera extrait est $[mid, l[$ où $mid = l - \frac{l-f}{2}$. Pour tous les algorithmes nous avons fixé le grain d'extraction du parallélisme à 2 (grain fin).
- **local_run** : exécute l'appel de l'algorithme séquentiel de la STL sur l'intervalle $[f', f' + \alpha \log (l' - f')[$ qui a été extrait lors de l'opération **extract_seq**.

Analyse théorique : nous allons calculer le temps d'exécution de ces algorithmes en utilisant le théorème 6.4.2 du chapitre 6 qui a été généralisé pour tous les algorithmes linéaires. On peut remarquer que pour tous les algorithmes ici, tous les processeurs font une seule passe sur les intervalles qui les ont été alloués pour effectuer leurs calculs, donc à partir de ce constat, la constante σ (proportion de surcoût arithmétique dû au parallélisme, voir théorème 6.4.2) est nulle. Étant donné que le troisième niveau de boucle (**macro-loop**) n'a pas été utilisé ici et en plus aussi les processeurs ne préemptent pas, le surcoût d'ordonnancement est $\mathcal{O}(\log n)$ (nombre de vols). D'où le temps d'exécution sur p processeurs est égal à :

$$T_p = \frac{T_{seq}}{p} + \mathcal{O}(\log n).$$

2) Algorithmes avec fusion de résultats : comme dans cette catégorie de classe les processeurs ont besoin de fusionner leurs résultats, nous utiliserons alors plus de fonctions de l'interface générique que pour les algorithmes sans fusion de résultats. Nous utiliserons en plus des trois fonctions **extract_seq**, **extract_par**, **local_run** les fonctions suivantes : **join**, **get_thieft_result**, **must_jump**, **jump** et **afterJump**. Nous autoriserons tous les processeurs victimes à préempter les voleurs afin de pouvoir faciliter les fusions des résultats partiels calculés par les voleurs. La préemption permet en effet d'éviter les surcoûts de création des tâches de fusion. A noter que, pendant toute préemption, un seul processeur voleur est préempté et non pas tous. Pour permettre cette autorisation de préemption par tous les processeurs victimes, nous faisons retourner par la fonction **must_jump** la valeur booléenne **true** qui par défaut retourne **false**. Nous faisons retourner par la fonction **get_thieft_result** le résultat calculé localement dans chaque descripteur de travail. La fonction **join** récupère le résultat calculé par le voleur à l'aide de la fonction **get_thieft_result** sur le descripteur de travail volé, puis applique la fonction permettant de faire la fusion du résultat de la victime avec celui du voleur récupéré. La fonction **jump** permet de construire un nouvel intervalle à partir du travail restant à faire par le voleur : par exemple si initialement l'intervalle de travail du voleur est de $[i, j[$ et que lors de la préemption le voleur a été arrêté à l'itérateur i' avec $i' > i$, alors le nouveau travail construit par la fonction **jump** sera de $[i', j[$. La fonction **afterJump** affecte le travail construit par la fonction **jump** comme nouveau travail à faire ; le processeur victime et ce dernier vont continuer séquentiellement sur ce travail. Les implémentations des fonctions **extract_seq**, **extract_par** et **local_run** sont similaires à celles des algorithmes sans fusion de résultats.

Analyse théorique : comme pour les algorithmes sans fusion de résultats nous allons calculer le temps d'exécution de ces algorithmes en utilisant le théorème 6.4.2 du chapitre 6. Aussi comme pour les algorithmes sans fusion de résultats on peut remarquer que tous les processeurs font une seule passe sur les intervalles qui leur ont été alloués pour effectuer leurs calculs. Donc à partir de ce constat, la constante σ est nulle. Comme la préemption a été autorisée ici, le surcoût de préemption s'ajoute au surcoût d'ordonnancement. La profondeur de chaque algorithme étant de $\log n$, il y aura donc au plus $p \log n$ vols sur p processeurs et lors de la préemption chaque processeur victime doit attendre au plus $\beta \log n$ (la taille extraite par l'opération **extract_seq**) unités de temps, donc le surcoût total d'ordonnancement et de préemption est de $\mathcal{O}\left(\log^2 n\right)$. D'où le temps d'exécution sur p processeurs est égal à :

$$T_p = \frac{T_{seq}}{p} + \mathcal{O}\left(\log^2 n\right).$$

3) Les algorithmes avec terminaison anticipée : la difficulté de ces types d'algorithmes est qu'on ne peut pas prédire leurs temps d'exécution. Un exemple typique est l'algorithme **find_if** qui prend en entrée une séquence de données (un intervalle $[f, l[$ où f et l sont des itérateurs) et retourne le premier itérateur N de la séquence vérifiant un prédicat **pred** (*N) donné : si **pred** (*N) est $vrai$ alors l'élément a été trouvé et $N \in [f, l[$; sinon si **pred** (*N) est faux alors l'élément n'a pas été trouvé et $N = l$. Alors le travail séquentiel W_{seq} de l'algorithme **find_if** est $W_{seq} = \sum_{k=1}^{N} \tau_{pred}(k)$ où $\tau_{pred}(k)$ est le temps unitaire de la fonction **pred** appliquée sur l'élément à la position k. Pour assurer que le travail W exécuté par l'algorithme parallèle soit proche du travail de l'algorithme séquentiel W_{seq}, nous avons utilisé la fonction **extract_nextmacro_work** de l'interface générique en plus des autres fonctions utilisées pour

l'adaptation des algorithmes avec fusion de résultats. Ainsi, pour la parallélisation adaptative de cette catégorie de classe, nous avons utilisé les trois niveaux de boucle du schéma adaptatif. A la $i^{ème}$ étape de la macro-loop, la fonction **extract_nextmacro_work** extrait $\frac{\sum_{j=1}^{i-1} s_j}{\log \sum_{j=1}^{i-1} s_j}$ éléments qui seront traités en parallèle par l'ensemble des processeurs actifs : on rappelle que $\sum_{j=1}^{i-1} s_j$ est la taille de la séquence parcourue et dans laquelle l'élément à chercher n'a pas été trouvé. Tous les processeurs cherchent ensemble dans l'étape i et, si l'élément n'est pas trouvé dans cette étape, ils avancent dans l'étape $i+1$; sinon ils s'arrêtent et le calcul est terminé. La technique de préemption accélère l'algorithme, puisque dès qu'un processeur trouve l'élément, il arrête tous les processeurs qui lui ont volé du travail (travaillant sur sa partie droite) ; puisqu'on cherche le premier élément et il a été trouvé à gauche, il est donc inutile de continuer à droite. Les implémentations des fonctions **extract_seq**, **extract_par**, **local_run**, **join**, **get_thieft_result**, **must_jump**, **jump** et **afterJump** sont similaires à celles des algorithmes avec fusion de résultats.

Analyse théorique : comme pour les algorithmes sans fusion de résultats nous allons calculer le temps d'exécution de ces algorithmes en utilisant le théorème 6.4.2 du chapitre 6. Aussi comme pour les algorithmes sans fusion de résultats on peut remarquer que tous les processeurs font une seule passe sur les intervalles qui les ont été alloués pour effectuer leurs calculs. Donc à partir de ce constat, la constante σ est nulle. La profondeur de chaque algorithme étant de $\mathcal{O}(\log n)$, il y aura donc au plus $\mathcal{O}(p \log n)$ vols sur p processeurs et lors de la préemption chaque processeur victime doit attendre au plus $\beta \log n$ **(**la taille extraite par l'opération **extract_seq)** unités de temps. Donc, le surcoût total d'ordonnancement et de préemption est de $\mathcal{O}\left(\log^2 n\right)$ dans une seule étape de la macro-loop. Enfin, il y a au plus $\mathcal{O}(\log n)$ étapes dans la macro-loop, donc le surcoût total d'ordonnancement et préemption est $\mathcal{O}\left(\log^3 n\right)$ dans l'ensemble de l'exécution. D'où le temps d'exécution sur p processeurs :

$$T_p = \frac{T_{seq}}{p} + \mathcal{O}\left(\log^3 n\right).$$

4) Les algorithmes avec surcoûts de parallélisme : ces types d'algorithmes nécessitant deux passes pour leurs traitements en parallèle, nous avons implémenté alors la classe **FinalizeWork** de l'interface générique. Dans la STL, il y a trois algorithmes qui appartiennent à cette catégorie : `partial_sum`, `unique_copy` et `remove_copy_if`. L'algorithme de `partial_sum` de la STL est équivalent à l'algorithme du calcul des préfixes donc son implémentation adaptative est la même que celle de l'algorithme adaptatif du calcul parallèle des préfixes présentée dans le chapitre 4. Les implémentations de `unique_copy` et `remove_copy_if` sont similaires à l'implémentation du calcul des préfixes : la seule différence dans ces deux implémentations est que la taille finale du tableau en sortie n'est connue qu'après la fin du calcul. Pour adapter ces deux algorithmes, nous avons fait une implémentation similaire à celle de l'algorithme **processeur-indépendants** du traitement du traitement de flux proposé dans [11].

D'abord avant la présentation de l'algorithme qui traite ces cas, nous rappelons que l'algorithme `remove_copy_if` permet de supprimer dans une séquence les éléments correspondant à un prédicat, et l'algorithme `unique_copy` permet de supprimer dans une séquence des éléments contigus égaux. Comme dans le calcul des préfixes, il y un seul processus P_s qui exécute toujours l'algorithme séquentiel et les autres processus P_v exécutent l'algorithme parallèle. Initialement, le processus P_s démarre le calcul dans l'intervalle initial $[1, n[$. Mais l'intervalle

$[1, n[$ peut être volé et découpé récursivement par les processus P_v devenus inactifs. Un processus actif est toujours en train de traiter un intervalle $[a, b[$ et place le résultat directement soit dans le tableau de sortie final pour P_s, soit dans un tableau intermédiaire pour P_v.

Lorsqu'un processus P_v devient inactif, il choisit au hasard un processus jusqu'à trouver un processus actif (victime), vole la deuxième moitié $[\frac{a+b}{2}, b[$ de l'intervalle restant sur la victime et démarre le calcul sur ce nouveau intervalle dans un tableau intermédiaire.

Le processus P_s exécute toujours l'algorithme séquentiel, jusqu'à atteindre un intervalle $[a, b[$ qui a été volé et traité par un processus P_v dans un tableau intermédiaire $[u, v[$ de taille L, P_s préempte alors P_v et effectue un saut (**jump**) de la manière suivante. Soit i la position courante à laquelle pointe P_s dans le tableau de sortie, le tableau intermédiaire $[u, v[$ peut être finalisé de manière asynchrone et copié dans le tableau de sortie à la position i. Pendant ce temps, le processus P_s saute directement pour traiter l'intervalle commençant à la position $b + 1$, en écrivant le résultat dans le tableau de sortie final à la position $i + L$.

Notons que le processus P_s n'est jamais en attente, sauf pour les préemptions (effectuées en temps constant). Quand P_s atteint la fin du tableau à traiter, c'est à dire l'indice n, tous les traitements séquentiels sont nécessairement terminés et il reste éventuellement des finalisations et copies des tableaux intermédiaires dans le tableau de sortie final. P_s participe aux copies (finalisations) en devenant voleur comme les autres processus.

Les implémentations des fonctions **extract_seq**, **extract_par** et **extract_nextmacro_work** sont exactement similaires à celles du calcul des préfixes.

Analyse théorique : Le temps d'exécution T_p sur p processeur de l'algorithme présenté a été donné dans [11] et vérifie :

$$T_p \leq \frac{T_{seq}}{\Pi_{ave}} \frac{\omega_c + \omega_o}{p + \frac{\omega_o}{\omega_c}} + O\left(\frac{n}{\epsilon(n)}\right)$$

où ω_c est le temps unitaire du traitement d'un élément (temps de la fonction prédicat par exemple) et ω_o et le temps de copie d'une donnée (lors de la phase de finalisation).

La preuve de ce temps d'exécution est similaire à celle du calcul des préfixes. On obtient le temps d'exécution du calcul des préfixes pour $\omega_c = \omega_o = 1$. On peut aussi observer que lorsque $\omega_c \gg \omega_o$, l'accélération obtenue par l'algorithme est linéaire en p.

Enfin, on peut remarquer que pour le théorème général (théorème 6.4.2) la proportion de surcoût arithmétique dû au parallélisme (σ) est égale à 1 pour `partial_sum` et à $\frac{w_o}{w_c}$ pour `unique_copy` et `remove_copy_if`.

5) Les algorithmes de partition et merge : l'implémentation de la `partition` parallèle est la même que celle présentée dans le chapitre 5, et l'implémentation de `merge` est la même que celle de la fusion parallèle adaptative présentée dans le chapitre 5.

6) Les algorithmes de tris : la bibliothèque standard STL fournit deux algorithmes génériques différents de tri d'un tableau (ensemble disposant d'un itérateur avec accès aléatoire à un élément) : `sort` (tri introspectif basé sur une partition quicksort) et `stable_sort` (basé sur un tri par fusion), ce dernier garantissant l'ordre relatif de deux éléments de même valeur par rapport à la relation d'ordre choisie. L'implémentation parallèle du tri stable est la même

que celle du tri par fusion adaptatif présentée dans le chapitre 5 et, l'implémentation parallèle du tri non stable est la même que celle du tri parallèle introspectif présentée dans le chapitre 5.

7.5 Expérimentation

Nos expérimentations ont été faites sur une machine à mémoire partagée NUMA AMD Opteron composée de 8 noeuds bi-coeurs. Les algorithmes ont été implantés avec l'interface adaptative générique qui utilise l'ordonnancement par vol de travail fourni par Kaapi/Athapascan. Toutes les exécutions ont été réalisées dix fois et pour chaque test nous avons pris la moyenne. Nous avons utilisé la version 0.8.0-beta de la bibliothèque MCSTL et la version stable 20_014 de la bibliothèque TBB. Tous les programmes (MCSTL, TBB, Adaptative) ont été compilés avec le même compilateur gcc 4.2.3 et la même option de compilation -O2. Les données en entrée sont tirées aléatoirement dans un tableau de taille n. Les données sont de type double. La constante α a été fixée à $\alpha = 100$ pour les opérations de durée unitaire petite et à $\alpha = 1$ sur les deux machines après calibrage expérimental.

partial_sum. La figure 7.2 compare, pour $n = 10^8$, notre `partial_sum` adaptatif au `partial_sum` de MCSTL qui est basé sur une découpe en $p+1$ parties (même algorithme que celui présenté dans le chapitre 3). Le temps moyen séquentiel du `partial_sum` de la librairie standard C++ (STL) est de $1,24s$. On observe que le `partial_sum` adaptatif se comporte mieux que le `partial_sum` de MCSTL. Cette différence s'explique par le fait qu'il y a des petites charges dans le système, ce qui perturbe un peu l'algorithme parallèle statique, et notre algorithme s'adapte en fonction de ces charges. Nous pouvons remarquer que la meilleure accélération obtenue par ces expérimentations ne dépasse pas 4. Ceci est dû à la contention d'accès mémoire sur ces machines NUMA où chaque bi-processeurs partage la même mémoire. Mais si nous augmentons le temps de l'opération $\star$, l'accélération augmente, ce qui été montré dans les expérimentations précédentes.

Les figures 7.3 et 7.4 comparent notre `partial_sum` adaptatif avec celui implémenté dans la bibliothèque TBB qui est basé sur une découpe récursive dépendant de la taille du grain choisi. La figure 7.3 montre que notre algorithme adaptatif est meilleur que celui de TBB avec **auto_partioner** (grain adaptatif de TBB) sur un tableau contenant $n = 10^8$ doubles. La figure 7.4 compare les deux algorithmes avec un temps par opération $\star$ assez élevé de l'ordre de $1,5ms$. On peut observer que notre algorithme atteint la borne théorique inférieure $\frac{2W_{seq}}{p+1}$; tandis que le temps d'exécution T_p de l'algorithme de TBB est de $T_p = \frac{2W_{seq}}{p}$. Cette différence s'explique par le fait que le nombre d'opérations effectuées par l'algorithme de TBB est de l'ordre de $2n$ (algorithme récursif avec une double passe). Sur la figure 7.4, la comparaison a été faite avec les trois meilleures tailles de grain pour TBB après calibrage expérimental.

unique_copy. La figure 7.5 compare les temps d'exécution de notre algorithme adaptatif `unique_copy` à l'algorithme statique implémenté dans MCSTL sur un tableau contenant $n = 10^8$ doubles. L'algorithme de MCSTL est basé sur une découpe en $p + 1$ parties. Le temps séquentiel moyen de l'algorithme séquentiel de la STL est $0.61s$. On observe que l'algorithme

adaptatif `unique_copy` est meilleur que celui de MCSTL. Ceci s'explique par le fait que dans une découpe statique, chaque partie peut éventuellement avoir des tailles différentes, car dans l'algorithme `unique_copy` on ne connaît la taille du calcul final qu'après la fin du calcul : notre algorithme s'adapte dynamiquement dans ce contexte. Ceci est garanti par le troisième niveau de boucle et les techniques de préemption du schéma adaptatif qui permettent de minimiser le surcoût arithmétique.

remove_copy_if. La figure 7.6 montre les temps d'exécution obtenus par notre algorithme adaptatif `remove_copy_if` avec un temps de la fonction de test (prédicat) égal à $16\mu s$ et $n = 10^6$. Avec ce temps assez élevé, les performances restent scalables jusqu'à 16 processeurs et, l'algorithme atteint la borne inférieure [11]. Ceci est garanti par le troisième niveau de boucle les techniques de préemption du schéma adaptatif qui permet de minimiser le surcoût arithmétiques.

find_if. La figure 7.7 montre les performances obtenues par notre algorithme adaptatif en fonction de la position où se trouve le premier élément vérifiant le prédicat donné. La taille du tableau en entrée est de $n = 10^6$; la position k où doit se trouver cet élément est $10^2, 10^3, 10^4, 10^5, 5.10^5, 10^6$. Le temps d'exécution de la fonction testant le prédicat est de l'ordre de $\tau_{pred} = 35\mu s$. On obtient une accélération égale à 12 sur 16 processeurs pour $k = 10^6$. On observe que l'algorithme garantit une accélération supérieure ou égale à celle de l'algorithme séquentiel de la STL quelque soit la position où se trouve le premier élément à chercher. Ceci est garanti par le troisième niveau de boucle du schéma adaptatif qui permettent de détecter rapidement la terminaison anticipée.

merge. La figure 7.8 compare notre algorithme adaptatif de fusion de deux listes triées avec l'algorithme de fusion de listes triées de la bibliothèque MCSTL. Elle montre l'accélération obtenue par notre algorithme sur 8 processeurs en faisant varier la taille (n) de la séquence de données à trier. Dans les deux cas, l'accélération augmente avec n et est limitée lorsque n est petit. Cependant, pour $n > 10000$, on obtient toujours pour l'algorithme adaptatif une accélération supérieure à 1 par rapport au temps séquentiel, tandis que pour la fusion parallèle de MCSTL, l'accélération dévient supérieure à 1 lorsque $n > 316228$. On observe sur cette figure que notre algorithme est au moins 25% supérieur pour n assez grand à la fusion parallèle statique. Ceci s'explique par le mécanisme d'adaptation qui garantit qu'un processeur suit l'exécution séquentielle et s'adapte automatiquement à la charge en fonction de la disponibilité des processeurs.

stable_sort. La figure 7.9 montre la performance sur un tableau de taille $n = 10^8$ de notre algorithme de tri stable par rapport à l'algorithme de tri stable de la MCSTL. On observe que notre algorithme est au moins 25% plus rapide pour n assez grand. Ceci s'explique aussi de la même manière que pour la figure 7.8 (merge).

sort. La figure 7.10 montre la performance sur un tableau de taille $n = 10^8$ de notre algorithme de tri rapide (non stable) par rapport à l'algorithme de tri rapide de TBB. On observe que notre algorithme se comporte mieux que celui de TBB. Ceci s'explique par le fait que notre algorithme utilise une partition adaptative parallèle tandis que celui de TBB utilise une partition

séquentielle.

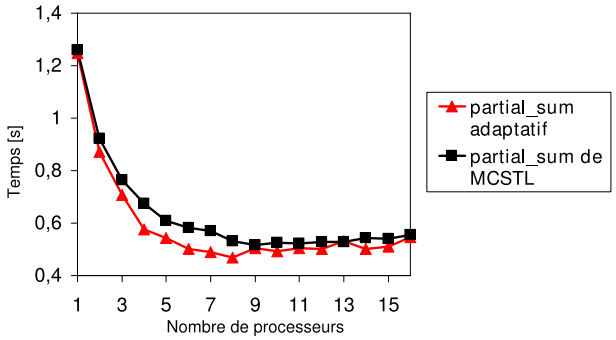

FIG. 7.2 – partial_sum : comparaison de l'adaptatif avec MCSTL

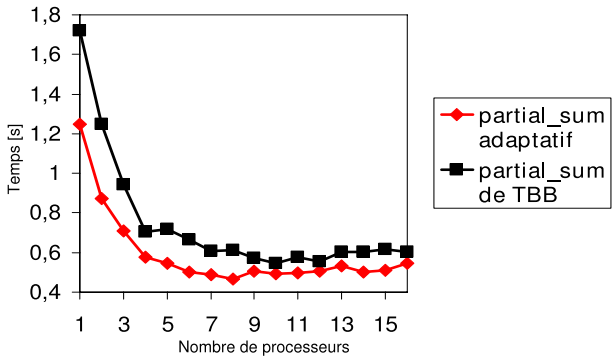

FIG. 7.3 – partial_sum : comparaison de adaptatif avec TBB pour $n = 10^8$ sur des doubles

7.6 Conclusion

Nous avons présenté dans ce chapitre une parallélisation adaptative de plusieurs algorithmes de la librairie standard C++ (*Standard Template Library*) avec des conteneurs à accès aléatoires. Nous avons donné des analyses théoriques prouvant que les algorithmes parallélisés atteignent des performances asymptotiquement optimales avec une grande probabilité. Chaque algorithme effectue sur p processeurs un nombre optimal d'opérations qui est à un facteur de 1 de la borne inférieure sur p processeurs (*e.g.* le nombre optimal d'opérations de l'algorithme `partial_sum` de la STL est de $\frac{2p}{p+1}W_{seq}$). Nous avons vérifié aussi expérimentalement que les algorithmes implémentés sont performants et stables par rapport à des algorithmes implémentés dans les bibliothèques récentes (TBB[70] et MCSTL[84]) dédiés à la programmation parallèle sur des architectures muli-cœurs. A notre connaissance, nous sommes les premiers à fournir des algorithmes optimaux pour : `partial_sum` , `unique_copy` , et `remove_copy_if`.

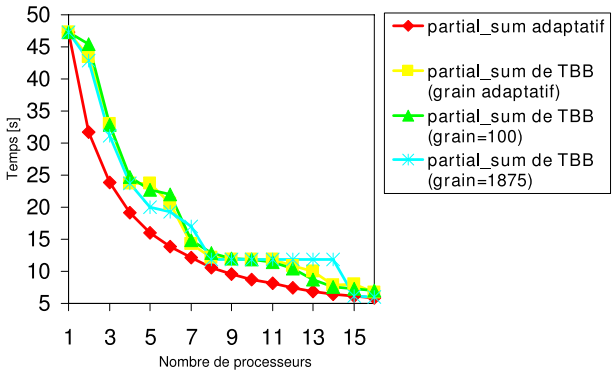

FIG. 7.4 – partial_sum : comparaison de adaptatif avec TBB

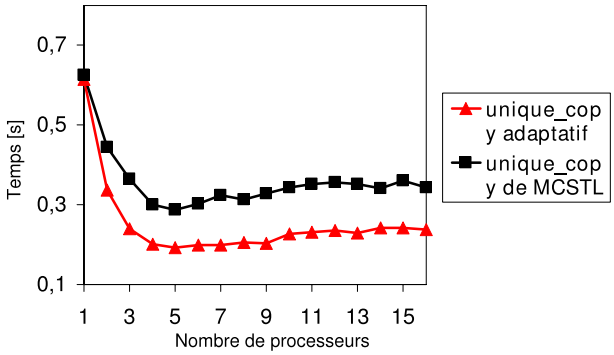

FIG. 7.5 – unique_copy : comparaison de adaptatif avec MCSTL

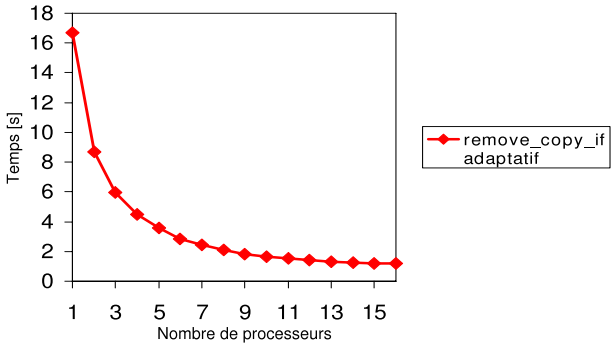

FIG. 7.6 – remove_copy : comparaison de adaptatif avec TBB

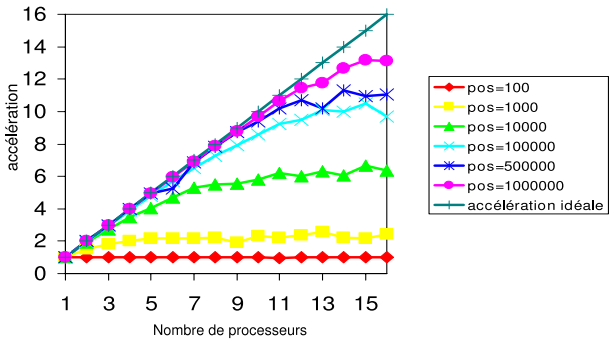

FIG. 7.7 – Accélération obtenue par l'algorithme adaptatif de find_if pour $n = 10^8$ en fonction des position de l'élément à chercher

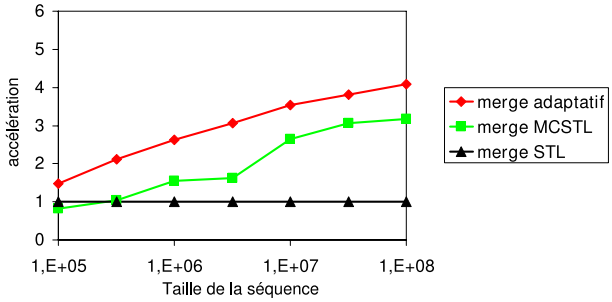

FIG. 7.8 – merge : comparaison de adaptatif avec MCSTL

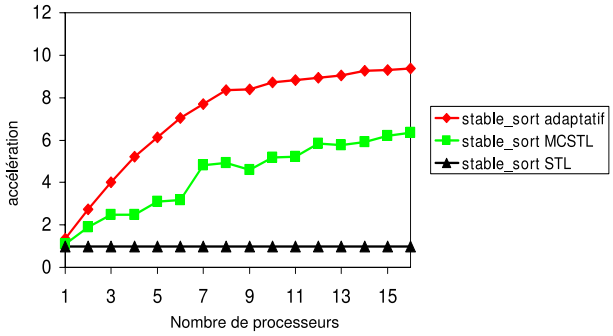

FIG. 7.9 – stable_sort : comparaison de adaptatif avec MCSTL pour $n = 10^8$

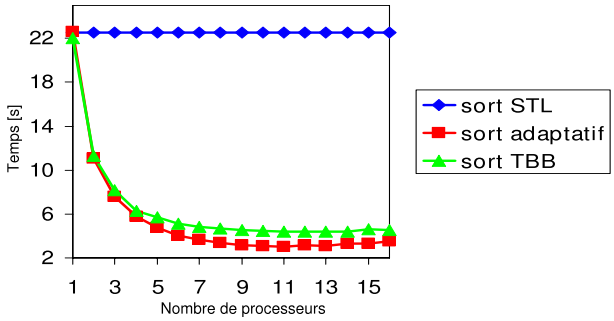

FIG. 7.10 – sort : comparaison de adaptatif avec TBB pour $n = 10^8$

Chapitre 8

Conclusion et perspectives

Dans cette thèse, nous avons spécifié et validé un schéma générique original qui permet de développer des programmes parallèles ayant des garanties d'efficacité. Il exploite un ordonnancement dynamique de type vol de travail, mais contrairement à l'ordonnancement par vol de travail classique qui utilise une liste distribuée à double tête (*deque*) de tâches prêtes à être volées, notre ordonnancement dynamique utilise une liste distribuée de tâches volées (*deque-free*) qui permet de minimiser le surcoût de création de tâches et de limiter le surcoût lié à la récursivité. Ainsi, la génération du parallélisme n'est effectuée qu'en cas d'inactivité d'un processeur. Lors de l'exécution sur un nombre restreint ou variable de ressources, ce schéma permet de limiter le surcoût lié à la génération de parallélisme, sans limiter le degré de parallélisme potentiel. Il est basé sur le couplage de deux algorithmes, l'un séquentiel, et l'autre parallèle à grain fin. Il est adapté aux problèmes pour lesquels la parallélisation entraîne, malgré un gain de temps, une pénalité en nombre d'opérations ou en performances.

Dans le chapitre 3, nous avons présenté les besoins d'utiliser des algorithmes adaptatifs. Pour illustrer ces besoins, nous avons utilisé le calcul parallèle des préfixes comme un cas d'étude. Nous avons d'abord donné une borne inférieure du temps d'exécution de ce calcul sur des processeurs à vitesses variables, puis nous avons présenté trois algorithmes parallèles de ce calcul qui ont permis de mettre en évidence les besoins d'algorithmes parallèles adaptatifs. Dans ce chapitre, nous avons proposé un nouvel algorithme optimal du calcul des préfixes sur des processeurs identiques, et nous avons montré les avantages et inconvénients de cet algorithme.

Dans le chapitre 4, nous avons proposé un nouvel algorithme parallèle pour le calcul des préfixes qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles. Nous avons montré que son temps d'exécution était asymptotiquement optimal. Il est équivalent à celui de l'algorithme séquentiel lorsqu'un seul processeur est disponible et à celui d'un algorithme parallèle optimal lorsque p processeurs identiques sont disponibles. Dans le cas de p processeurs de vitesses variables, son temps est équivalent à celui d'un algorithme optimal sur p processeurs identiques de vitesse égale à la moyenne des vitesses. Ces résultats théoriques sont validés par les expérimentations menées sur des machines SMP à 16 et 8 processeurs dans les quatre cas suivants : processeurs dédiés, processeurs perturbés par des processus addition-

nels, processeurs hétérogènes et processeurs distribués ; conformément à l'analyse théorique, l'algorithme adaptatif est le plus rapide.

Dans le chapitre 5, nous avons proposé une parallélisation adaptative de la fusion de deux listes triées et de l'algorithme de partition. Nous avons utilisé la fusion parallèle adaptative pour paralléliser d'une manière adaptative l'algorithme de tri par fusion, et nous avons utilisé la partition parallèle adaptative pour paralléliser d'une manière adaptative l'algorithme de tri rapide. Grâce au couplage d'un algorithme séquentiel avec un algorithme parallèle à grain fin ordonnancé par vol de travail, des garanties théoriques de performances sont obtenues par rapport au temps séquentiel sur des machines à mémoire partagée même lorsqu'elles sont utilisées en concurrence par d'autres applications. Les expérimentations menées montrent le bon comportement des algorithmes même sur des machines dont la charge des processeurs est perturbée ce qui est particulièrement intéressant en contexte multi-utilisateur.

Dans le chapitre 6, nous avons présenté l'algorithme décrivant le schéma générique adaptatif, et nous l'avons analysé théoriquement. Nous avons montré (théorème 6.4.1) que le surcoût de parallélisme de ce schéma sur un processeur est asymptotiquement négligeable devant le nombre d'opérations du meilleur algorithme séquentiel. Dans le théorème 6.4.2 nous avons montré que pour tout algorithme parallèle de surcoût linéaire utilisant ce schéma algorithmique générique, le temps parallèle d'exécution est asymptotiquement optimal. Pour pouvoir valider ce schéma expérimentalement, nous avons développé une interface générique en C++ basée sur un moteur exécutif implémentant le vol de travail. Cette interface générique a été implantée sur le noyau exécutif Kaapi et son interface applicative Athapascan. Cette interface permet de développer des programmes parallèles adaptatifs.

Dans le chapitre 7, nous avons utilisé l'interface générique basée sur le schéma spécifié pour paralléliser d'une manière adaptative plusieurs algorithme de la librairie standard C++ (*Standard Template Library*). Nous avons vérifié aussi expérimentalement que les algorithmes implémentés sont performants et stables par rapport à des algorithmes implémentés dans les bibliothèques récentes (TBB[70] et MCSTL[84]) dédiés à la programmation parallèle sur des architectures muli-cœurs.

Les perspectives que nous envisageons de poursuivre dans cette thèse sont les suivantes :

– Optimisation de l'interface développée : pour pouvoir exécuter efficacement en contexte distribué, l'interface actuelle devra être améliorée et optimisée. Une des améliorations pour l'exécution en contexte distribué consiste à minimiser le surcoût de communication, et cela peut être réalisé en gérant un seuil automatique de calcul qui permettra de masquer le surcoût de calcul par rapport au surcoût de communication. Puis après cette optimisation, le schéma devra être expérimenté en contexte distribué pour pouvoir observer son efficacité. Une autre direction à considérer dans cette partie consiste à supprimer les verrous, ce qui est important pour un travail de petite taille.
– Implémenter le schéma sur d'autres bibliothèques : pour valider d'avantage l'efficacité de ce schéma générique, il est important de le valider sur d'autres bibliothèques implantant le vol de travail comme intel TBB et CilK++, puis de comparer ces différentes implantations.

– Avoir des algorithmes caches et processeurs indépendants : Certains algorithmes déjà implémentés les sont déjà théoriquement comme le calcul des préfixes, partition, fusion. Mais pour mieux valider, nous devons analyser expérimentalement des défauts de cache en intégrant l'organisation de la hiérarchie mémoire (mémoire commune ou NUMA) et selon l'organisation on pourrait observer le comportement des algorithmes adaptatifs lorsque le nombre de cœurs devient en encore plus grand (32, 64, 128 cœurs par exemple). Une autre direction de cette partie est de construire des algorithmes de tris caches et processeurs indépendants, car les algorithmes implémentés pour le moment ne les sont pas.
– Utilisation du schéma pour des applications de simulation 3D : Une direction dans cette partie est d'utiliser des algorithmes déjà implémenté dans des situations réelles sur des vraies applications comme les applications de simulation 3D (ce travail peut être réalisé par exemple en utilisant SOFA [96]). Déjà des travaux ont été réalisées dans cette direction par Soares et al [86] sur des applications 3D à temps contraint (octree).

Bibliographie

[1] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *PPoPP '07 : Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 112–120, New York, NY, USA, 2007. ACM.

[2] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl : An adaptive, generic parallel c++ library. In *Languages and Compilers for Parallel Computing*, pages 195–210. Springer Berlin / Heidelberg, 2003.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2) :115–144, 2001.

[4] Matthew H. Austern, Ross A. Towle, and Alexander A. Stepanov. Range partition adaptors : a mechanism for parallelizing stl. *SIGAPP Appl. Comput. Rev.*, 4(1) :5–6, 1996.

[5] David A. Bader, Varun Kanade, and Kamesh Madduri. Swarm : A parallel programming framework for multicore processors. In *International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–8, Long Beach, California, USA, March 2007. IEEE.

[6] Didier Baertschiger. Multi-processing template library. Master's thesis, Université de Genève, Switzerland, 2006.

[7] O. Beaumont, E.M. Daoudi, N. Maillard, P. Manneback, and J.-L. Roch. Tradeoff to minimize extra-computations and stopping criterion tests for parallel iterative schemes. In *3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA04)*, CIRM, Marseille, France, October 2004.

[8] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory Comput. Syst.*, 35(3) :289–304, 2002.

[9] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5) :1260–1279, 2003.

[10] Julien Bernard, Jean-Louis Roch, Paoli Serge de and Miguel Santana. Adaptive encoding of multimedia streams on mpsoc. In LNCS 3994 Springer-Verlag, editor, *ICCS'06 International Conference on Computational Science (4), workshop Real-Time Systems and Adaptive Applications*, pages 999–1006, Reading, UK, May 2006.

[11] Julien Bernard, Jean-Louis Roch, and Daouda Traore. Processor-oblivious parallel stream computations. In *Proceedings of the 16th Euromicro Conference on Parallel, Distribu-*

ted and Network-Based Processing (PDP'08), pages 72–76, Toulouse, France, Frebruary 2008. IEEE Computer Society.

[12] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[13] Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann. Cost optimality and predictability of parallel programming with skeletons. *Parallel Processing Letters*, 13(4) :575–587, 2003.

[14] Holger Bischof, Sergei Gorlatch, and Roman Leshchinskiy. Generic parallel programming using c++ templates and skeletons. In Springer-Verlag LNCS 3016, editor, *Domain-Specific Program Generation*, pages 107–126, 2004.

[15] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11) :1526–1538, 1989.

[16] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2) :135–167, / 1998.

[17] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[18] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1) :202–229, 1998.

[19] C.A.R. Quicksort. *The Computer Journal*, 5(1) :10–16, 1962.

[20] Christophe Cerin, Jean-Christophe Dubacq, and Jean-Louis Roch. Methods for partitioning data and to improve parallel execution time for sorting on heterogeneous clusters. In LNCS 3947 Springer-Verlag, editor, *International conference on Grid and Pervasive Computing, IGC'2006*, pages 175–186, Tunghia, Taiwa, May 2006.

[21] Christophe Cérin, Hazem Fkaier, and Mohamed Jemni. Accessing hardware performance counters in order to measure the influence of cache on the performance of integer sorting. In *IPDPS '03 : Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 274.1, Washington, DC, USA, 2003. IEEE Computer Society.

[22] Cole. Parallel programming, list homomorphism and the maximum segment sum problem. In *PARCO : Proceedings of the International Conference on Parallel Computing*. North-Holland, 1993.

[23] Van-Dat Cung, Vincent Danjean, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive and hybrid algorithms : classification and illustration on triangular system solving. In JG Dumas, editor, *Transgressive Computing TC'2006*, pages 131–148, Granada, Spain, April 2006.

[24] Van-Dat Cung, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive algorithms : theory and application. In SIAM PP'06, editor, *SIAM Parallel Processing 2006, Mini-Symposium MS1 : Adaptive algorithms for scientific computing*, pages 49–50, San Francisco, USA, February 2006.

[25] Vincent Danjean, Roland Gillard, Serge Guelton, Jean-Louis Roch, and Thomas Roche. Adaptive loops with kaapi on multicore and grid : Applications in symmetric cryptography. In *ACM PASCO'07*, London, Canada, July 2007.

[26] El-Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques*, 24 :1—20, 2005.

[27] Giorgos Dimitrakopoulos. High-speed parallel-prefix vlsi ling adders. *IEEE Trans. Comput.*, 54(2) :225–231, 2005. Member-Dimitris Nikolos.

[28] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications, 2002.

[29] Mathias Doreille, François Galilée, and Jean-Louis Roch. Construction dynamique du graphe de flot de données en Athapascan. In *RenPar'9*, Lausanne, Suisse, May 1997.

[30] Jean-Guillaume Dumas, Clément Pernet, and Jean-Louis Roch. Adaptive triangular system solving. In W. Decker, M. Dewar, E. Kaltofen, and S. Watt, editors, *Dagstuhl Seminar Proceedings – Challenges in Symbolic Computation Software*, Dagstuhl, Germany, July 2006.

[31] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. Fast parallel sorting under logP : Experience with the CM-5 :. *IEEE Transactions on Parallel and Distributed Systems*, 7(8) :791–805, 1996.

[32] Omer Egecioglu and Cetin Kaya Koc. Parallel prefix computation with few processors. *CANDM : An International Journal : Computers & Mathematics, with Applications*, 24(4) :77–84, 1992.

[33] Faith E. Fich. New bounds for parallel prefix circuits. In *STOC '83 : Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 100–109, New York, NY, USA, 1983. ACM.

[34] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9) :948–960, 1972.

[35] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conf. PLDI*, pages 212–223, 1998.

[36] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS 99)*, pages 285–297, New York, USA, October 1999.

[37] François Galilée, Jean-Louis Roch, Gerson Cavalheiro, and Matthias Doreille. Athapascan-1 : On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, October 1998.

[38] T. Gautier, J.-L. Roch, and F. Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *PAPP 2007 4th Int. Workshop on Practical Aspects of High-Level Parallel Programming*, China, May 2007.

[39] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *ACM PASCO*, pages 15–23, London, Canada, 2007.

[40] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of the European Conference on Parallel Processing, Euro-Par'96*, volume 1124, pages 401–408. Springer-Verlag, 1996.

[41] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[42] P. Heidelberger, A. Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Trans. Comput.*, 39(1) :133–138, 1990.

[43] David R. Helman and Joseph JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2) :265–278, 2001.

[44] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming*, pages 83–97, 2002.

[45] Bing-Chao Huang and Michael A. Langston. Practical in-place merging. *Commun. ACM*, 31(3) :348–352, 1988.

[46] Samir Jafar, Thierry Gautier, Axel W. Krings, and Jean-Louis Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In LNCS Springer-Verlag, editor, *EUROPAR'2005*, Lisboa, Portugal, August 2005.

[47] Samir Jafar, Axel W. Krings, Thierry Gautier, and Jean-Louis Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In IEEE, editor, *IEEE Electro/Information Technology Conference , (EIT 2005)*, Lincoln, Nebraska, May 2005. This paper received the EIT'05 Best Paper Award.

[48] Samir Jafar, Laurent Pigeon, Thierry Gautier, and Jean-Louis Roch. Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In IEEE, editor, *ICTTA'06 IEEE Conference on Information and Communication Technologies : from Theory to Applications*, pages 3347–3352, Damascus, Syria, April 2006.

[49] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[50] Minsoo Jeon and Dongseung Kim. Parallelizing merge sort onto distributed memory parallel computers. In *ISHPC '02 : Proceedings of the 4th International Symposium on High Performance Computing*, pages 25–34, London, UK, 2002. Springer-Verlag.

[51] E. Johnson and D. Gannon. Programming with the hpc++ parallel standard template library, 1997.

[52] Elizabeth Johnson and Dennis Gannon. HPC++ : Experiments with the parallel standard template library. In *International Conference on Supercomputing*, pages 124–131, 1997.

[53] Buisson Jérémy. Un modèle pour l'adaptation dynamique des programmes parallèles. In *RenPar'16/CFSE'4/SympAAA'2005/Journées Composants*, Le Croisic, France, Avril 2005.

[54] Lampros Kalampoukas, Dimitris Nikolos, Costas Efstathiou, Haridimos T. Vergos, and John Kalamatianos. High-speed parallel-prefix modulo 2n - 1 adders. *IEEE Trans. Comput.*, 49(7) :673–680, 2000.

[55] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71 :95–132, 1990.

[56] CLYDE P KRUSKAL, LARRY RUDOLPH, and MARC SNIR. The power of parallel prefix. *IEEE Transactions on Computers*, 34(10) :965–968, 1985.

[57] Bradley C. Kuszmaul. A segmented parallel-prefix vlsi circuit with small delays for small segments. In *SPAA '05 : Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 213–213, New York, NY, USA, 2005. ACM.

[58] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4) :831–838, 1980.

[59] LaMarca and Ladner. The influence of caches on the performance of sorting. In *SODA : ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[60] Yen-Chun Lin and Jun-Wei Hsiao. A new approach to constructing optimal parallel prefix circuits with small depth. *J. Parallel Distrib. Comput.*, 64(1) :97–107, 2004.

[61] Yen-Chun Lin and Chin-Yu Su. Faster optimal parallel prefix circuits : new algorithmic construction. *J. Parallel Distrib. Comput.*, 65(12) :1585–1595, 2005.

[62] Jigang Liu, Fenglien Lee, and Kai Qian. A parallel prefix convex hill algorithm using maspar. In *PDPTA '02 : Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1089–1095. CSREA Press, 2002.

[63] Frank Mueller. A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, pages 29–41, 1993.

[64] David R. Musser. Introspective sorting and selection algorithms. *Software - Practice and Experience*, 27(8) :983–993, 1997.

[65] David R. Musser, Gilmer J. Derge, and Atul Saini. *STL tutorial and reference guide, second edition*. Addison-Wesley, Boston, MA, USA, 2001.

[66] Yanik Ngoko. Les poly-algorithmes pour une programmation efficace des problèmes numériques. exemple du produit des matrices. Master's thesis, Laboratoire d'Informatique de Grenoble, Grenoble, 2005.

[67] Victor Y. Pan and Franco P. Preparata. Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput.*, 24(3) :811–821, 1995.

[68] Swann perraneau. Mesure, caractérisation et injection de charge à l'usage des machines parallèles. Master's thesis, Laboratoire d'Informatique de Grenoble, Grenoble, 2008.

[69] Brigitte Plateau, Anne Rasse, Jean-Louis Roch, and Jean-Pierre Verjus. Parallélisme, 1994. available at http ://www-id.imag.fr/Laboratoire/Membres/Roch_Jean-Louis/perso_html/polycops/poly-parallelisme-2a.pdf.

[70] James Reinders. *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[71] Rémi Revire. *Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée* . PhD thesis, Institut National Polytechnique de Grenoble, Septembre 2004.

[72] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15 :65–118, 1976.

[73] Jean-Louis Roch. Complexité parallèle et algorithmique pram. In Gérard Authié, Afonso Ferreira, Jean-Louis Roch, Gilles Villard, Jean Roman, Catherine Roucairol, and Bernard Virot, editors, *Algorithmes Parallèles : Analyse et Conception*, chapter 5, pages 105–126. Hermès, 1994.

[74] Jean-Louis Roch. Complexité parallèle, 1995. available at http ://www-id.imag.fr/Laboratoire/Membres/Roch_Jean-Louis/perso_html/polycops/poly-complexite-par.pdf.

[75] Jean-Louis Roch. Parallel efficient algorithms and their programming, 1997. available at http ://www-id.imag.fr/Laboratoire/Membres/Roch_Jean-Louis/perso_html/polycops/polycop-algo-par.pdf.

[76] Jean-Louis Roch. Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. In *Actes du Congrès International ALA 2001, Université Mohamm V*, pages 131–144, Rabat, Maroc, May 2001.

[77] Jean-Louis Roch and Daouda Traore. Un algorithme adaptatif optimal pour le calcul parallèle des préfixes. In INRIA, editor, *CARI'2006*, Cotonou, Benin, November 2006.

[78] Jean-Louis Roch, Daouda Traoré, and Julien Bernard. On-line adaptive parallel prefix computation. In LNCS 4128 Springer-Verlag, editor, *EUROPAR'2006*, pages 843–850, Dresden, Germany, August 2006.

[79] Jean-Louis Roch and Gilles Villard. Parallel computer algebra (tutorial). In *ISSAC '97 : Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, New York, NY, USA, July 1997. ACM Press.

[80] P. Sanders and S. Winkel. Super scalar sample sort. In *12th Annual European Symposium on Algorithms, ESA*, pages 14–17, Bergen, Norway, September 2004.

[81] Peter Sanders and Jesper Larsson Träff. Parallel prefix (scan) algorithms for mpi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 49–57. Springer Berlin Heidelberg, LNCS 4192, 2006.

[82] Hongzhang Shan, Jaswinder P. Singh, Leonid Oliker, and Rupak Biswas. A comparison of three programming models for adaptive applications on the origin2000. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 11, Washington, DC, USA, 2000. IEEE Computer Society.

[83] Johannes Singler and Benjamin Konsik. The gnu libstdc++ parallel mode : software engineering considerations. In *IWMSE '08 : Proceedings of the 1st international workshop on Multicore software engineering*, pages 15–22, New York, NY, USA, 2008. ACM.

[84] Johannes Singler, Peter Sanders, and Felix Putze. The multi-core standard template library. In Springer-Verlag LNCS 4641, editor, *Euro-Par 2007*, August 2007.

[85] Marc Snir. Depth-size trade-offs for parallel prefix computation. *J. Algorithms*, 7(2) :185–201, 1986.

[86] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch. Work stealing for time-constrained octree exploration : Application to real-time 3d modeling. In *EGPGV*, Lugano,Switzerland, May 2007.

[87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[88] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPoPP '05 : Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288, New York, NY, USA, 2005. ACM.

[89] Daouda Traoré, Jean-Louis Roch, and Christophe Cérin. Algorithmes adaptatifs de tri parallèle. In *RenPar'18 / SympA'2008 / CFSE'6*, Fribourg, Switzerland, Feb. 2008.

[90] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. Deque-free work-optimal parallel stl algorithms. In LNCS Springer-Verlag, editor, *EUROPAR'2008*, Canary Island, Spain, August 2008.

[91] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'03)*, pages 372–381, Genova, Italy, February 2003. IEEE Computer Society.

[92] H. T. Vergos, D. Nikolos, and C. Efstathiou. High speed parallel-prefix modulo 2n+1 adders for diminished-one operands. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH '01)*, pages 211–217, 2001.

[93] David W. Walker and Jack J. Dongarra. Mpi : a standard message passing interface. *Supercomputer*, 12 :56–68, 1996.

[94] Haigeng Wang, Alexandru Nicolau, and Kai-Yeng S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Transactions on Computers*, 45(11) :1257–1271, 1996.

[95] Site web de intel tbb. http ://www.threadingbuildingblocks.org/.

[96] Site web de SOFA. http ://www.sofa-framework.org/.

[97] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2) :3–35, 2001.

[98] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Trans. Par. Dist. Syst.*, 17(10) :1084–1096, 2006.

[99] Haikun Zhu, Chung-Kuan Cheng, and Ronald Graham. On the construction of zero-deficiency parallel prefix circuits with minimum depth. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2) :387–409, 2006.

Résumé :

Cette thèse porte sur la construction d'algorithmes et de programmes parallèles qui s'adapte automatiquement à la plate-forme d'exécution (nombre de processeurs, vitesses des processeurs, ...) et ce, de manière dynamique inconsciente (en anglais *oblivious*). La construction que nous proposons est basée sur la technologie développée au sein de l'équipe Moais consistant au couplage récursif et dynamique : d'un algorithme séquentiel (qui minimise le nombre d'opérations, mais pas le temps parallèle) ; et d'un algorithme parallèle à grain fin (qui minimise le temps parallèle sur un nombre non borné de ressources, mais pas le nombre d'opérations). Les deux algorithmes sont entrelacés à la volée par un ordonnancement à grain fin de type vol de travail. Outre une analyse théorique du couplage (borne inférieure, optimalité asymptotique), nous proposons une implantation " générique " que nous instancions sur différents exemples (un nouvel algorithme parallèle adaptatif de calcul des préfixes, algorithmes adaptatifs de fusion, de partition et tris, plusieurs algorithmes adaptatifs de la librairie standard C++). Dans cette thèse, nous proposons aussi un nouvel algorithme parallèle statique optimal du calcul des préfixes.

Mots clés : algorithme parallèle, parallélisme, ordonnancement par vol de travail, préfixe.

Abstract :

This thesis focuses on the building of algorithms and parallel programs that obliviously adapts the execution platform (number of processors, speeds of processors, ...). The building that we propose is based on the technology developed in the Moais team ; it consists on the recursive and dynamic coupling of two algorithms : one sequential that minimizes the work (the number of operations), but not the parallel time ; one parallel that minimizes the depth (parallel time) on an unbounded number of resources, but not the work. The two algorithms are interleaved on-line based on a work-stealing schedule. The contribution of this thesis consists in a theoretical analysis of coupling (lower bound, asymptotic optimality in some cases), and a "generic" implementation of the coupling scheme witch is applied to several examples (new on-line adaptive parallel prefix computation, adaptive merging and sorting, adaptive parallelization of several STL algorithms). In this thesis, we also propose a new strict optimal parallel static algorithm for prefix computation.

Keywords : parallel algorithm, parallelism, work stealing, prefix.