UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité Informatique

Arrêté ministériel : 7 août 2006

Présentée et soutenue publiquement par

Marc Tchiboukdjian

le 9 décembre 2010

Algorithmes parallèles efficaces en cache Applications à la visualisation scientifique

Thèse dirigée par Denis Trystram et codirigée par
Vincent Danjean, Jean-Philippe Nominé et Bruno Raffin

JURY

M. Christoph Dürr	DR CNRS	Président
M. Jean-Michel Dischler	Prof. Université de Strasbourg	Rapporteur
M. Enrique S. Quintana Ortì	Prof. Universidad Jaime I	Rapporteur
M. Vincent Danjean	MdC Université Joseph Fourier	Examinateur
M. Jean-Philippe Nominé	Chercheur CEA,DAM,DIF	Examinateur
M. Bruno Raffin	CR INRIA	Examinateur
M. Denis Trystram	Prof. Grenoble INP	Examinateur
M. Frédéric Vivien	DR INRIA	Examinateur

Thèse préparée au sein du Laboratoire d'Informatique de Grenoble dans l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Sommaire

Sommaire iii

Introduction 1

I État de l'art 13

1 Applications limitées par les accès mémoire 15

2 Algorithmes utilisant efficacement les caches 25

3 Programmation parallèle par vol de travail 39

4 Algorithmes parallèles efficaces en cache 61

II Contributions 75

5 Maillage cache-oblivious pour la visualisation scientifique 77
Binary Mesh Partitioning for Cache-Efficient Visualization 81

6 Extraction d'isosurfaces parallèle et efficace en cache 95
Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores . 103

7 Vol de travail efficace en cache pour les boucles parallèles 113
A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores . 119

8 Nouvelle analyse des ordonnancements par vol de travail 129
Decentralized List Scheduling 133

Conclusion 155

Bibliographie 159

Table des figures 169

Table des matières 171

Résumés 177

Introduction

Le classement du Top500 [1] reporte deux fois par an les performances atteintes par les 500 calculateurs les plus puissants au monde sur une application de référence, le LINPACK [DLP03]. Lors du dernier classement en juin 2010, la machine la plus puissante a atteint la vitesse de $1,759$ petaflops [2] à l'aide de 224162 cœurs. Les performances de la meilleure machine sont multipliées par environ 1000 tous les 10 ans ce qui permet de prévoir que la puissance d'un exaflops sera vraisemblablement atteinte dès 2018.

Un exemple de supercalculateur est la machine TERA-100 du CEA DAM qui s'architecture comme suit. La brique de base est le nœud de calcul composé de 4 processeurs octocœurs Intel Nehalem EX à 2.27 Ghz. Ces nœuds sont ensuite regroupés par paquets de 324 en un "ilôt" et connectés par un switch de bande passante 51.8 Tbps. Les 14 ilôts communiquent entre eux par l'intermédiare de 27 switch de 36 ports chacun ayant une bande passante de 2.88 Tbps. Au total, la machine TERA-100 contient 138000 cœurs donnant une puissante théorique crête de $1,25$ pétaflops, 291 To de mémoire, consomme mois de 6 MW et occupe 750m^2.

Dans une telle machine, on peut distinguer deux niveaux suivant le mode de communication entre les cœurs. Tous les cœurs à l'échelle d'un nœud accèdent la même mémoire et peuvent donc partager des données et coopérer très rapidement par des lectures et écritures en mémoire. A l'échelle de l'ilôt, les cœurs communiquent par messages sur le réseau et les données doivent être distribuées sur les différentes zones de mémoire.

Exploiter efficacement la puissance d'une telle machine soulève principalement deux défis :

1. *le parallélisme :* répartir de manière équilibrée les calculs sur plus de 100000 cœurs,
2. *la hiérarchie mémoire :* la vitesse des accès mémoire et des communications est fonction de la distance.

Pour réduire les temps de communications entre les cœurs et la latence des accès à la mémoire, la structure d'une application doit reflèter la hiérarchie de l'organisation de la machine. Les parties de code séquentielles sont optimisées pour tirer parti des différents niveaux de caches. L'exécution parallèle profite du cache partagé à l'échelle d'un processeur et de la mémoire partagée à l'échelle d'un nœud. Les cœurs ne communiquent par le réseau que pour sortir d'un nœud et on essaye de réduire les communications à l'échelle de la machine au profit des communications à l'échelle de l'ilôt.

Lorsque le comportement de l'application est statique et connu avant l'exécution, il est déjà difficile pour le programmeur de concevoir et d'optimiser le code à tous ces niveaux. De plus, du bruit inhérent à la taille d'une telle machine vient compromettre l'équilibre parfait prévu et espéré par le programmeur [Sni09]. Enfin, si le comportement

1. `http://www.top500.org`
2. Un petaflop correspond à 10^{15} opérations flottantes par seconde

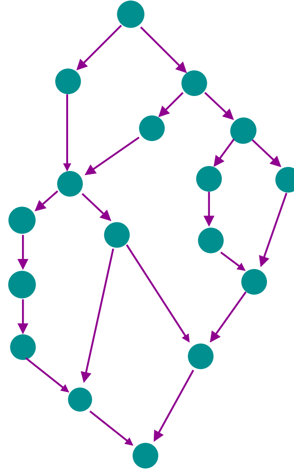


FIGURE 1 – Programme parallèle à base de tâches. L'exécution des tâches doit respecter les contraintes de précédence.

même de l'application devient dynamique, il est extrèmement complexe d'adapter la décomposition de l'application pour garder un bon équilibrage de charge et une bonne localité des accès mémoire et des communications.

Idéalement, on aimerait pouvoir cacher cette complexité au programmeur à travers une interface de programmation simple mais donnant suffisament d'informations sur le comportement de l'application. Un moteur d'exécution s'occupe alors de décomposer l'application en fonction de l'architecture de la machine et gère la répartition de charge et les communications en optimisant la localité. On fournit au programmeur un modèle de machine abstraite exprimant les contraintes du parallélisme et de la localité pour guider ses choix lors de la conception. Si l'application respecte les contraintes de performances du modèle, elle s'exécute alors aussi efficacement qu'une application spécifiquement conçue et optimisée pour la machine cible. Les avantages d'une telle approche sont un code plus simple et des applications portables qui peuvent s'adapter à un comportement dynamique (du matériel ou de l'application).

A l'heure actuelle, il est difficile d'envisager une telle interface et son moteur associé qui soit capable de s'exécuter efficacement à l'échelle de la machine entière. La programmation pour mémoire distribuée nécessite la description des données dans l'interface et on ne dispose pas de modèles généraux et efficaces. De plus, une décomposition de l'application résultant en une localité non optimale est très coûteuse à cause de la lenteur des communications sur le réseau. Les approches actuelles ont une efficacité qui est limitée à des cas particuliers : peu de transferts de données [GRCD98] ou applications très régulières [GBH^{+}06].

Pour la programmation parallèle en mémoire partagée, il n'est pas nécessaire de décrire les données et de gérer les communications. L'interface est donc plus simple. On doit quand même gérer les accès à la mémoire mais ils sont moins coûteux que les communications donc un comportement non optimal n'empèche pas d'obtenir de bonnes performances.

Une interface simple pour décrire le parallélisme d'une application est la programmation à base de tâches (*cf.* figure 1). L'application crée des tâches qui décomposent

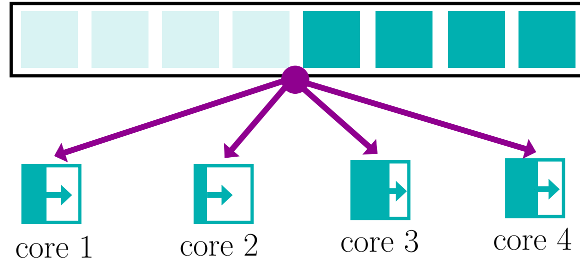


FIGURE 2 – Ordonnancement par liste centralisée. La contention lors de l'accès à la liste limite le passage à l'échelle.

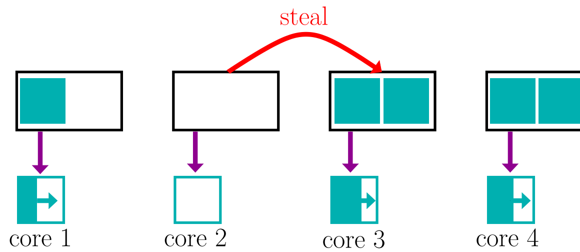


FIGURE 3 – Ordonnancement par vol de travail. La liste est décentralisée et cela réduit fortement la contention.

le calcul en petites parties séquentielles. Elle déclare aussi des précédences entre ces tâches pour déterminer à partir de quand une tâche peut être exécutée, par exemple si cette tâche dépend du résultat d'autres tâches. Le moteur exécutif est responsable de l'ordonnancement de ces tâches. Il décide à quel moment et sur quel cœur exécuter chaque tâche. Pour une exécution efficace, le moteur doit trouver un ordonnancement qui minimise le temps de complétion de l'application. De plus, le coût de calcul de cet ordonnancement doit être faible afin de laisser le plus de ressources possibles à l'application.

Un ordonnancement efficace et rapide à obtenir est donné par l'algorithme de liste de Graham [Gra69]. Toutes les tâches prêtes, c'est-à-dire dont les précédences sont satisfaites, sont stockées dans une liste (*cf.* figure 2). Dès qu'un cœur est inactif, il retire une tâche de cette liste et l'exécute. Les nouvelles tâches générées sont ajoutées à la liste. L'ordonnancement généré par cet algorithme est proche de l'optimal. Si on note W le travail total de toutes les tâches et D le travail des tâches sur un plus long chemin de précédences, le temps d'exécution T_m de l'application sur m cœurs est majoré par

$$T_m \leq \frac{W}{m} + \left(1 - \frac{1}{m}\right) \cdot D. \tag{1}$$

Les deux termes de cette équation, W/m et D, sont des bornes inférieures du temps d'exécution. Cet ordonnancement est donc au plus à un facteur 2 de l'optimal. Cependant, l'implémentation d'une telle liste de tâches n'est pas efficace en pratique. En effet, cette liste est centralisée ce qui provoque de la contention lors de son accès en concurrence par de nombreux cœurs.

La technique clé qui permet de construire un moteur d'exécution efficace pour les programmes parallèles à base de tâches est le vol de travail [BL99]. A la place d'une liste de tâches centralisée, chaque cœur maintient ses tâches à exécuter dans une liste

locale (*cf.* figure 3). Lorsque qu'une liste devient vide, un cœur tente de voler des tâches dans la liste d'un autre cœur sélectionné au hasard. Comme les opérations de vol de tâches, c'est-à-dire l'accès à une liste distante, sont peu fréquentes, il y a peu de chance que deux cœurs accèdent à la même liste au même moment ce qui réduit fortement la contention. De plus, le vol de travail génère un ordonnancement aussi bon que l'algorithme de liste de Graham. Blumofe *et al.* [BL99, ABP98] ont montré que le temps d'exécution était borné par

$$T_m \leq \frac{W}{m} + O(D) \tag{2}$$

en prenant en compte le coût du calcul de l'ordonnancement.

L'équation (2) permet de guider le programmeur lors de la conception d'un algorithme parallèle. L'algorithme parallèle doit à la fois minimiser le travail, c'est-à-dire le nombre de calculs à effectuer W, mais aussi minimiser la profondeur D du calcul, c'est-à-dire le nombre maximal d'instructions sur un chemin de précédences. On définit le parallélisme de l'algorithme comme le rapport du travail sur la profondeur W/D. Si le parallélisme est très supérieur au nombre de processseurs, c'est-à-dire $W/D \gg m$, alors l'exécution de l'application avec un moteur de vol de travail sera très efficace. En effet, la borne sur le temps d'exécution donnée par l'équation (2) devient

$$T_m \leq (1 + o(1)) \cdot \frac{W}{m}. \tag{3}$$

Le surcoût par rapport à une décomposition parfaite et statique du travail W/m est donc très faible. De plus, dans le cas du programme parallèle par tâches, cette décomposition n'est pas fonction de l'architecture de la machine (nombre et vitesses des cœurs). L'application est donc portable et peut s'adapter à un comportement dynamique. On qualifie de *processor-aware* un programme parallèle dont la décomposition est fonction du nombre de cœurs. Lorsque la décomposition du calcul n'est pas fonction du nombre de cœurs, comme pour le programme par tâches, le programme est dit *processor-oblivious*. Un moteur à base de vol de travail permet d'exécuter un programme *processor-oblivious* exposant beaucoup de parallélisme aussi efficacement qu'un programme *processor-aware*.

L'interface de programmation parallèle par tâches associée à un ordonnancement par vol de travail s'est imposée comme le standard pour la programmation efficace en mémoire partagée. Une des premières implémentations est le logiciel Cilk du MIT [BJK+96] en 1994. Cette technique a été reprise dans les logiciels TBB de Intel [KV07], Cilk++ startup du MIT rachetée par Intel, TPL de Microsoft [LSB09], etc. La programmation par tâches a fait son apparition dans le standard OpenMP [Ope] à partir de la version 3.

Le deuxième défi à relever pour exploiter efficacement les performances des supercalculateurs est la prise en compte de la hiérarchie mémoire. Pour réduire la latence des accès à la mémoire, les processeurs contiennent plusieurs niveaux de caches entre les cœurs et la mémoire centrale (*cf.* figure 4). Chaque niveau de cache est plus rapide que le précédent mais a une capacité plus faible. Les données transitent entre ces caches par blocs contigus appelés lignes de cache. Les processeurs de la machine TERA-100 ont 3 niveaux de cache de tailles 32KO, 256KO et 24MO et des lignes de cache de 64O.

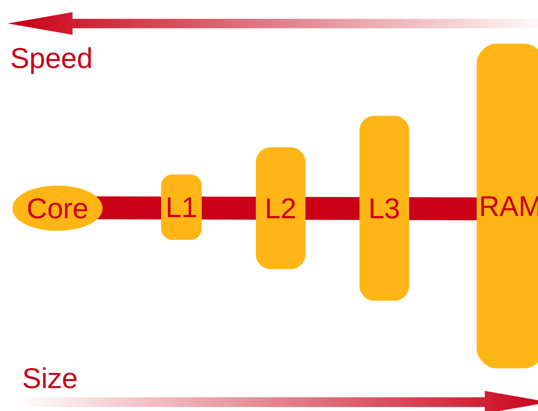


Figure 4 – Hiérarchie mémoire composée de 3 niveaux de cache et de la mémoire centrale.

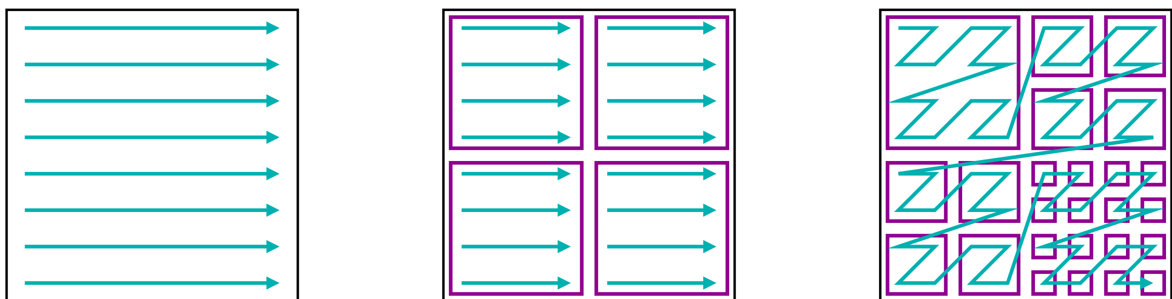


Figure 5 – Trois schémas d'accès. Le schéma de gauche n'est pas local. Le schéma du milieu est *cache-aware*, c'est-à-dire optimisé pour une taille de cache précise. Le schéma de droite est *cache-oblivious*, c'est-à-dire optimisé pour toutes les tailles de cache.

Pour tirer parti de ses caches, les accès mémoire de l'application doivent être locaux : réutiliser la même donnée ou une donnée proche dans la mémoire dans un court laps de temps.

Pour améliorer la localité des accès mémoire d'une application, on peut décomposer le calcul de la façon suivante (*cf.* figure 5) :

1. charger un bloc de données correspondant à la taille du cache,
2. exécuter tous les calculs utilisant ces données.

Lorsqu'il y a plusieurs niveaux de cache, on décompose le calcul de façon hiérarchique : on charge un bloc de données correspondant à la taille du plus gros cache, puis un bloc de données correspondant à la taille du niveau inférieur, etc. Cette technique est utilisée, par exemple, dans l'implémentation des BLAS pour les calculs d'algèbre linéaire sur les matrices denses [WP05]. On qualifie les algorithmes utilisant cette technique d'algorithmes *cache-aware* [AV88] car ils utilisent la taille des caches. Les inconvénients de cette approche *cache-aware* sont identiques à ceux de l'approche *processor-aware* : le code devient complexe dès qu'il y a plusieurs niveaux de cache, l'application n'est pas portable et ne peut pas s'adapter à une variation des paramètres (*e.g.* réduction de la taille du cache due à la présence d'une autre application ou pour économiser de l'énergie). Une difficulté supplémentaire à laquelle les programmeurs doivent faire face

dans le cas des algorithmes *cache-aware* réside dans la taille des blocs de données : la taille optimale ne correspond pas toujours à la taille du cache. Il est nécessaire de recourir à l'expérimentation pour trouver la taille de blocs optimale [WP05].

Pour pallier les inconvénients de l'approche *aware*, on peut utiliser une approche similaire à celle utilisée pour le parallélisme. L'approche est beaucoup plus simple pour les caches car ils sont gérés de manière transparente par le matériel. Ce sont des fonctions internes au processeur qui décident quelles données charger dans quel cache. Les données répondent aux accès mémoire de façon transparente qu'elles soient en cache ou non. L'étape numéro 1 de la technique *cache-aware*, charger un bloc de données dans le cache, n'est pas réalisée par l'application mais par le matériel. On n'a donc pas besoin ici d'une interface ni de moteur d'exécution.

De la même façon que la programmation par tâches décompose le calcul au grain le plus fin sous forme de tâches pour pouvoir s'exécuter efficacement quel que soit le nombre de processeurs, on peut décomposer les accès mémoire au grain le plus fin pour obtenir une bonne localité à toutes les échelles (*cf.* figure 5). Un exemple de technique qui permet de s'affranchir de la taille des caches dans la décomposition du calcul par blocs est de découper le calcul en blocs de taille moitié récursivement. De tels algorithmes qui n'utilisent pas les paramètres des caches sont qualifiées de *cache-oblivious* [FLPR99].

Le modèle *cache-oblivious* permet de guider le programmeur lors de la conception d'un algorithme. Ce modèle est composé de deux niveaux de mémoire, un niveau rapide mais de capacité limitée M et un niveau lent de capacité infinie. Les données sont transférées entre ces deux niveaux par blocs contigus de taille B. Une particularité intéressante de ce modèle est d'interdire à l'algorithme d'utiliser ces paramètres M et B. Sous certaines conditions de régularité des accès mémoire, on peut montrer qu'un algorithme optimal dans le modèle *cache-oblivious* est optimal à tous les niveaux de la hiérarchie mémoire quelque soient le nombre de niveaux et la taille des caches.

Un avantage supplémentaire des techniques *cache-oblivious* apparait lors de l'utilisation de nœuds de calculs hybrides contenant des processeurs classiques et des processeurs graphiques (GPU). Une structure de donnée optimisée dans le modèle *cache-oblivious* peut être partagée par ces deux types de processeurs ayant des hiérarchies de cache très différentes sans avoir besoin de l'optimiser pour chacun des processeurs séparemment.

Cependant, il n'est pas toujours possible d'obtenir un algorithme *cache-oblivious* aussi performant qu'un algorithme *cache-aware*, soit pour des raisons d'impossibilité théorique soit parce que les algorithmes *cache-oblivious* connus ne sont pas efficaces en pratique, souvent à cause d'un surcoût en nombre d'instructions.

Les techniques *oblivious* pour la hiérarchie mémoire et pour le parallélisme permettent de limiter l'impact de la complexité de l'architecture sur le code de l'application. De plus, les applications obtenues sont portables et peuvent s'adaptater a une variation des paramètres de l'architecture.

Pour développer des applications parallèles et efficaces en cache, nous proposons de combiner les approches *processor-oblivious* et *cache-oblivious*.

Deux difficultés majeures rendent ce couplage non trivial. La hiérarchie mémoire est plus complexe en parallèle qu'en séquentiel. En effet, en parallèle, plusieurs niveaux de caches séparent les cœurs de la mémoire et ces niveaux peuvent être :

– privés : chaque cœur possède son propre cache, ou
– partagés : plusieurs cœur partagent un même cache.

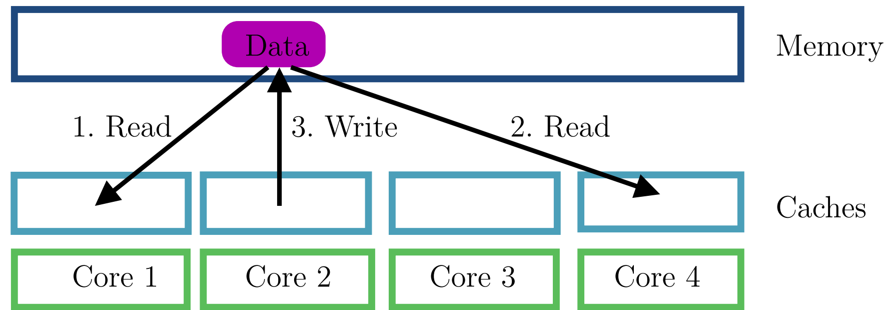


Figure 6 – Le contenu d'un cache privé n'est pas indépendant des actions des autres cœurs. Ici le cœur 2 modifie des données qui ont été préalablement chargées dans les caches des cœurs 1 et 4. Pour garder les caches et la mémoire en cohérence, les données des cœurs 1 et 4 sont invalidées. Une lecture de la même donnée par le cœur 1 nécessitera un accès à la mémoire et non pas au cache pour obtenir une version à jour.

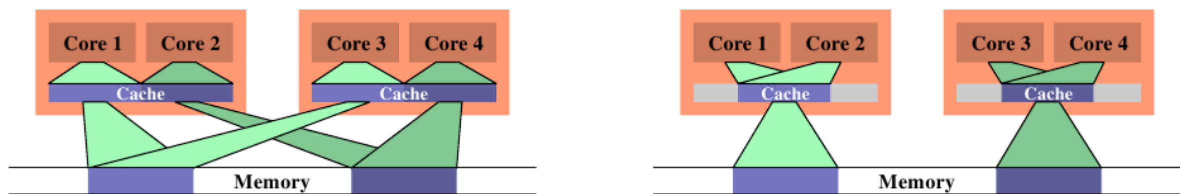


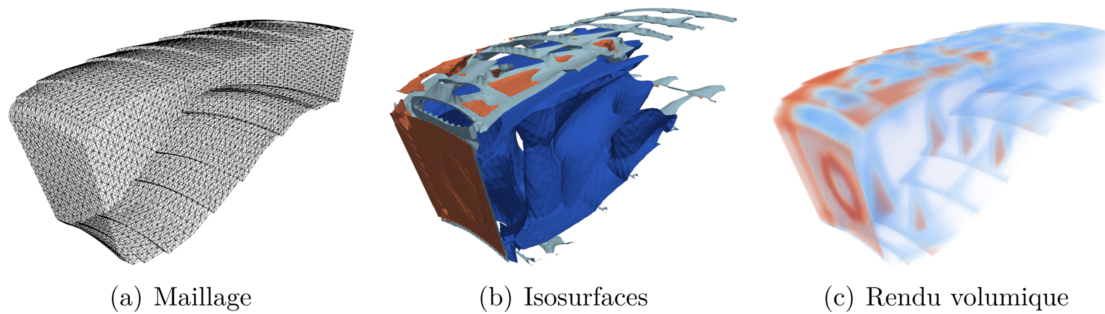
Figure 7 – Influence de l'ordonnancement sur l'utilisation du cache. L'ordonnancement de droite est plus efficace que l'ordonnancement de gauche car les tâches utilisant des données communes sont exécutées sur des cœurs partageant le même cache. Images de [Dre07].

Il faut donc adapter le comportement de l'algorithme en fonction du type de cache. De plus, l'état du cache d'un cœur n'est pas indépendant du comportement des autres cœurs, même dans le cas des caches privés à cause du protocole de cohérence de cache (voir la figure 6).

Il faut adapter l'algorithme mais aussi l'ordonnanceur car la politique d'ordonnancement des tâches a un fort impact sur le comportement du cache (voir la figure 7). En particulier les stratégies de vol qui sont efficaces pour minimiser le coût de l'ordonnanceur ne sont pas les mêmes que les stratégies qui maximisent l'utilisation des caches, surtout pour les caches partagés. Une politique d'ordonnancement efficace s'efforcera de ne pas perturber les caches privés tout en garantissant un usage coopératif des caches partagés.

Afin de vérifier expérimentalement l'efficacité des politiques d'ordonnancement que nous proposons dans cette thèse, nous avons utilisé ces techniques pour optimiser plusieurs applications du domaine de la visualisation scientifique. Les applications de visualisation scientifique traitent de grosses masses de données et sont donc gourmandes à la fois en accès mémoire et en puissance de calculs. Mieux tirer parti du parallélisme et de la hiérarchie mémoire permet donc d'améliorer les performances de ces applications.

La simulation numérique consiste à calculer par ordinateur l'évolution d'un système physique complexe dont on connait un modèle théorique mais qui est difficile ou impossible à résoudre analytiquement. Elle prend une importance grandissante dans de



(a) Maillage (b) Isosurfaces (c) Rendu volumique

Figure 8 – Simulation de la densité d'essence au sein d'une chambre de combustion. De gauche à droite, le maillage, une représentation graphique de la densité d'essence avec 3 isosurfaces et avec un rendu volumique. (maillage de VTK, courtoisie de W. J. Schroeder).

nombreux domaines scientifiques ainsi que dans l'industrie. Un exemple intéressant de simulation numérique à grande échelle est la simulation de l'univers depuis le big bang par le projet Horizon [3]. Cette simulation a durée deux mois sur les 6144 processeurs d'un supercalculateur du centre de calcul du CEA (CCRT) et a généré 50To de données.

La visualisation scientifique fournit sous une forme graphique les résultats de la simulation et aide ainsi à la compréhension du phénomène simulé. C'est aujourd'hui un outil indispensable pour analyser les grandes quantités de données produites par ces simulations. Pour répondre aux utilisateurs en attente d'une interaction fluide lors de l'analyse de ces données, il faut des algorithmes performants adaptés aux architectures des processeurs modernes.

Une représentation graphique de données scientifiques s'obtient généralement en appliquant un ou plusieurs filtres sur un maillage [HJ04]. Un maillage est une représentation discrète pour stocker une fonction de l'espace tridimensionnel ou champ (voir la figure 8(a)). L'espace est divisé en unités élémentaires appelée cellules (*e.g.* tétraèdres, hexaèdres, etc.), chaque cellule étant définie par une liste de points. Chaque point du maillage a une position dans l'espace ainsi que la valeur du champ en ce point. La valeur de la fonction dans le reste de l'espace du maillage est reconstruite par interpolation linéaire à l'intérieur de chaque cellule.

Les filtres de visualisation de champs scalaires 3D les plus classiques sont l'extraction d'isosurfaces (voir la figure 8(b)) et le rendu volumique (voir la figure 8(c)). Une isosurface est l'ensemble des points de l'espace ayant la même valeur de champ. Une isosurface est habituellement représentée par une surface triangularisée.

Pour générer un rendu volumique, on associe à chaque valeur du champ scalaire une couleur et une opacité par l'intermédiaire de fonctions de transfert. La couleur d'un pixel de l'image finale correspond à l'accumulation des valeurs de couleur atténuée par les valeurs d'opacité sur la trajectoire d'un rayon lancé à partir du point de vue et passant par le centre du pixel.

3. `http://www.projet-horizon.fr`

Contributions

Nos contributions s'articulent principalement autour de deux points.

– Nous appliquons tout d'abord les techniques *cache-oblivious* pour améliorer la localité mémoire des filtres de visualisation scientifique.
– Nous montrons ensuite comment modifier l'ordonnancement de tels filtres pour que l'exécution parallèle conserve la localité de l'exécution séquentielle.

Afin d'améliorer la localité des filtres de visualisation, nous proposons de réorganiser la structure de données représentant le maillage. En effet, c'est sur cette structure que se concentrent la majorité des accès mémoire. Il existe deux grandes familles de maillages : les maillages réguliers et les maillages irréguliers. La structure de données représentant les maillages réguliers est similaire à celle des matrices denses. Les techniques *cache-oblivious* utilisées pour l'optimisation des calculs d'algèbre linéaire dense (*e.g.* les *space-filling curves*) s'adaptent parfaitement aux maillages réguliers [PF01] Le cas des maillages irréguliers est plus difficile. Yoon *et al.* proposent dans [YLPM05] un algorithme de réorganisation de maillages irréguliers appelé OpenCCL. Cet algorithme donne de bons résultas expérimentaux mais sans garantie théorique. Nous proposons une alternative appelée FastCOL qui permet d'obtenir un niveau de performance équivalent mais avec une garantie théorique. Cette organisation est obtenue par une découpe récursive du maillage sous forme d'arbre de BSP (*Binary Space Partitioning*) utilisant le séparateur de Miller *et al.* [MTTV98]. Nous donnons une garantie théorique sur le nombre de défauts de cache générés dans le cas de schémas d'accès classiques. Lorsque le schéma d'accès du filtre respecte la localité définie par les relations de voisinage du maillage, nous montrons que le nombre de défauts de cache sur un maillage de taille S est borné par

$$\frac{S}{B} + O\Big(\frac{S}{M^{1/3}}\Big)$$

avec B la taille des lignes de cache et M la taille du cache. Le premier terme représente le nombre de défauts de cache nécessaire pour lire entièrement le maillage avec une localité spatiale parfaite. Le deuxième terme est le surcoût induit par la découpe des cellules lors de l'utilisation du séparateur. Cette garantie est valable quel que soit la valeur de ces paramètres et est donc *cache-oblivious*. Des expérimentations sur de nombreux filtres de visualisation usuels confirment des gains de performance significatifs sur CPU et sur GPU.

Certains filtres de visualisation utilisent également une structure d'arbre pour accélérer les traitements sur le maillage. Nous montrons qu'il est possible d'améliorer la localité de tels filtres en utilisant la même structure que l'arbre de BSP calculé lors de la réorganisation. Le schéma d'accès de ces filtres est alors adapté au maillage généré par FastCOL ce qui améliore la localité. Nous appliquons cette technique à l'arbre min-max qui permet d'accélérer l'extraction d'isosurfaces. Des expériences montrent le gain de performance par rapport à l'utilisatoin d'un arbre géométrique standard (*e.g. kd-tree* ou *BSP tree*). Cette technique permet en outre, dans le cas particulier de l'arbre min-max, de réduire la consommation mémoire.

Nous montrons ensuite comment conserver la localité des accès mémoire lors d'une exécution parallèle. Nous étudions le cas particulier des boucles parallèles où chaque itération peut être traitée indépendamment. Ce schéma parallèle est très courant, particulièrement dans les filtres de visualisation scientifique qui appliquent souvent

une même opération à chaque élément du maillage. Les ordonnancements par vol de travail permettent de paralléliser les boucles avec très peu de surcoût par rapport à une implémentation statique *processor-aware*. Cependant, nous montrons que de tels ordonnancements génèrent plus de défauts de cache que l'exécution séquentielle sur un cache partagé. Si l'exécution séquentielle cause $Q_{\text{seq}}(M)$ défauts de cache sur un cache de taille M, l'exécution parallèle standard sur p cœurs partageant un cache de taille M cause autant de défauts de cache que l'exécution séquentielle sur un cache de taille M/p :

$$Q_{\text{par}}(M) = Q_{\text{seq}}\left(\frac{M}{p}\right).$$

En effet, les cœurs travaillent sur des données très éloignées en mémoire et sont donc en compétition pour l'utilisation du cache. Ils n'obtiennent chacun qu'une fraction M/p du cache partagé. Pour favoriser l'usage coopératif du cache partagé, nous proposons un nouvel ordonnancement des boucles parallèles par vol de travail qui incite les cœurs à accéder à des données proches en mémoire de celles déjà chargées dans ce cache partagé. Cet ordonnancement est basé sur une fenêtre de la taille du cache partagé glissant sur les données. Une modification de l'algorithme de vol contraint tous les cœurs à travailler sur des données à l'intérieur de la fenêtre et donc dans le cache. Nous montrons qu'une telle exécution n'induit q'une augmentation légère des défauts de cache par rapport à l'exécution séquentielle. Des expériences sur les filtres d'extraction d'isosurfaces confirment l'analyse théorique. De plus, l'implémentation de l'ordonnancement avec fenêtre est très efficace et ne génère que peu de surcoût par rapport à l'implémentation sans fenêtre.

Enfin ce travail nous a conduit à développer une nouvelle analyse théorique du vol de travail qui vise à mieux comprendre le nombre de vols S lors d'une exécution par vol de travail. En effet, l'analyse de Blumofe *et al.* [ABP98] ne donne qu'une borne très large sur le nombre de vols avec un facteur constant important, $S = O(m \cdot D)$ avec m le nombre de cœurs et D la profondeur du graphe de précédences. De plus, cette analyse ne s'applique qu'à une classe restreinte de graphes de précédences ne comprenant pas directement les boucles parallèles. Notre analyse caractérise précisément le nombre de vols S. Des simulations montrent que la constante obtenue est à moins de 40% de la valeur théorique. En outre, elle ne se base pas sur la profondeur du graphe mais sur l'équilibrage de charge au moment du vol. Elle peut donc s'appliquer directement au cas des boucles parallèles et permet également d'évaluer l'efficacité de nouveaux mécanismes de vols (comme la combinaison des requêtes de vols).

Contexte

J'ai étudié les algorithmes *cache-oblivious* lors d'un master réalisé à l'université de Stony Brook aux États-Unis sous l'encadrement du professeur Michael Bender, spécialiste en ordonnancement et en algorithmique.

Les applications en visualisation scientifique sont le fruit d'une collaboration avec l'équipe de visualisation scientifique du département des sciences de la simulation et de l'information du CEA de Bruyères-Le-Chatel. C'est Jean Philippe Nominé qui m'a encadré sur les problématiques de visualisation de grosses masses de données au CEA.

La plupart de ce travail s'est déroulé dans l'équipe projet MOAIS [4] qui fait partie de l'INRIA et du laboratoire d'informatique de Grenoble [5]. Les thèmes de recherches de l'équipe en relation avec cette thèse sont l'ordonnancement, l'algorithmique et la programmation parallèle. En plus de l'encadrement de Vincent Danjean et Bruno Raffin sur toutes ces thématiques ainsi que sur la visualisation scientifique, j'ai travaillé en collaboration avec :

- Thierry Gautier et Fabien Le Mentec sur les aspects de performances en pratique du vol de travail et en particulier sur le moteur exécutif de vol de travail X-Kaapi,
- Nicolas Gast, Jean-Louis Roch, Denis Trystram et Frédéric Wagner sur l'analyse théorique du vol de travail.

Pendant cette thèse, j'ai aussi eu l'occasion d'encadrer et de travailler avec plusieurs stagiaires :

- Bruno Berthier sur les arbres min-max pour l'extraction d'isosurface,
- Siméon Marijon sur les organisations mémoire de maillage spécifiques aux GPU,
- Aymerick Chincolla, Adrien Graton et Matthieu Westphal sur le rendu volumique par lancer de rayons,
- Thibault Bruguet sur les algorithmes parallèles pour cache partagé.

Organisation du manuscrit

Le manuscrit est composé de deux parties. La première partie intitulée "État de l'art" invoque les notions générales et les résultats connus à travers les 4 premiers chapitres. Dans le chapitre 1, nous étudions l'impact des accès mémoire sur le temps d'exécution d'une application et les techniques pour réduire le temps d'accès à la mémoire. Dans le chapitre 2, nous passons en revue les modèles *cache-aware* et *cache-oblivious*. Dans le chapitre 3, nous présentons la programmation parallèle à base de tâches et l'ordonnancement par vol de travail. Dans le chapitre 4, nous analysons les différents impacts du parallélisme sur les caches. Nous présentons les résultats existants sur l'ordonnancement et les algorithmes parallèles pour les caches privés et les caches partagés.

La deuxième partie intitulée "Contributions" est constituée des 4 publications réalisées dans le cadre de cette thèse. Dans l'article *Binary Mesh Partitioning for Cache-Efficient Visualization* [TDR10a], nous présentons notre nouvelle organisation mémoire *cache-oblivious* FastCOL. Dans l'article *Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores* [TDR10b], nous étudions le cas particulier de l'extraction d'isosurface. Nous montrons comment adapter l'arbre min-max au maillage FastCOL et proposons un schéma de parallélisation *processor-aware* qui ne génère pas plus de défauts de cache que l'exécution séquentielle. Dans l'article *A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores* [TDG$^+$10], nous rendons *processor-oblivious* le schéma proposé dans l'article précédent en utilisant du vol de travail. Nous montrons également que cet ordonnancement garantit un nombre de défauts de cache équivalent à l'exécution séquentielle dans le cas général et non pas uniquement dans le cadre des filtres de visualisation basé sur le maillage FastCOL. Dans l'article *A Tighter Analysis of Work Stealing* [TGT$^+$10], nous donnons une nouvelle analyse plus précise

4. `http://moais.imag.fr`
5. `http://www.liglab.fr`

du vol de travail qui permet d'étudier des modifications fines de l'ordonnanceur.

Nous terminons ce manuscrit par un bilan des travaux effectués et une suggestion des pistes qui sont, à notre avis, les plus intéressantes à poursuivre.

État de l'art

Applications limitées par les accès mémoire 1

Sommaire

1.1 Les caches **15**
1.1.1 Définition 15
1.1.2 La localité des accès mémoire 16
1.1.3 Fonctionnement d'un cache 16
1.1.4 Classification des défauts de cache 17
1.2 Caractérisation des applications **17**
1.2.1 Applications limitées par les accès mémoire 17
1.2.2 Prédictibilité des accès mémoire et *prefetching* 18
1.2.3 Applications limitées par la bande passante ou la latence . . 19
1.2.4 Réutilisation et localité temporelle 19
1.3 Les principes pour concevoir une application efficace en cache **20**
1.3.1 Les 3 principes 20
1.3.2 Exemple : parcourir un ensemble de points en 3 dimensions . 20
1.3.3 Exemple : la multiplication de matrices 22

Dans ce chapitre, nous présentons le fonctionnement des caches. Nous montrons quelle est l'impact des accès mémoire sur le temps d'exécution d'une application et étudions les différentes manières de réduire cet impact. Nous montrons en particulier comment mieux utiliser les caches à travers deux principes simples que nous illustrons sur des exemples.

1.1 Les caches

Nous commençons par présenter rapidement le fonctionnement des caches. On pourra consulter [HP06, Dre07] pour plus d'informations.

1.1.1 Définition

Un cache est une petite zone de mémoire placée entre le processeur et la mémoire centrale (RAM) pour accélérer les accès d'un programme à ces données. La capacité du cache est plus réduite que celle de la mémoire centrale mais le temps d'accès est plus petit. En général, on retrouve plusieurs caches entre le processeur et la mémoire, avec

Mémoire	Capacité	Latence	Associativité
cache de niveau 1 (L1)	32 Ko	4 cycles	8
cache de niveau 2 (L2)	256 Ko	10 cycles	8
cache de niveau 3 (L3)	8 Mo	40 cycles	16
RAM	8 Go	65 cycles	NA

Table 1.1 – Caractéristiques des caches du processeur Intel Nehalem (Xeon 5530) [MHSM09]

une capacité et une latence croissantes. Par exemple dans le cas du processeur Intel Nehalem, il y a 3 niveaux de caches dont les capacités et latences sont reportées en table 1.1.

1.1.2 La localité des accès mémoire

Les accès mémoire des programmes ont souvent certaines propriétés qui expliquent l'efficacité des caches, c'est la localité des accès mémoire. Une donnée accédée récemment a de bonnes chances d'être accédée à nouveau dans un futur proche : c'est la localité temporelle. Deux données ayant des adresses proches sont souvent accédées l'une après l'autre : c'est la localité spatiale. Ce sont ces deux propriétés qui assurent l'efficacité des caches. Si un programme ne les respecte pas, il ne bénéficiera pas de la présence de caches.

1.1.3 Fonctionnement d'un cache

Pour réduire la latence des accès au cache grâce à la localité spatiale, les transferts entre les différents niveaux de cache et la RAM ne sont pas atomiques mais gérés par bloc de plusieurs octets : une ligne de cache. Sur le processeur Intel Nehalem, la taille des lignes de cache est de 64 octets.

Pour des raisons d'efficacité, il n'est pas toujours possible de charger une ligne à n'importe quel endroit du cache. Lorsque c'est le cas, on dit que la cache est pleinement associatif (*fully associative*). Si une ligne de cache ne peut être chargée qu'à une seule position, on dit que le cache est direct (*direct mapped*). Dans la plupart des processeurs actuels, un compromis est utilisé. Le cache est divisé en plusieurs ensembles. Chaque ensemble est pleinement associatif mais une ligne de cache ne peut être chargée que dans un seul ensemble. Ces caches sont appelés *set associative*. La taille d'un ensemble est l'associativité du cache.

Comme un cache a une capacité limitée, toutes les données du programme ne peuvent être chargées dans le cache au même moment. Il faut donc décider quelle ligne de cache doit être supprimée lorsqu'une nouvelle ligne doit être chargée. L'algorithme prenant cette décision est appelé algorithme de remplacement. Un algorithme très souvent utilisé est la politique LRU pour Least Recently Used. On supprime la ligne de cache dont l'accès est le moins récent. LRU étant coûteux en pratique, une approximation est utilisée dans les processeurs actuels.

Classiquement la gestion des caches est effectuée par le processeur. Toutes les données

accédées [1] sont placées dans le cache et le programmeur n'a pas de contrôle direct sur l'algorithme de remplacement utilisé. Dans certain cas, par exemple pour la mémoire partagée (*shared memory*) de certains GPU Nvidia, le contrôle du cache est laissé au programmeur qui peut décider quelles données seront cachées et quel algorithme de remplacement utiliser.

1.1.4 Classification des défauts de cache

Lorsqu'une donnée accédée n'est pas dans le cache, elle doit y être chargée, ce qui retarde l'exécution du programme. On dit qu'accéder cette donnée cause un défaut de cache. On peut classer les défauts de cache en 3 catégories suivant le modèle 3C[HS89]. On qualifie un défaut de cache de :

Compulsory (obligatoire) si c'est le premier accès à cette donnée. Ce type de défauts de cache est inévitable car la donnée ne peut résider dans le cache.

Capacity (capacité) si on accède à une donnée chargée dans le cache précédemment mais qui n'y réside plus car elle a été expulsée par l'algorithme de remplacement. Ce type de défaut de cache peut être réduit en utilisant un cache de capacité supérieure.

Conflict (conflit) si la ligne de cache a été évincée car trop de lignes de caches sont chargées dans un même ensemble. Ce type de défaut de cache aurait pu être évité en utilisant un cache avec une associativité plus grande ou un cache pleinement associatif.

1.2 Caractérisation des applications

Dans cette partie, nous montrons quel est l'impact des accès mémoire sur le temps d'exécution d'une application. Nous en déduisons des caractéristiques qui déterminent la manière d'améliorer l'application afin de réduire le temps des accès mémoire.

1.2.1 Applications limitées par les accès mémoire

Pour un processeur avec un seul niveau de cache [2], on peut évaluer le temps d'exécution d'un programme T_{exe} avec la formule suivante [MPS02] :

$$T_{exe} = N \cdot [(1 - F_{memory}) \cdot T_{proc} + F_{memory} \cdot T_{memory}] \quad (1.1)$$

$$T_{memory} = G_{hit} \cdot T_{cache} + (1 - G_{hit}) \cdot T_{RAM} \quad (1.2)$$

avec

N le nombre d'instructions,

T_{proc} le durée moyenne d'exécution d'une instruction,

F_{memory} la proportion des instructions qui accèdent à la mémoire,

1. Des instructions sont disponibles dans le jeu d'instruction SSE pour spécifier qu'une donnée écrite ne doit pas être placée dans le cache.

2. Cette formule s'étend facilement au cas de plusieurs niveaux de cache.

T_{memory} le temps moyen pour accéder à la mémoire,

G_{hit} la proportion des accès mémoire trouvées dans le cache,

T_{cache} la latence du cache et

T_{RAM} la latence de la mémoire centrale.

Le premier terme de l'équation (1.1) correspond au temps passé à exécuter des calculs alors que le second terme correspond au temps passé à accéder à la mémoire. En général un processeur calcule plus rapidement qu'il n'accède à sa mémoire ($T_{proc} \leq T_{memory}$).

On dira qu'une application est limitée par le calcul lorsque F_{memory} est faible et donc le premier terme domine le second terme. En effet, diminuer T_{proc} en augmentant par exemple la fréquence du processeur, réduira significativement le temps d'exécution de l'application. A l'inverse une application est limitée par les accès mémoire lorsque F_{memory} est grand et donc le second terme domine le premier. Il faudra diminuer T_{memory} pour accélérer l'application. Pour cela, on peut modifier le programme pour améliorer la localité des accès mémoire ce qui rend le cache plus efficace : G_{hit} augmente. Les sections 1.3 et 2.3.1 présentent des principes et techniques pour améliorer cette localité. Une autre possibilité analysée dans les deux sous-sections suivantes est de réduire l'influence de T_{RAM} en recouvrant la latence des accès mémoire par des calculs.

Remarque. L'équation (1.1) ne s'applique pas dans le cas des processeurs *out-of-order* et *superscalar*. Cela concerne certains processeurs actuels. Dans cette configuration, le processeur peut exécuter des opérations en attendant la fin d'un accès mémoire si ces opérations sont indépendantes de l'accès mémoire. Il y a alors recouvrement entre les calculs T_{proc} et les accès mémoire T_{mem} ce qui diminue le coût apparent des accès mémoire. Cependant dans le cas des applications limitées par les accès mémoire cet effet est relativement faible.

Remarque. On peut évaluer F_{memory} en mesurant le nombre d'instructions accédant à la mémoire lors de l'exécution du programme grâce aux compteurs de performance. La bibliothèque PAPI[BDG+00] permet d'accéder facilement à ces compteurs.

1.2.2 Prédictibilité des accès mémoire et *prefetching*

Dans la plupart des processeurs actuels, il existe un mécanisme appelé *prefetching* qui permet de charger dans le cache une donnée dont on aura besoin dans le futur. Le *prefetching* réduit la latence des accès à la mémoire centrale T_{RAM} car le transfert de la ligne de cache est lancé pendant que le processeur exécute les instructions de calculs précédant l'accès mémoire.

Le *prefetching* peut être réalisé automatiquement par le processeur lorsque celui-ci détecte un schéma d'accès mémoire régulier, par exemple un accès linéaire lors du parcours d'un tableau. Lorsque les accès mémoire ne sont pas suffisamment réguliers pour être prévus par le processeur mais sont prévisibles par le programmeur, le *prefetching* peut être réalisé de manière logicielle en faisant appel à certaines instructions spécialisées. Quand le *prefetching* matériel est possible il est préférable au *prefetching* logiciel car ce dernier engendre un coût en temps pour exécuter les instructions de *prefetching*. Quand le *prefetching* peut être réalisé, on dit que l'application a des accès mémoire prédictibles.

Parfois, ni le processeur ni le programmeur ne peuvent prévoir l'accès mémoire suivant. Par exemple, lors du parcours d'une liste chaînée, on doit charger l'accès mémoire précédant dans le cache avant de pouvoir lancer l'accès mémoire suivant.

1.2.3 Applications limitées par la bande passante ou la latence

Pour une application limitée par les accès mémoire, trois cas sont possibles après utilisation du *prefetching*.

1. Les accès mémoire sont prédictibles et le *prefetching* réduit très fortement T_{RAM}. L'application n'est plus limitée par les accès mémoire.
2. Les accès mémoire ne sont pas prédictibles et le *prefetching* est impossible. On dit que l'application est limitée par la latence.
3. Les accès mémoire sont prédictibles mais il n'y a pas assez de calculs pour recouvrir la latence de tous les accès mémoire. On dit que l'application est limitée par la bande passante.

Une fois que le *prefetching* a été considéré, si l'application est toujours limitée par ses accès mémoire (par la latence ou par la bande passante), il faut améliorer G_{hit} en augmentant la localité des accès mémoire.

Remarque. Le SMT (*simultaneous multithreading* ou *hyperthreading*) est une autre technique pour réduire l'effet de la latence des accès mémoire même dans le cas où les accès mémoire ne sont pas prédictibles. L'application doit être programmée en utilisant plusieurs threads. Lorsque l'un des threads est bloqué sur l'attente d'un accès mémoire, on exécute un autre thread. Cette technique est utilisée par les GPUs et certains processeurs.

1.2.4 Réutilisation et localité temporelle

Dans une application, on définit le taux de réutilisation des données par le nombre moyen de fois où une donnée est accédée au cours de l'exécution. Faire le produit scalaire de deux vecteurs a un taux de réutilisation de 1 car chaque donnée (une composante du vecteur) n'est utilisée qu'une fois. A l'inverse, un produit $C = A \cdot B$ de deux matrices n par n a une réutilisation de n car chaque coefficient des matrices A et B est utilisé n fois, une fois par coefficient de la matrice C sur la même ligne ou la même colonne. Lorsqu'une application a une réutilisation de 1, on ne peut pas tirer parti de la localité temporelle. S'il y a de la réutilisation, on peut évaluer la localité temporelle avec la distance de réutilisation [GST70].

On considère pour cela un espace mémoire divisé en segments de la taille d'une ligne de cache. Soit la suite $(a_i)_{i \leq 0}$ des références à des segments mémoire d'une application. Lors de l'accès a_i à un nouveau segment $\forall k < i, a_k \neq a_i$, la distance de réutilisation est infinie : $d(a_i) = +\infty$. Lors de l'accès suivant au même segment $a_j = a_i$ et $\forall i < k < j, a_k \neq a_i$, la distance de réutilisation est le nombre d'accès différents entre ces deux accès : $d(a_j) = \text{Card}\{a_k \mid i < k < j\}$. Dans le cas d'un cache pleinement associative géré par la politique de remplacement LRU, l'histogramme des distances de réutilisation permet d'évaluer le nombre de défauts de cache de l'application. On note

h_d le nombre d'accès dont la distance de réutilisation est d. Le nombre de défaut de caches pour un cache contenant C ligne de caches est

$$Q(C) = \sum_{d=C}^{+\inf} h_d.$$

En effet, les accès ayant une distance de réutilisation infinie correspondent à des défauts de cache obligatoire (*compulsory*) et les accès ayant une distance de réutilisation supérieure à C sont des défauts de cache de capacité (*capacity*). Lors du $C^{\text{ième}}$ accès à un nouveau segment, la ligne correspondante est évincée du cache car c'est celle dont le dernier accès est le moins récent. Ce modèle ne tient pas compte des défauts de cache de conflit (*conflict*) mais reste une bonne approximation [BD01] dans le cas des caches partiellement associatifs.

Améliorer la localité temporelle d'une application, revient à réduire les distances de réutilisation. Lorsque toutes les distances de réutilisation finies sont inférieures à C, le nombre de défauts de cache est minimal car les seuls défauts de cache sont les défauts de cache de type obligatoires.

1.3 Les principes pour concevoir une application efficace en cache

Dans cette partie, nous énonçons trois principes qu'une application doit respecter pour optimiser l'utilisation des caches. Nous illustrons ensuite notre propos par deux exemples.

1.3.1 Les 3 principes

Les deux premiers principes sont à rapprocher aux propriétés de localité présentées dans la partie 1.1.2.

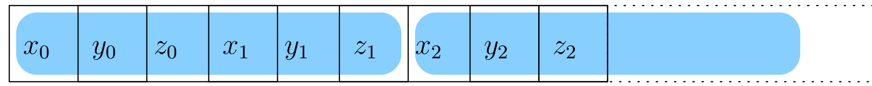
Localité Spatiale Une ligne de cache doit contenir autant de données utiles que possible.

Localité Temporelle Lorsqu'une donnée est dans le cache, elle doit être réutilisée le plus tôt possible.

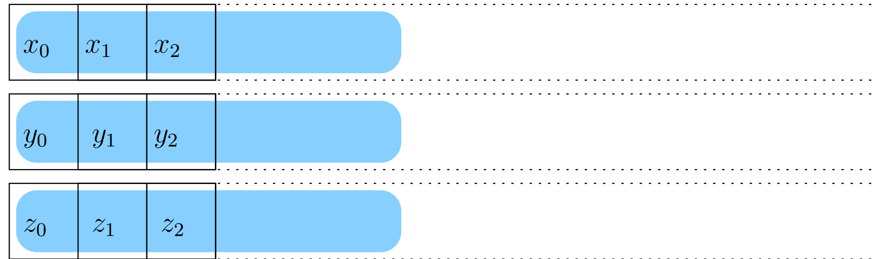
Efficacité Appliquer les deux principes précédents doit augmenter le moins possible le nombre d'opérations.

1.3.2 Exemple : parcourir un ensemble de points en 3 dimensions

Dans cet exemple, on travaille sur un ensemble n de points en 3 dimensions. On suppose que l'ensemble de ces points est bien plus grand que la taille du cache (`n*sizeof(double)>C`). Il existe deux façons de stocker ces points en mémoire (*cf.* figure 1.1). La première méthode, que l'on appelle *Array of Structs* ou AoS, consiste à stocker les coordonnées de chaque point de manière contigüe dans une structure puis de stocker l'ensemble de ces structures linéairement en mémoire dans un tableau. La



(a) Organisation mémoire des données AoS



(b) Organisation mémoire des données SoA

```
// donnees AoS
struct point {
  double x,y,z ;
} ;
point mesh[n] ;


// calcul par coordonnees
for ( int i=0; i<n; ++i )
  mesh[i].x -= x_A ;
for ( int i=0; i<n; ++i )
  mesh[i].y -= y_A ;
for ( int i=0; i<n; ++i )
  mesh[i].z -= z_A ;

// calcul par points
for ( int i=0; i<n; ++i ){
  mesh[i].x -= x_A ;
  mesh[i].y -= y_A ;
  mesh[i].z -= z_A ;
}
```

(c) Code pour données AoS

```
// donnees SoA
struct mesh {
  double x[n] ;
  double y[n] ;
  double z[n] ;
}

// calcul par coordonnees
for ( int i=0; i<n; ++i )
  mesh.x[i] -= x_A ;
for ( int i=0; i<n; ++i )
  mesh.y[i] -= y_A ;
for ( int i=0; i<n; ++i )
  mesh.z[i] -= z_A ;

// calcul par points
for ( int i=0; i<n; ++i ){
  mesh.x[i] -= x_A ;
  mesh.y[i] -= y_A ;
  mesh.z[i] -= z_A ;
}
```

(d) Code pour données SoA

FIGURE 1.1 – Comparaison entre les données AoS et SoA.

deuxième méthode, *Struct of Arrays* ou SoA, stocke les coordonnées dans 3 tableaux différents.

Supposons que l'on réalise un changement de repère pour l'ensemble de ces points dans un repère centré au point A de coordonnées (x_A, y_A, z_A). Il existe deux méthodes pour mettre à jour les coordonnées. On peut procéder coordonnée par coordonnée ou point par point. La façon dont on stocke les données et réalise le calcul influence fortement le nombre de défauts de cache.

Ici, les données ne sont jamais réutilisées : une coordonnée d'un point n'est utilisée qu'une fois lors de sa mise à jour. On prête donc attention à la localité spatiale. Si on réalise le calcul par coordonnées avec un stockage AoS, on ne tire pas bien parti de la localité spatiale. En effet, lors du traitement d'une coordonnée, on charge dans le cache les 3 coordonnées de chaque point car elles sont dans la même ligne de cache mais une seule est utilisée. Si on réalise le calcul par points, chaque coordonnée est utilisée immédiatement après son chargement dans le cache. Le calcul par coordonnées cause 3 fois plus de défauts de cache que le calcul par points. Dans le cas d'un stockage SoA, le calcul par coordonnées tire pleinement parti de la localité spatiale. Le calcul par points ne cause pas plus de défauts de cache si le cache peut contenir au moins 3 lignes de cache et qu'il n'y a pas de défauts de cache de conflits.

Dans tous les cas, cet exemple a des accès mémoire prédictibles par le processeur et le *prefetching* matériel rend le problème limité par la bande passante mémoire. Profiter pleinement de la localité spatiale permet de réduire le besoin en bande passante d'un facteur 3 et donc diminuer le temps d'exécution d'environ autant.

Remarque. En règle générale, on utilise le stockage AoS lorsque toutes les données de la structure sont utilisées à chaque opération sur la structure. Dans le cas inverse il vaut mieux utiliser le stockage SoA.

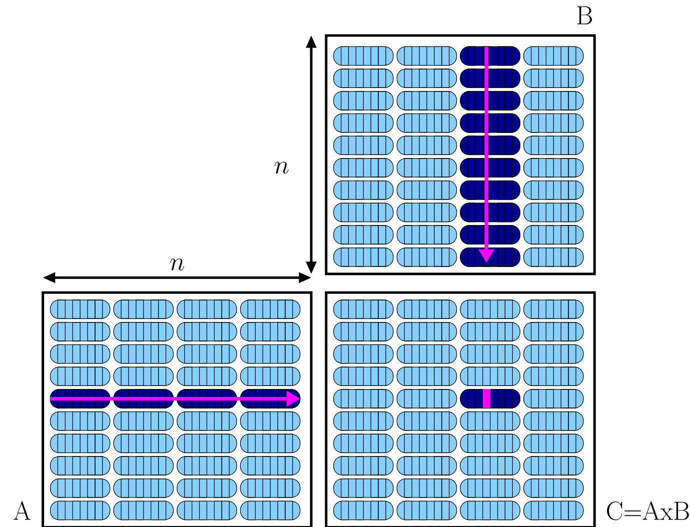
1.3.3 Exemple : la multiplication de matrices

On étudie maintenant la multiplication $C = A \times B$ de deux matrices n par n par un algorithme classique en $O(n^3)$. On suppose qu'une ligne de la matrice ne tient pas dans le cache.

Avec l'implémentation naïve en 3 boucles for, la localité spatiale est bonne pour les accès à la matrice A mais mauvaise pour les accès à la matrice B (*cf.* figure 1.2). En effet, chaque lecture d'un coefficient de B charge une ligne de cache entière contenant plusieurs coefficients mais un seul est utilisé.

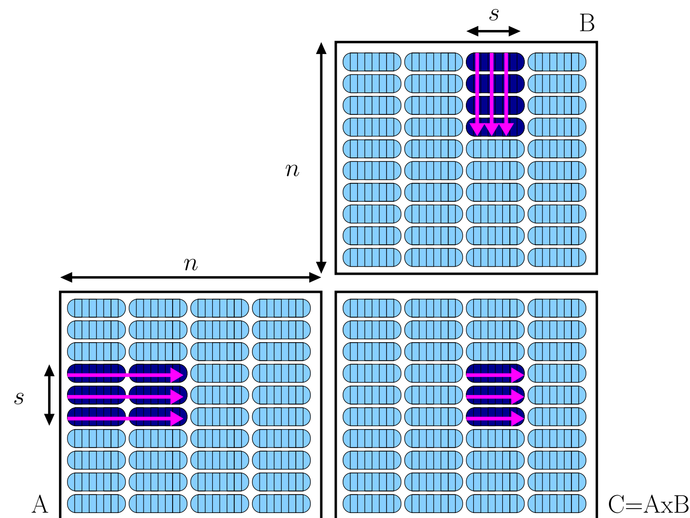
La réutilisation est de n, on peut donc examiner la localité temporelle. Lors du calcul de $C[i, j]$ on utilise la ligne $A[i, :]$ et la colonne $B[:, j]$. A cause de la taille limitée du cache, la ligne $A[i, :]$ doit être rechargée entièrement dans le cache lors du calcul de $C[i, j+1]$. De même pour la colonne $B[:, j]$ lors du calcul de $C[i+1, j]$.

Pour améliorer la localité temporelle, on peut réorganiser le calcul par blocs de taille s par s (*cf.* figure 1.2). On utilise un algorithme de multiplication de matrices par blocs ayant la même complexité en nombre d'opérations. Par contre, si la taille des blocs est choisie de manière à ce que 3 blocs rentrent dans le cache, c'est-à-dire $3s^2 \leq C$, le nombre de défauts de cache est fortement diminué. En effet, la distance de réutilisation aux lignes de A et colonnes de B est réduite ce qui améliore la localité temporelle. Lors de la multiplication de deux blocs, on peut maintenant réutiliser la partie de la




```
for ( int i=0; i<n; ++i )
  for ( int j=0; j<n; ++j )
    for ( int k=0; k<n; ++k )
      C[i,j] += A[i,k] * B[k,j]
```

(a) Algorithme de multiplication de matrice standard




```
for ( int ii=0; ii<n/s; ++ii )
 for ( int jj=0; jj<n/s; ++jj )
  for ( int kk=0; kk<n/s; ++kk )
   for ( int i=ii*s; i<(ii+1)*s; ++i )
    for ( int j=jj*s; j<(jj+1)*s; ++j )
     for ( int k=kk*s; k<(kk+1)*s; ++k )
      C[i,j] += A[i,k] * B[k,j]
```

(b) Algorithme de multiplication de matrice par blocs

FIGURE 1.2 – Comparaison de deux algorithmes de multiplication de matrices. La multiplication de matrices par bloc cause moins de défauts de cache grâce à des distances de réutilisation plus faibles.

ligne $A[i,:]$ chargée lors du calcul partiel de $C[i,j]$ pour effectuer le calcul partiel de $C[i,j+1]$. De même pour les colonnes de B lors du calcul de $C[i+1,j]$. Cette méthode présente encore un avantage. Lors du calcul partiel de $C[i,j+1]$, on peut utiliser la colonne de $B[:,j+1]$ qui avait été chargée dans le cache lors du calcul de $C[i,j]$ car elle se trouve dans les mêmes lignes de cache que la colonne $B[:,j]$.

On analysera dans la partie 2.2.2 le nombre de défauts de cache causé par la multiplication de matrices par blocs.

Algorithmes utilisant efficacement les caches 2

Sommaire

2.1 Analyser les défauts de cache d'un algorithme 25
2.1.1 Le modèle *cache-aware* 26
2.1.2 Le modèle *cache-oblivious* 26
2.1.3 Limites du modèle *cache-oblivious* 27
2.2 Analyse de quelques algorithmes classiques 27
2.2.1 Parcours d'un tableau 28
2.2.2 Multiplication de matrices 28
2.2.3 Recherche d'éléments dans un arbre 30
2.3 Conception d'algorithmes efficaces en cache 33
2.3.1 Techniques algorithmiques CA et CO 33
2.3.2 Performances en pratique des techniques CA et CO 35
2.4 Algorithmes CA et CO : revue des résultats 35
2.4.1 Revue des algorithmes CA et CO 36
2.4.2 Comparaison des algorithmes CA et CO 37

Dans ce chapitre, nous donnons des techniques algorithmiques pour améliorer la localité de accès mémoires. Nous passons en revue les modèles *cache-aware* et les modèles *cache-oblivous* et montrons comment les utiliser pour analyser les défauts de cache de trois algorithmes classiques. Nous décrivons ensuite des techniques génériques pour concevoir des algorithmes efficaces dans ces modèles. Enfin, nous donnons quelques résultats théoriques et pratiques en se concentrant sur la comparaison entre les algorithmes classiques, les algorithmes *cache-aware* et les algorithmes *cache-oblivious*.

2.1 Analyser les défauts de cache d'un algorithme

Concevoir des algorithmes utilisant efficacement les caches nécessite de disposer de modèles simplifiés de fonctionnement des caches. C'est grâce à ces modèles que l'on pourra analyser les performances algorithmes. Nous présentons deux de ces modèles dans les sections suivantes.

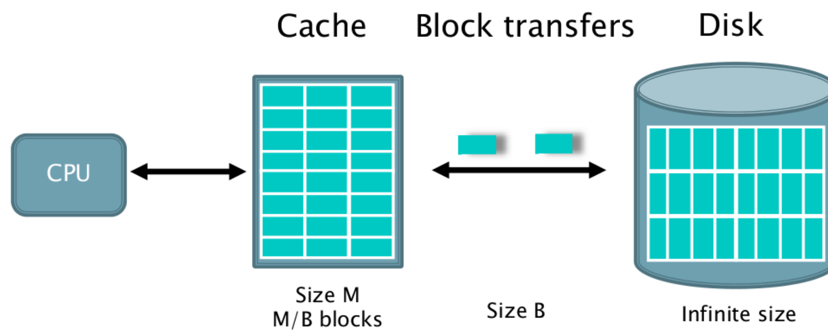


FIGURE 2.1 – Modèle *cache-aware*

2.1.1 Le modèle *cache-aware*

Ce modèle porte plusieurs noms suivant les parties de la hiérarchie mémoire qu'il modélise : la mémoire centrale et le disque ou le cache et la mémoire centrale. Dans le premier cas, on parle du modèle à mémoire externe ou du modèle I/O [AV88], dans le deuxième cas, du modèle *cache-aware* (CA). Dans ce modèle, il y a deux niveaux de mémoire : le cache et la mémoire centrale. Les données transitent entre ces deux niveaux par blocs de taille B. Le cache a une taille M et peut donc contenir M/B blocs. L'algorithme a le contrôle du cache : lorsque le cache est plein, l'algorithme choisit quel bloc doit être évincé. De nombreux algorithmes ont été étudiés dans ce modèle. On peut consulter [Vit01]. On analysera quelques algorithmes dans la section 2.2.

2.1.2 Le modèle *cache-oblivious*

Le modèle *cache-oblivious* (CO) [FLPR99] est similaire au modèle CA. On utilise le modèle CA mais on interdit à l'algorithme d'utiliser les valeurs de B et M : la taille des blocs et la taille du cache (voir figure 2.2). Le modèle CO à deux avantages majeurs par rapport au modèle CA.

Multi niveaux. Comme l'algorithme n'utilise pas la taille des blocs et du cache, les niveaux cache et mémoire centrale peuvent correspondre à n'importe quel couple de niveaux : cache L_1 et cache L_2, cache L_2 et cache L_3, cache L_3 et mémoire centrale, etc. Un algorithme efficace dans le modèle CO est efficace sur toute la hiérarchie mémoire [FLPR99].

Portabilité. Comme l'algorithme n'utilise pas la taille des blocs et du cache, il est efficace quelque soient la taille des caches et des lignes de cache de la machine.

Le modèle CO comporte une autre différence avec le modèle CA. Le cache n'est plus contrôlé par l'algorithme comme dans le modèle CA mais est supposé idéal. Il est pleinement associatif et est géré par la politique de remplacement optimale : *Furthest in Future* (FIF). La ligne de cache évincée est celle qui sera accédée le plus tard possible. Bien sûr, cette politique de remplacement est impossible à mettre en place en pratique car elle nécessite la connaissance des accès futurs. Le papier présentant le modèle CO [FLPR99] contient des justifications théoriques pour ces deux suppositions (FIF et associativité) qu'on peut résumer ainsi : un algorithme qui cause $Q(B, M)$ défauts de cache dans le modèle CO peut être simulé sur un cache LRU pleinement associatif avec

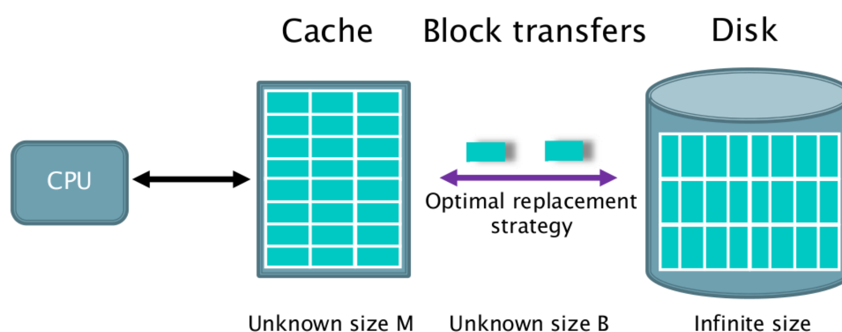


FIGURE 2.2 – Modèle *cache-oblivious*

$O(Q(B, M))$ défauts de cache si il respecte la condition de régularité suivante :

$$Q(B, M) = O(Q(B, 2M)) \tag{2.1}$$

De plus un cache LRU pleinement associatif peut être implémenté dans le modèle CA sans perte de performance asymptotique.

De nombreux algorithmes ont été étudiés dans le modèle CO [ABF05]. Nous analyserons quelques algorithmes dans la section 2.2.

2.1.3 Limites du modèle *cache-oblivious*

Le modèle CO est simple à utiliser et donne des analyses qui reflètent souvent les performances en pratique. Il comporte néanmoins quelques limites.

Le modèle CO ne prend pas en compte le *prefetching*. Dans certains cas, un algorithme A_1 causant moins de défauts de cache qu'un algorithme A_2 dans le modèle CO sera moins rapide en pratique si les accès mémoire de A_2 sont prédictibles. Par exemple, l'algorithme de Floyd-Warshall qui n'est pas optimal en terme de nombre de défauts de cache est plus rapide en pratique que son équivalent optimal dans le modèle CO [PCDL07]. Il est possible d'étendre le modèle CA pour prendre en compte le *prefetching* [VS07] mais il n'existe pas d'approche similaire pour le modèle CO.

Le modèle CO suppose que la politique de remplacement est idéale. Dans certains cas, il y a une différence notable entre le nombre de défauts de cache en pratique et le nombre de défauts de cache prédits par le modèle CO. Par exemple, lorsqu'un algorithme mêle deux types d'accès : un accès de type *streaming* et un accès avec de la réutilisation. Dans le modèle CO, l'accès *streaming* utilisera seulement une ligne du cache laissant l'espace restant pour l'accès avec de la réutilisation. En pratique, avec une politique LRU, l'accès *streaming* utilisera beaucoup d'espace dans le cache, limitant la réutilisation pour l'autre accès.

2.2 Analyse de quelques algorithmes classiques

Dans cette partie, nous analysons trois problèmes classiques dans les modèles CA et CO.

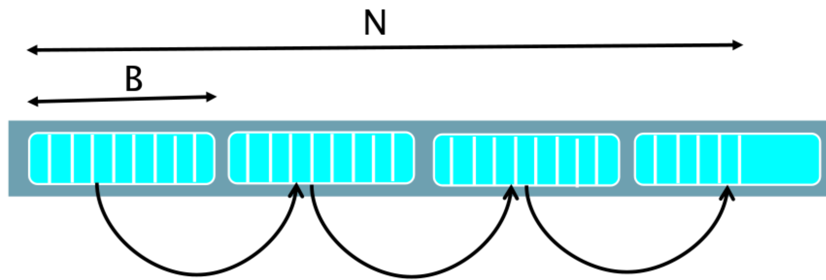


FIGURE 2.3 – Parcours d'un tableau

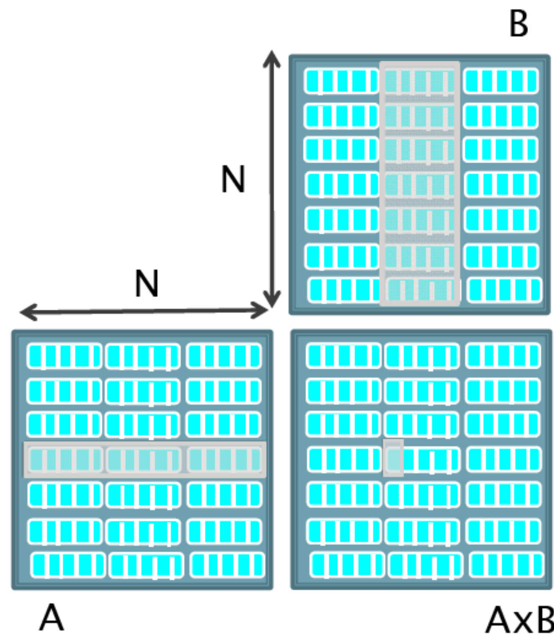


FIGURE 2.4 – Multiplication de matrices avec l'algorithme classique

2.2.1 Parcours d'un tableau

Accéder linéairement aux données est une technique efficace dans les modèles CA et CO. On considère l'exemple du parcours d'un tableau de taille N. L'algorithme naïf est optimal car il utilise parfaitement la localité spatiale. Le nombre de défauts de cache est $\left\lceil \frac{N}{B} \right\rceil$ (voir figure 2.3).

2.2.2 Multiplication de matrices

Nous avons vu dans la section 1.3.3 que l'algorithme de multiplication de matrices classique en $O(n^3)$ cause beaucoup de défauts de cache. Nous allons évaluer ce nombre de défauts de cache dans le modèle CO et proposer deux algorithmes asymptotiquement optimaux : l'un dans le modèle CA et l'autre dans le modèle CO.

Pour des raisons de simplicité, on se limite aux cas des matrices carrés N par N. On analyse le produit $C = A \times B$ par l'algorithme classique. Pour chaque élément $C[i, j]$

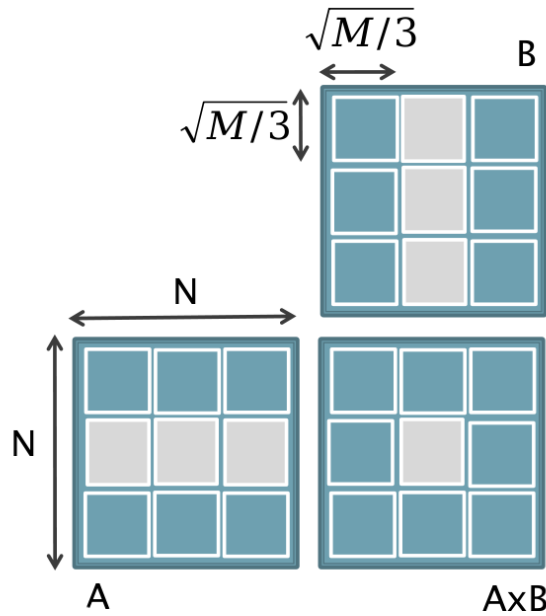


FIGURE 2.5 – Multiplication de matrices par blocs

de la matrice C, on doit charger dans le cache la ligne $A[i,:]$ ce qui cause $\frac{N}{B}$ défauts de cache et la colonne $B[:,j]$ ce qui cause N défauts de cache (voir figure 2.4). Si une ligne ou une colonne est plus grande que la taille du cache, c'est-à-dire $N > M$, on ne peut réutiliser les données entre les calculs des éléments de la matrice C. Les accès à la matrice C sont linéaires, ils causent $\frac{N^2}{B}$ défauts de cache. On a donc au total :

$$Q(N) = \frac{N^2}{B} + N^2 \cdot \left(\frac{N}{B} + N\right) = O(N^3)$$

défauts de cache.

On propose maintenant de calculer le produit de matrices par blocs de taille k par k, avec $k \geq B$ (voir figure 2.5). Si 3 blocs tiennent simultanément dans le cache, c'est-à-dire $3k^2 \leq M$, on peut conserver à tout instant un bloc de la matrice A, un bloc de la matrice B et un bloc de la matrice C dans le cache. Multiplier deux blocs ne cause pas de défauts de cache en dehors de la lecture des blocs de A et B et de l'écriture du résultat dans C. Pour chaque bloc dans la matrice C, on doit lire $\frac{N}{k}$ blocs dans la matrice A et dans la matrice B. La lecture d'un bloc cause $\frac{k^2}{B}$ défauts de cache. On a donc au total :

$$\left(\frac{N}{k}\right)^2 \cdot \left(\frac{N}{k} \cdot 2 \cdot \frac{k^2}{B} + \frac{k^2}{B}\right) = O\left(\frac{N^3}{k \cdot B}\right)$$

On veut k le plus grand possible sachant que $3k^2 \leq M$. On prend donc $k = \sqrt{\frac{M}{3}}$, ce qui donne

$$Q(N) = O\left(\frac{N^3}{B \cdot \sqrt{M}}\right) \tag{2.2}$$

défauts de cache. Ce résultat est asymptotiquement optimal [HK81].

L'algorithme précédent a besoin de connaître la taille du cache M. Nous montrons que l'on peut atteindre la même borne sans utiliser cette valeur et obtenir ainsi un

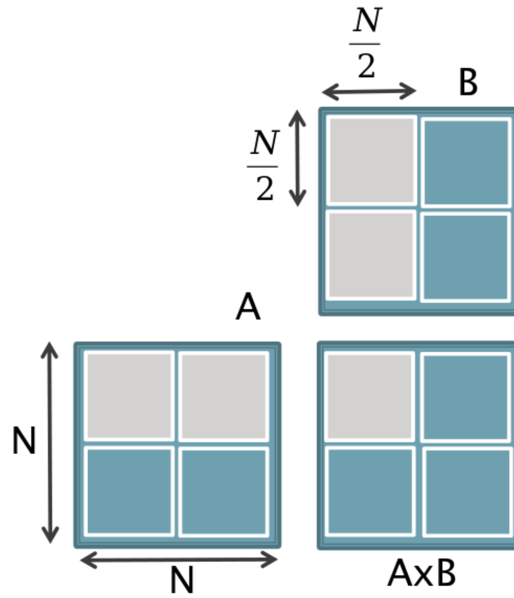


FIGURE 2.6 – Multiplication de matrices diviser pour régner

algorithme CO. Pour cela on utilise l'algorithme de multiplication de matrices diviser pour régner. On décompose récursivement le produit en 8 produits de matrices $\frac{N}{2}$ par $\frac{N}{2}$. La complexité en nombre d'instructions est

$$W(1) = O(1)$$
$$W(N) = 8W\left(\frac{N}{2}\right) + O(N^2)$$

ce qui donne $W(N) = O(N^3)$. On calcule de la même manière le nombre de défauts de cache. L'addition des sous-produits accède aux données linéairement et cause $O(\frac{N^2}{B})$ défauts de cache. De plus, lorsque $3N^2 \leq M$, les 3 sous-matrices tiennent dans le cache donc les appels récursifs ne génèrent plus de défauts de cache. On a

$$Q(N) = 8Q\left(\frac{N}{2}\right) + O\left(\frac{N^2}{B}\right) \text{ si } 3N^2 > M$$
$$Q(N) = \frac{N^2}{B} \text{ si } 3N^3 \leq M$$

ce qui donne

$$Q(N) = O\left(\frac{N^3}{B \cdot \sqrt{M}}\right) \tag{2.3}$$

Cet algorithme atteint asymptotiquement la même performance que l'algorithme précédent sans qu'il ne soit nécessaire de de connaître la taille du cache.

2.2.3 Recherche d'éléments dans un arbre

Nous considérons maintenant la recherche dans un arbre binaire équilibré contenant N éléments et de hauteur $O(\log_2 N)$.

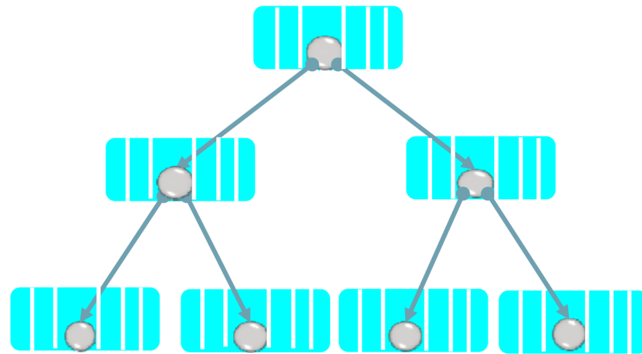


FIGURE 2.7 – Recherche dans un arbre binaire

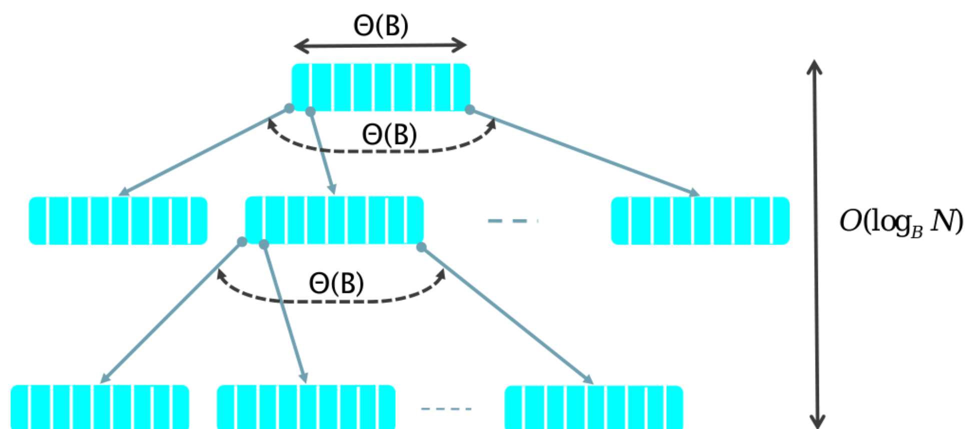


FIGURE 2.8 – Recherche d'éléments dans un B-arbre

Si on stocke chaque nœud de l'arbre séparément par exemple avec un appel à malloc, il est possible que chaque ligne de cache ne contiennent qu'un seul nœud de l'arbre (voir figure 2.7). Une autre manière classique de stocker un arbre binaire est d'utiliser un tableau T en stockant la racine dans la case T[0] et les fils gauche et droit du nœud stocké en T[i] dans les cases T[2∗i+1] et T[2∗i+2]. Dès que l'on s'éloigne de la racine, une ligne de cache ne contient que des nœuds qui ne sont pas sur le même chemin racine-feuille. On a donc besoin dans tous ces cas, d'un défaut de cache par nœud de l'arbre pour un coût total de :

$$Q(N) = h = O(\log_2 N) \tag{2.4}$$

La recherche d'éléments peut être améliorée en utilisant les B-arbres [BM72, CLRS09]. Chaque nœud de l'arbre contient $\Theta(B)$ éléments de manière à occuper exactement une ligne de cache (voir figure 2.8). Chaque nœud est trié et a $\Theta(B)$ fils. L'arbre est équilibré avec une hauteur $O(\log_B N)$. On recherche un élément en faisant une recherche binaire dans un nœud puis récursivement sur l'un des fils. La complexité est de $W(N) = O(\log_2 B) \cdot h = O(\log_2 N)$ identique à celle de l'arbre binaire. En revanche la localité spatiale est améliorée car chaque élément d'une ligne de cache est

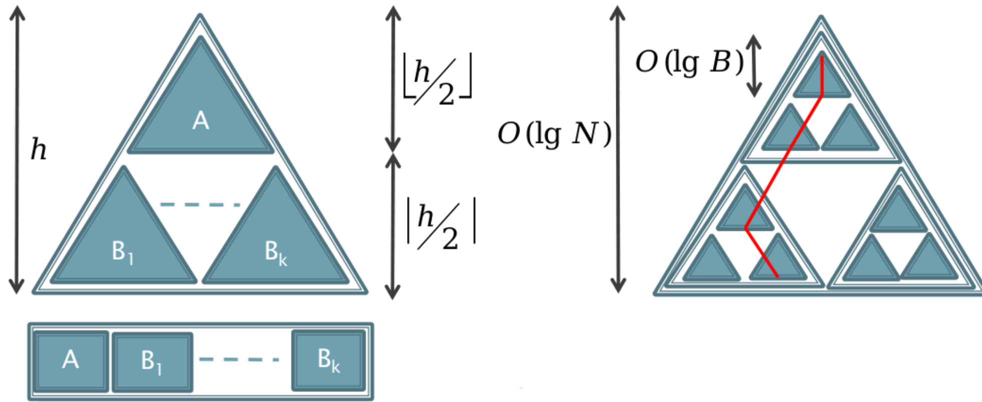


Figure 2.9 – Organisation mémoire CO d'un arbre binaire

utile. Le nombre de défauts de cache est de

$$Q(N) = 1 \cdot h = O(\log_B N) \tag{2.5}$$

soit un gain de $O(\log_2 B)$. Le B-arbre utilise la taille des lignes de cache, il est donc CA.

On peut obtenir des performances similaires au B-arbre dans le modèle CO en organisant avec précaution le stockage de l'arbre binaire. Pour simplifier la présentation on considère le cas d'un arbre binaire complet, pour plus de détails voir [BDFC05]. On va stocker l'arbre binaire dans un tableau T en utilisant une organisation récursive appelée van Emde Boas (voir figure 2.9). On découpe l'arbre au milieu de sa hauteur en un sous-arbre père A de hauteur $\frac{h}{2}$ et k sous-arbres fils $B_1, \ldots, B_k$ de hauteur $\frac{h}{2}$. On découpe récursivement ces sous-arbres et on les place dans le tableau dans l'ordre $A, B_1, \ldots, B_k$. La complexité de la recherche est identique à celle de l'arbre binaire car seule l'organisation des données en mémoire a été modifiée. Pour analyser le nombre de défauts de cache, on considère le moment où la découpe récursive a généré des sous-arbres de hauteur inférieure à $\log_2 B$, ainsi chaque sous-arbre tient dans une ligne de cache. Lors de la recherche d'un élément, on parcourt un chemin racine feuille qui passe successivement par

$$\frac{h}{\log_2 B} = O\left(\frac{\log_2 N}{\log_2 B}\right) = O(\log_B N)$$

sous-arbres. Or chaque sous-arbre cause un défaut de cache. On a donc au total :

$$Q(N) = O(\log_B N) \tag{2.6}$$

soit asymptotiquement le même nombre de défauts de cache que pour le B-arbre mais sans utiliser la taille des lignes de cache.

2.3 Conception d'algorithmes efficaces en cache

2.3.1 Techniques algorithmiques CA et CO

Dans cette partie, nous présentons des techniques utiles pour concevoir des algorithmes CA et CO. Ces techniques se divisent en 2 catégories :
– la réorganisation des calculs : changer l'ordre dans lesquels les calculs sont effectués
– la réorganisation des données : changer la façon dont les données sont organisées en mémoire.

Accès séquentiel aux données

L'accès séquentiel est une manière très efficace d'accéder aux données car il permet une utilisation optimale de la localité spatiale. Cette technique ne nécessite pas la connaissance des tailles de cache ni de ligne de cache, on peut donc l'utiliser dans les deux modèles CA et CO. Elle est analysée dans la section 2.2.1.

Trier les données

Il existe plusieurs algorithmes de tri CA et CO efficaces en cache. On peut évidemment les utiliser pour remplacer leurs alternatives classiques (tri fusion ou tri rapide) si l'algorithme à rendre efficace comporte une étape de tri. On peut également utiliser un tri CA ou CO pour rendre efficace en cache un algorithme qui ne comportait pas d'étape de tri. Cette technique permet d'améliorer une étape comportant beaucoup d'accès mémoire aléatoires qui sont mauvais pour la localité. Par exemple dans le cas des algorithmes sur les graphes, on visite souvent l'ensemble des nœuds en utilisant les informations de connectivité ce qui génère des accès aléatoires aux nœuds. Si on peut obtenir efficacement l'ordre dans lequel on doit visiter les nœuds du graphe, on peut trier les nœuds dans cet ordre puis utiliser des accès séquentiels.

Calcul par blocs

La technique de calcul par blocs consiste à partitionner l'ensemble des opérations à effectuer en blocs, chaque bloc accédant à un même sous-ensemble des données. Cette technique permet notamment d'augmenter la localité temporelle en réduisant les distances de réutilisation. Cette technique est en général utilisée pour des algorithmes CA car on choisit la taille des blocs de manière à ce que la quantité de données accédées par un bloc soit inférieure à la taille du cache. Un exemple d'utilisation de cette technique est la multiplication de matrices par blocs présentée dans la section 2.2.2.

Diviser pour régner

Les algorithmes de type diviser pour régner sont une alternative CO à la technique CA du calcul par blocs. L'ensemble des calculs à effectuer est divisé récursivement en blocs de plus en plus petits. Il existe donc un niveau de la découpe récursive dans lequel les blocs sont plus petits que la taille du cache. Ainsi il n'est pas nécessaire de connaître la taille du cache pour utiliser cette technique. C'est la méthode utilisée par l'algorithme de multiplication de matrices CO de la section 2.2.2.

Organisation mémoire par blocs

L'organisation mémoire par blocs est analogue à la technique de calcul par blocs mais pour les données. On découpe la structure de données de manière à ce qu'une sous-partie contigüe des calculs accède à un même bloc de données. Cela permet entre autres d'augmenter la localité spatiale. Cette technique est plutôt utilisée dans les algorithmes CA car on choisit en général la taille des blocs en fonction de la taille des lignes de cache. Un exemple d'utilisation de cette méthode est le B-arbre présenté dans la section 2.2.3.

Organisation mémoire récursive

L'organisation mémoire récursive est l'alternative CO de l'organisation mémoire par blocs. Les données sont découpées récursivement en morceaux de plus en plus petits. Il existe un niveau de la découpe récursive dans lequel les blocs sont plus petits que la taille d'une ligne de cache. Il n'est donc pas nécessaire de connaître la taille des lignes de cache. C'est la méthode utilisée par le B-arbre CO de la section 2.2.3.

Space Filling Curves

L'utilisation des *space filling curves* est un cas particulier des techniques diviser pour régner pour les calculs et d'organisation mémoire récursive pour les données. On considère des découpes en blocs récursives spécifiques qui correspondent à des *space filling curves* comme par exemple la courbe de Lebesgue ou la courbe de Hilbert. En utilisant ces courbes, on peut parcourir ou stocker de manière locale des données multidimensionnelles. Ces courbes fournissent une meilleure localité qu'une découpe récursive classique. De plus il est possible de les utiliser de manière itérative ce qui peut améliorer les performances. Ces courbes ont été utilisées par exemple pour construire des algorithmes CO de multiplication de matrices [BZ05] ou des organisations mémoire de maillages réguliers [PF01].

Partitionnement ou regroupement

Ces techniques sont analogues à l'organisation mémoire par blocs pour les structures de données irrégulières à base de pointeurs. Par exemple les structures de données utilisées pour les matrices creuses, les graphes, les maillages. Contrairement au cas régulier où on doit généralement changer le code du calcul si on réorganise les données, ce n'est pas nécessaire pour le cas irrégulier. En effet, comme le code du calcul suit des pointeurs, il suffit de changer leur destination pour changer l'organisation des données. Le partitionnement et le regroupement ont pour but de stocker de manière contigüe les données qui sont accédées consécutivement par le calcul. On peut soit regrouper en mémoire des données utilisées en même temps par le calcul, soit partitionner les données en sous-ensembles qui sont utilisés à différents moments du calcul. Pour cela on utilise souvent des techniques de partitionnement de graphes avec une connectivité déduite de l'utilisation des données dans l'algorithme. Si on cherche à obtenir des partitions de taille fixée on obtient une méthode CA. L'utilisation de cette technique de manière récursive (hiérarchique) permet d'obtenir une méthode CO.

Duplication de données

La duplication de données consiste à stocker une même donnée à plusieurs emplacements mémoire afin de supprimer des indirections générant des accès mémoire aléatoires. Par exemple dans le cas d'une maillage comprenant un tableau de points et un tableau de cellules, chaque cellule contient une suite de pointeurs vers chacun de ces points. Lors du parcours de l'ensemble des cellules, si on a besoin des informations concernant chaque point, cela va générer des accès mémoire aléatoires dans le tableau des points. Si on utilise temporairement un tableau de cellules, chacune contenant également les informations de ces points, on peut transformer les accès aléatoires en accès séquentiels. En revanche ce tableau temporaire occupe plus d'espace mémoire que les deux tableaux initiaux car les informations d'un point sont dupliquées dans chaque cellule utilisant ce point. Pour construire le tableau temporaire de manière efficace, on peut utiliser plusieurs passes de tris, autant que le nombre de points contenus dans une cellule. On trie d'abord les cellules en fonction de l'indice du premier point et on copie les informations du premier point de chaque cellule en utilisant un accès séquentiel. Puis on trie en fonction de l'indice du second point etc. L'utilisation du tri puis de la duplication de données permet d'éviter tout accès aléatoire.

2.3.2 Performances en pratique des techniques CA et CO

Lors de l'utilisation de ces techniques, il faut prêter attention à deux effets importants : le *prefetching* et le surcoût en calculs.

Dans le cas des réorganisations CO, les accès mémoire sont souvent plus complexes que dans le cas de leur alternative CA, par exemple si on utilise un algorithme diviser pour régner ou une organisation mémoire récursive. Le *prefetching* matériel a donc plus de mal à prédire les accès aux données ce qui peut fortement réduire les performances, par exemple pour les algorithmes de multiplication de matrices denses [YRP+07].

Lors de l'utilisation d'organisations mémoire complexes, par exemple les organisations mémoire récursives ou basées sur les *space filling curves*, il est plus coûteux de calculer l'adresse d'une donnée ; cela peut augmenter le nombre d'instructions et ainsi contrebalancer le gain en temps obtenu par réduction du nombre de défauts de cache. Par exemple dans le cas des arbres binaires de recherche CO [LFBH00] ou de la multiplication de matrices basées sur les *space-filling curves* [CLPT02].

Pour pallier ces deux effets limitant les performances en pratique, on peut utiliser les techniques CO jusqu'à un seuil faible pour ensuite utiliser une technique classique non optimisée pour les caches mais permettant le *prefetching* et limitant le surcoût en calcul.

2.4 Algorithmes CA et CO : revue des résultats

Dans cette partie, nous présentons quelques algorithmes CO classiques puis nous comparons les performances des algorithmes CA et CO en théorie et en pratique.

2.4.1 Revue des algorithmes CA et CO

Multiplication de matrices

Nous avons présenté dans la section 2.2.2 un algorithme CO de multiplication de matrices basé sur le principe diviser pour régner. Il existe aussi un algorithme de multiplication de matrices basé sur la courbe de Peano [BZ05, BFH07] et un autre basé sur la courbe de Morton [CJR+99].

Dans [CLPT02], les auteurs comparent les performances de plusieurs *space filling curves* à la fois en terme de localité et en terme de surcoût en calculs. Pour l'algorithme en $O(n^3)$, les organisations mémoire récursives sont plus performantes que l'organisation ligne par ligne, les différentes *space filling curves* ont des performances équivalentes. Dans le cas des algorithmes sous-cubiques, Strassen et Strassen-Winograd, toutes les organisations mémoire, récursive et standard, se comportent de la même manière. Les algorithmes sous-cubiques étant basés sur le principe diviser pour régner, ils sont déjà relativement efficaces en cache.

Algorithmes de tri

Il existe un algorithme optimal CA pour le tri basé sur l'algorithme *mergesort* mais avec un facteur de branchement de $\frac{M}{B}$ [AV88]. A chaque étape, on fusionne $\frac{M}{B}$ sous-tableaux triés. L'étape de fusion cause $\frac{N}{B}$ défauts de cache si N est la taille de la séquence à trier. En effet, chaque sous-tableau est parcouru de manière séquentielle et il y a de la place dans le cache pour le bloc de taille B en cours de traitement de chaque sous-tableau. Au total, le nombre de défauts de cache est de

$$Q(N) = O\Big(\frac{N}{B} \cdot \log_{M/B} N\Big)$$

Cette borne est optimale [AV88].

Il existe deux algorithmes de tri CO. Le premier, *funnelsort*, est basé sur l'algorithme *mergesort*, le deuxième est basé sur l'algorithme distribution sort [FLPR99]. Ces deux algorithmes sont optimaux.

Recherche d'éléments

Nous avons vu dans la section 2.2.3 l'équivalent de l'arbre binaire de recherche pour le modèle CA, le B-arbre, ainsi que sa version CO. Une version simplifiée du B-arbre CO est présentée dans [BDW02]. Dans [BBF+03], les auteurs améliorent les performances de l'organisation mémoire van Emde Boas du B-arbre CO et montrent que ces performances sont optimales sur une hiérarchie mémoire avec une infinité de niveaux. Dans le cas où les éléments à chercher ne sont pas équiprobables, il existe un arbre CO optimal à un facteur constant près [BDFC02]. Pour des arbres de recherche multidimensionnels voir [ABF05].

Structures de données utilisant des pointeurs

La structure de base utilisant des pointeurs est la liste chainée. Dans [FPR09], les auteurs présentent une implémentation CA d'une liste chainée en respectant l'interface

de la STL. Le principe est de stocker de manière contigüe en mémoire des éléments consécutifs dans la liste. Des gains en performance notables sont observés dans le cas d'un simple parcours ou d'un tri.

Un équivalent CO aux listes chainées est un PMA (*packed memory array*) [BH07]. La liste est stockée dans un tableau de taille $\Theta(n)$, n étant le nombre d'éléments. On peut insérer et supprimer des éléments n'importe où dans la liste. Le PMA garantit qu'un parcours de k éléments consécutifs cause au plus $O(k/B)$ défauts de cache, ce qui est optimal. Cependant l'insertion et la suppression d'éléments sont en $O(\log_2^2 n)$.

2.4.2 Comparaison des algorithmes CA et CO

En théorie

Pour tous les problèmes que nous avons abordés dans les sections précédentes, nous avons vu qu'il existe toujours un algorithme CO atteignant asymptotiquement les mêmes bornes que le meilleur algorithme CA. On peut se demander si c'est toujours le cas, c'est-à-dire, est-ce que la connaissance des paramètres du cache apporte un avantage ? Il existe des problèmes pour lesquels les algorithmes CO sont moins efficaces que les algorithmes CA si on ne fait aucune supposition sur les valeurs relatives de la taille des lignes de cache B et la taille du cache M. On dit qu'un cache est grand (*tall cache assumption* ou TCA) si

$$M = \Omega(B^{1+\epsilon})$$

On suppose aussi parfois $M = \Omega(B^2)$. Dans le cas des algorithmes de tri, il n'existe pas d'algorithme CO atteignant les mêmes performances qu'un algorithme CA optimal si on ne suppose pas que les caches sont grands [BF03]. C'est aussi le cas des algorithmes de transposition de matrices [Sil06]. En pratique, la plupart des caches vérifient la TCA. Cependant il n'existe pas d'algorithme CO aussi bon que les meilleurs algorithmes CA pour appliquer une permutation même en supposant la TCA [BF03]. Il y a donc un vrai avantage à connaître les paramètres du cache.

En pratique

En pratique, les algorithmes CO souffrent des deux problèmes abordés dans la section 2.3.2 : le *prefetching* et le surcoût en calcul. Les algorithmes CA ont souvent des schémas d'accès plus simples que leur équivalent CO ce qui favorise le *prefetching* et limite le surcoût des calculs d'adresses.

Dans le cas des algorithmes de multiplication de matrices, les algorithmes CA sont aujourd'hui plus efficaces que les algorithmes CO [YRP+07]. La raison principale apportée par les auteurs est le *prefetching*. Pour les files de priorité, les algorithmes CO testés se comportent moins bien que le meilleur algorithme CA à cause du surcoût des calculs [OS02]. Pour les arbres de recherche, la version CO du B-arbre est légèrement moins performante que la version CA [LFBH00]. Dans le cas des algorithmes de tri, l'algorithme CO (*funnelsort*) testé dans [BFV04] est aussi rapide que l'algorithme CA (*multiway mergesort*) dans le cas des calculs ne dépassant pas la taille de la mémoire RAM, l'algorithme CA prenant l'avantage dès que les calculs s'effectuent en mémoire externe (en utilisant le disque dur).

Programmation parallèle par vol de travail 3

Sommaire

3.1 Programmation parallèle 40
3.1.1 Machine parallèle à mémoire partagée 40
3.1.2 Interfaces de programmation parallèle 42
3.1.3 Programmation parallèle à base de tâches 43
3.1.4 Exemple : algorithme parallèle de tri fusion 43
3.2 Ordonnancement d'un programme parallèle à base de tâches 44
3.2.1 Graphe de précédences, travail, profondeur 44
3.2.2 Ordonnancement avec une liste centralisée 44
3.2.3 Temps d'exécution du tri fusion 45
3.2.4 Surcoût d'une gestion centralisée des tâches 46
3.3 Ordonnancement par vol de tâches 46
3.3.1 Une liste de tâches décentralisée 47
3.3.2 Garantie sur le nombre de vols 47
3.4 Programmation par tâches efficace 50
3.4.1 Surcoûts par rapport au programme séquentiel 50
3.4.2 La gestion des listes de tâches 52
3.4.3 La création des tâches 52
3.4.4 La gestion des dépendances entre les tâches 53
3.4.5 Le surcoût algorithmique 54
3.5 Programmation parallèle adaptative 55
3.5.1 Algorithmes parallèles adaptatifs 55
3.5.2 Algorithme adaptatif de tri fusion 56
3.5.3 Moteur adaptatif à vol concurrent 57
3.5.4 Moteur adaptatif à vol coopératif 58
3.5.5 La préemption 59

Dans ce chapitre, nous présentons la programmation parallèle à base de tâches et ses avantages. Nous expliquons en détail l'ordonnancement par vol de travail et résumons son analyse théorique. Nous discutons ensuite des performances en pratique d'un programme parallèle ordonnancé par vol de travail en examinant les surcoûts par rapport au programme séquentiel. Enfin, nous présentons les algorithmes adaptatifs qui permettent de réduire ces surcoûts et comment adapter le moteur exécutif de vol de travail pour qu'il prenne en charge ces algorithmes.

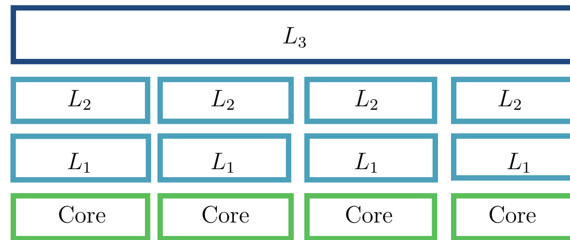


FIGURE 3.1 – Processeur multicœur avec quatre cœurs, deux niveaux de cache privés et un niveau de cache partagé.

Ce chapitre n'aborde pas la conception d'algorithmes parallèles et de structures de données concurrentes. On pourra consulter [BM96, MS05, HS08] pour plus de détails sur ces sujets.

3.1 Programmation parallèle

Nous présentons ici les techniques de programmation parallèle pour les machines à mémoire partagée. La programmation parallèle pour machines à mémoire distribuée dépasse le cadre de cette thèse.

3.1.1 Machine parallèle à mémoire partagée

Une machine parallèle est une machine qui comporte plusieurs cœurs de calculs. Chaque cœur est capable de gérer un flot d'instructions indépendant et possède un ou plusieurs caches. Ces cœurs sont regroupés dans un processeur multicœur qui est connecté avec la mémoire centrale. Un processeur peut avoir un dernier niveau de cache partagé entre tous les cœurs. Les processeurs actuels (AMD Opteron Shanghai ou Istanbul, AMD Phenom II, Intel Nehalem et IBM Power7) ont une organisation assez similaire (voir la figure 3.1) :

- De 2 à 10 cœurs ;
- Chaque cœur possède deux niveaux de cache privés L_1 et L_2 de faible capacité ;
- Le dernier niveau de cache L_3, de grande capacité, est partagé par tous les cœurs.

Dans certain cas, un cœur peut gérer plusieurs threads simultanément en entrelaçant les instructions, c'est la technologie *Symetric Multi Threading* (SMT) ou *Hyperthreading* (HT) chez Intel. Cette technique permet de mieux utiliser les unités de calcul d'un cœur si les threads utilisent des unités différentes (calcul entier ou calcul flottant par exemple). Elle permet aussi de recouvrir les lectures et écritures mémoire d'un thread par les calculs d'un autre thread. Si plusieurs threads sont exécutés sur le même cœur grâce à SMT on peut considérer qu'ils partagent les caches L_1 et L_2.

Dans une machine à mémoire partagée, l'ensemble des cœurs accèdent au même espace mémoire et communiquent par des lectures et des écritures en mémoire. Une machine à mémoire partagée actuelle est constituée de plusieurs processeurs multicœur et de plusieurs bancs mémoire reliés par des bus (par exemple Intel QuickPath ou AMD HyperTransport). Le coût d'un accès mémoire dépend du couple processeur-banc

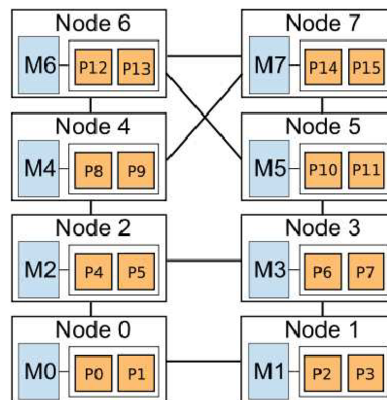


FIGURE 3.2 – Machine NUMA composée de 8 processeurs dualcore AMD Opteron 875. Chaque processeur est associé à un banc mémoire.

mémoire. On dit que les accès mémoire sont non uniformes (NUMA pour *non-uniform memory access*, voir la figure 3.2).

On peut considérer les cartes graphiques (*graphics processing unit* ou GPU) comme des machines à mémoire partagée. Une carte graphique comporte plusieurs cœurs de calculs qui accèdent à la mémoire mémoire globale. Les cartes graphiques modernes sont similaires aux processeurs multicœurs. Par exemple, le GPU Fermi de NVIDIA possède 16 cœurs, chacun ayant un cache privé L_1 et un cache L_2 partagé par tous les cœurs (voir la figure 3.3). Les différences majeures avec un processeur multicœurs sont les suivantes.

- Un cœur de GPU est plus simple qu'un cœur de processeur multicœur. Il n'y a pas de *prefetching* matériel et pas d'exécution des instructions dans le désordre (*out-of-order*). Cela a pour effet d'augmenter la latence perçue des accès mémoire.
- La taille des caches. Les caches des GPU ont une capacité plus faible.
- Les caractéristiques de la mémoire. La mémoire d'un GPU a une meilleure bande passante mais une moins bonne latence.
- L'hyperthreading. Un cœur de GPU peut gérer plus de threads en hyperthreading ce qui permet de réduire la latence perçue.
- La largeur SIMD. Un cœur de GPU peut exécuter la même instruction sur plusieurs données différentes en parallèle, jusqu'à 32 traitements en parallèle sur le GPU Fermi.
- La présence de matériel spécifique pour gérer les textures ou la rastérisation par exemple.

Un GPU sera plus efficace qu'un CPU pour des applications hautement parallèles qui exigent une bande passante élevée.

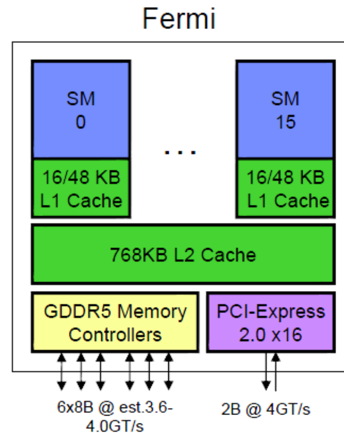


FIGURE 3.3 – Architecture du GPU Fermi de NVIDIA. Image de `www.realworldtech.com`.

3.1.2 Interfaces de programmation parallèle

La programmation parallèle consiste à écrire un programme qui peut s'exécuter sur plusieurs cœurs. Le concepteur d'un programme parallèle est amené à résoudre les problèmes suivants :

- identifier les sous-parties du calcul qui peuvent s'exécuter en parallèle ;
- gérer les dépendances entre les sous-parties du calcul ;
- gérer les accès concurrents aux structures de données ;
- répartir le travail de manière équilibrée entre les différents cœurs.

De nombreux outils existent pour écrire un programme parallèle, sous la forme de bibliothèque ou d'extension de langage.

La méthode de programmation parallèle de plus bas niveau est la programmation par threads [pth] et l'utilisation de primitives de synchronisation telles que les verrous (mutex) ou les instructions atomiques (*Test and Set*, *Compare and Swap*, etc.). Les autres outils de programmation parallèle sont basés sur les threads. En théorie, la programmation par threads est donc la méthode la plus efficace car toute application parallèle utilisant un autre outil peut être programmée de manière aussi efficace en utilisant les threads. En pratique, il est difficile de programmer efficacement des mécanismes de répartition de charge en utilisant les threads. Les applications par threads se contentent d'une répartition statique du travail moins efficace en général. On peut considérer que l'équivalent des threads pour programmer les GPU est le langage CUDA de NVIDIA [CUD10].

Il existe des interfaces de programmation parallèle de plus haut niveau tel que OpenMP [Ope] ou TBB [KV07]. Ces méthodes gèrent automatiquement les threads et proposent des primitives de synchronisation plus simples, par exemple la section critique de OpenMP garantit que la section concernée n'est utilisée que par un seul thread en même temps. De plus, ces méthodes gèrent aussi la répartition du travail entre les threads. Il est beaucoup plus simple de programmer avec ces outils et l'application produite est en général très efficace.

Une autre possibilité pour construire une application parallèle est d'utiliser des noyaux de calculs qui sont déjà parallélisés. Il existe des versions parallèles des BLAS

(*basic linear algebra subprograms*) pour les traitements d'algèbre linéaire. Il existe aussi des versions parallèles de certaines fonctions de la STL (*standard template library*). Les noyaux de calculs sont en général très efficaces mais il n'est pas toujours possible de les composer.

3.1.3 Programmation parallèle à base de tâches

La programmation parallèle par tâches est aujourd'hui un des standards pour la programmation parallèle. Elle est utilisée dans plusieurs langages et bibliothèques comme Cilk [FLH98], TBB [KV07], OpenMP [Ope], Task Parallel Library [LSB09], Kaapi [GBP07]. Le principe est d'écrire un programme qui crée des tâches et déclare des dépendances entre elles. En se basant sur les dépendances, le moteur exécutif décide quelles tâches peuvent être exécutées en parallèle et répartit la charge entre les processeurs.

Avantages. Dans une application parallèle à base de tâches, c'est le moteur exécutif qui s'occupe de répartir les tâches sur les différents cœurs. Il existe des ordonnanceurs de tâches efficaces en théorie comme en pratique. Le programmeur n'a donc pas à gérer la répartition du travail mais seulement l'expression du parallélisme à travers la création des tâches et la déclaration des dépendances. Cela permet au programmeur d'abstraire la machine, il peut penser en terme de tâches et non en terme de cœurs disponibles. Les programmes parallèles à base de tâches sont portables sur des machines à mémoire partagée.

La programmation par tâches est bien adaptée à l'expression du parallélisme imbriqué : une fonction parallèle qui appelle une autre fonction parallèle. Il est donc possible de composer facilement des algorithmes parallèles. Cela permet en particulier de programmer efficacement la classe importante des algorithmes parallèles diviser pour régner.

3.1.4 Exemple : algorithme parallèle de tri fusion

Voici un exemple de programmation par tâches avec le langage Cilk pour l'algorithme de tri fusion. Cet algorithme est de type diviser pour régner et donc se prête bien à une programmation parallèle par tâches. Le tri fusion divise le tableau à trier en deux sous-tableaux qui sont triés récursivement et ensuite fusionnés. Les tris des deux sous-tableaux peuvent être réalisés en parallèle. Nous créons donc deux tâches indépendantes avec le mot clé **spawn**, chacune réalisant le tri d'un sous-tableau. Puis une dernière tâche est créée pour fusionner les deux sous-tableaux en appelant la fonction MERGE. Cette tâche ne doit pas s'exécuter avant que les deux tâches de tri ne soient terminées : une relation de précédence entre ces tâches est spécifiée par le mot clé **sync**. Le point de synchronisation représenté par le **sync** ne sera franchi que lorsque toutes les tâches générées avant auront été exécutées.

Algorithme 1 Algorithme de tri fusion en Cilk

```
1: function MergeSort(A,i,j)
2:     if i < j then
3:         k ← (i+j)/2
4:         spawn MergeSort(A,i,k)
5:         MergeSort(A,k + 1,j)
6:         sync
7:         Merge(A,i,k,j)
8:     end if
9: end function
```

3.2 Ordonnancement d'un programme parallèle à base de tâches

Nous introduisons ici des notions théoriques pour analyser l'ordonnancement d'un programme parallèle à base de tâches.

3.2.1 Graphe de précédences, travail, profondeur

On représente une application par un graphe $G = (V, E)$ orienté sans cycle (DAG pour *directed acyclic graph*). Les sommets V du graphe sont des tâches et les arcs E des relations de précédences. Une tâche est un ensemble d'instructions qui doit s'exécuter séquentiellement. Une relation de dépendance entre deux tâches $i \rightarrow j$ indique que la tâche j ne peut s'exécuter que lorsque la tâche i est terminée. Le temps d'exécution de la tâche i est noté p_i. Le travail W (parfois noté T_1) est la somme des temps d'exécution de toutes les tâches :

$$W = \sum_{i \in V} p_i$$

La profondeur ou chemin critique D (parfois noté T_∞) est le temps d'exécution maximal sur un chemin C du DAG :

$$D = \max_C \sum_{i \in C} p_i$$

Le parallélisme d'une application est le rapport entre le travail et la profondeur soit $\frac{W}{D}$. On pourra consulter [Ble96, CLRS09] pour plus de détails sur le modèle du travail et de la profondeur.

3.2.2 Ordonnancement avec une liste centralisée

Pour exécuter une application représentée par son DAG, on peut utiliser une liste comprenant toutes les tâches prêtes. Quand un processeur est inactif, il retire une tâche de la liste pour l'exécuter. A la fin de l'exécution de cette tâche, si d'autres tâches sont prêtes, on les rajoute dans la liste. Cet ordonnancement est qualifié de glouton car si un processeur est inactif, la liste des tâches prêtes est vide. On peut analyser le temps d'exécution T_m (noté aussi $C_{\max}$) de cet ordonnancement sur une machine à m processeurs identiques. C'est le modèle $P|prec|C_{\max}$ [LKA04]. On néglige le coût de

gestion de la liste : insérer et retirer des tâches de la liste a un coût nul. On néglige aussi le temps nécessaire pour décider si une tâche est prête.

Théorème 3.1 (Graham [Gra69]). *Le temps d'exécution T_m d'un DAG de travail W et de profondeur D sur une machine à m processeurs par un ordonnancement par liste vérifie*

$$T_m \leq \frac{W}{m} + \left(1 - \frac{1}{m}\right) \cdot D$$

Soit T_m^* le temps d'exécution optimal du DAG. On a les relations suivantes :

$$T_m^* \geq \frac{W}{m} \text{ et } T_m^* \geq D$$

On peut donc en déduire que $T_m \leq 2 \cdot T_m^*$, l'ordonnancement par liste est à un facteur 2 de l'optimal. De plus si D est petit devant $\frac{W}{m}$ ou encore si le parallélisme de l'application est grand devant le nombre de processeurs, l'ordonnancement par liste est proche de l'optimal.

L'ordonnancement par liste est très intéressant car il n'utilise ni les temps d'exécution des tâches p_i ni la structure du graphe qui sont difficiles à obtenir en pratique. De plus cet algorithme est très simple et permet donc de limiter le surcoût du calcul de l'ordonnancement.

Pour plus de détails sur l'ordonnancement des DAG on pourra consulter [LKA04].

3.2.3 Temps d'exécution du tri fusion

Revenons sur l'exemple du tri fusion. La fusion est linéaire, on a donc un travail pour trier un tableau de n éléments de

$$W(n) = 2W\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

et une profondeur

$$D(n) = D\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n)$$

En utilisant un ordonnancement par liste, on a donc un temps de calcul sur m processeurs de

$$T_m(n) = \Theta\left(\frac{n \log n}{m}\right) + \Theta(n)$$

Le temps d'exécution de l'algorithme de tri fusion 1 n'est pas très bon car dans le cas où $m > log_n$, le temps d'exécution est dominé par la dernière fusion en $\Theta(n)$. On a besoin de plus de parallélisme pour que l'ordonnancement par liste soit efficace.

On peut améliorer le temps d'exécution de l'algorithme de tri fusion en parallélisant la fusion (voir figure 3.4). Pour fusionner deux tableaux en parallèle, on prend le médian x du premier tableau et on cherche sa position dans le deuxième tableau en utilisant une recherche dichotomique. On a ainsi divisé chacun des tableaux en deux sous-tableaux, les éléments inférieurs à x et ceux supérieurs à x. On peut maintenant fusionner en parallèle les deux sous-tableaux inférieurs à x et les deux sous-tableaux supérieurs à x.

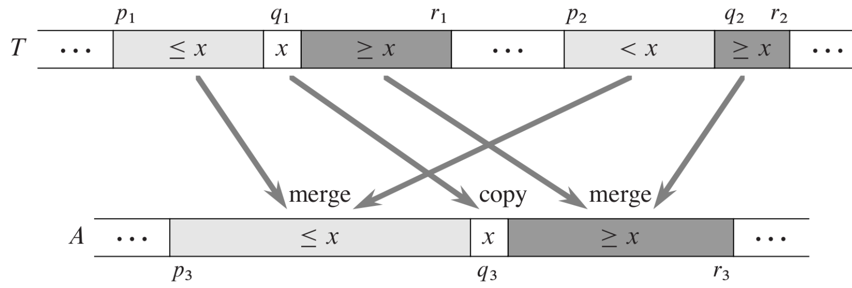


FIGURE 3.4 – Algorithme de fusion parallèle (image de [CLRS09])

Soit n_1 le nombre d'éléments inférieurs à x et n_2 le nombre d'éléments supérieurs à x. On a $n = n_1 + n_2 + 1$, $n_1 \geq n/4$ et $n_2 \geq n/4$. Le travail de l'algorithme de fusion parallèle est de

$$W(n) = W(n_1) + W(n_2) + \Theta(\log n) = \Theta(n)$$

La profondeur est de

$$D(n) = \max\Big(D(n_1), D(n_2)\Big) + \Theta(\log n) = \Theta(\log^2 n)$$

Si on utilise la fusion parallèle dans l'algorithme de tri fusion, la profondeur devient

$$D(n) = D\left(\frac{n}{2}\right) + \Theta(\log^2 n) = \Theta(\log^3 n)$$

ce qui est meilleur que la profondeur de $\Theta(n)$ obtenue avec la fusion séquentielle. Le temps parallèle de l'algorithme de tri fusion avec fusion parallèle est de

$$T_m(n) = \Theta\left(\frac{n \log n}{m}\right) + \Theta(\log^3 n)$$

Nous ne raffinerons pas cet algorithme de tri mais on se réfèrera à [Col88] pour un algorithme de tri fusion avec une profondeur de $\Theta(\log n)$.

3.2.4 Surcoût d'une gestion centralisée des tâches

Mettre en pratique l'ordonnancement par liste nécessite d'implémenter l'insertion et la suppression de tâches dans la liste. Cette liste étant utilisée de manière concurrente par toutes les unités de calcul, elle doit être protégée par des techniques classiques (utilisation d'un verrou ou liste *lock-free* avec insertion et suppression de tâches implémentées avec des instructions atomiques). Ces solutions sont peu efficaces en pratique car il y a beaucoup de contention sur la liste de tâches et les unités de calcul passent beaucoup de temps à attendre une nouvelle tâche à exécuter [KR04, HKR04].

3.3 Ordonnancement par vol de tâches

Une gestion distribuée de la liste de tâches permet d'éviter les problèmes mentionnés précédemment. Mais elle nécessite de revisiter quelque peu l'ordonnancement. En particulier, il faut montrer que le temps de complétion de l'ordonnancement n'est pas trop dégradé lorsqu'on utilise cette liste.

3.3.1 Une liste de tâches décentralisée

Pour réduire la contention et donc le surcoût de la gestion de la liste, on utilise une liste distribuée. Chaque unité de calcul a une liste locale. Tant que la liste locale n'est pas vide, une tâche de la liste est exécutée et les tâches devenues prêtes sont insérées. Lorsque la liste locale est vide, le processeur choisit au hasard et uniformément une autre liste et récupère si possible une ou plusieurs tâches pour les insérer dans sa liste. Cette opération est appelée un vol, le processeur initiant l'opération étant le voleur, le processeur ciblé la victime.

Une liste locale est toujours accédée de manière concurrente. On doit donc la protéger par un verrou ou utiliser une liste sans verrou. Cependant, comme le nombre de processeurs accédant à une liste au même moment est faible, le surcoût dû à la contention est réduit. Par contre cet ordonnancement n'est plus glouton : il est possible qu'une tâche soit disponible dans une des listes alors qu'un processeur est inactif (il n'exécute pas de tâche) car sa liste locale est vide. On ne peut donc pas appliquer la borne sur le temps d'exécution du théorème 3.1.

3.3.2 Garantie sur le nombre de vols

On veut obtenir une borne similaire à celle du théorème 3.1 dans le cas d'une liste distribuée. On néglige toujours les opérations de gestion des listes de tâches sauf pour le vol. On fait l'hypothèse qu'un vol dure un temps unitaire si il n'y a pas de contention, c'est-à-dire qu'un seul processeur tente d'accéder à la liste. Si plusieurs processeurs essayent de voler la même victime, un seul réussit et les autres échouent. On prend donc bien en compte les délais dûs à la contention sur les listes.

Soit S le nombre de vols. On a la relation suivante :

$$m \cdot T_m = W + S \tag{3.1}$$

En effet, un processeur est soit en train d'exécuter une tâche, soit en train de voler. Si on borne le nombre de vols S, on peut borner le temps d'exécution T_m.

On va borner le nombre de vol dans un cas particulier. On se limite à des tâches unitaires et aux DAG ayant une seul source (sommet sans arcs entrants) et un degré sortant d'au plus deux (au plus deux arcs sortant d'un sommet).

On spécifie de quelle manière on choisit les tâches à l'exécution et au moment du vol. L'exécution est en profondeur d'abord, le vol en largeur d'abord. L'exécution en profondeur permet de borner l'espace mémoire utilisé par le programme parallèle [BL99]. Le vol en largeur permet de voler des tâches proches de la source du DAG. Dans le cas courant des algorithmes diviser pour régner, les tâches proches de la source contiennent plus de travail. Pour réaliser cet ordonnancement, chaque processeur maintient sa liste locale comme une pile. Lorsqu'une nouvelle tâche est créée, elle est ajoutée en bas dans la pile. Lors de l'exécution, c'est la tâche en bas de la pile qui est exécutée. En revanche, lors d'un vol, c'est la tâche en haut de la pile qui est volée.

Théorème 3.2 (Arora, Blumofe, Plaxton [ABP98])**.** *Le nombre de vols S lors de l'exécution d'un DAG de tâches unitaires de travail W et de profondeur D avec une seule source et un degré sortant d'au plus 2 ordonnancé par vol de travail sur m processeurs vérifie*

$$\mathbb{E}\left[S\right] = O(m \cdot D)$$

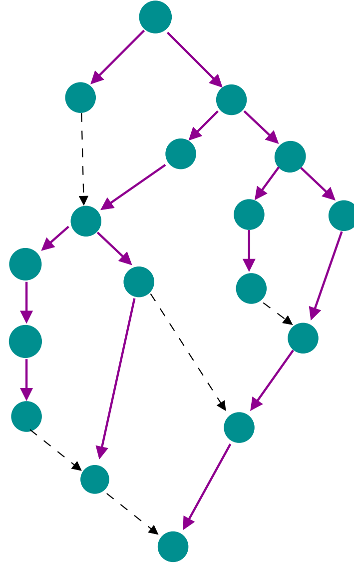


FIGURE 3.5 – Arbre d'activation. Les arcs pleins sont les arcs d'activation, les arcs en pointillés les arcs du DAG restants.

De plus, avec une probabilité supérieure à $1-\epsilon$*, le nombre de vols est borné par*

$$S = O\Big(m \cdot (D + \log(1/\epsilon))\Big)$$

Démonstration. La preuve borne le nombre de vols en utilisant une analyse amortie basée sur une fonction de potentiel. L'exécution est divisée en phases dans lesquelles le potentiel décroit d'un facteur constant avec probabilité constante.

On commence par définir la fonction de potentiel. Lorsque l'exécution d'une tâche i active (rend prête) une autre tâche j, on appelle l'arc $i \to j$ arc d'activation. On considère l'arbre ayant les mêmes sommets que ceux du DAG avec comme arcs le sous-ensemble des arcs d'activation (voir la figure 3.5). On peut montrer par induction que les parents dans l'arbre d'activation des tâches d'une pile sont sur un chemin racine feuille dans l'arbre d'activation (voir figure 3.6).

On définit le poids d'une tâche i par $w(i) = D - d(i)$ avec $d(i)$ la profondeur d'une tâche dans l'arbre d'activation. Le potentiel d'une tâche prête i est

$$\phi(i) = \begin{cases} 3^{2w(i)-1} \text{ si } i \text{ est en cours d'exécution} \\ 3^{2w(i)} \text{ sinon} \end{cases}$$

et le potentiel du système Φ_t à l'étape t est la somme de tous les potentiels des tâches prêtes. A l'état initial, seule la source du DAG est prête et $\Phi_0 = 3^{2D-1}$. A l'état final, toutes les tâches ont été exécutées et $\Phi_T = 0$. Le potentiel décroit lors de l'exécution, $\Phi_{t+1} \leq \Phi_t$, sous l'effet de l'exécution des tâches en bas de la pile et le vol de tâches en haut de la pile.

On analyse d'abord la diminution du potentiel d'une tâche.

1. Lors de l'exécution d'une tâche i, celle-ci active une ou deux tâches j_1, j_2 de

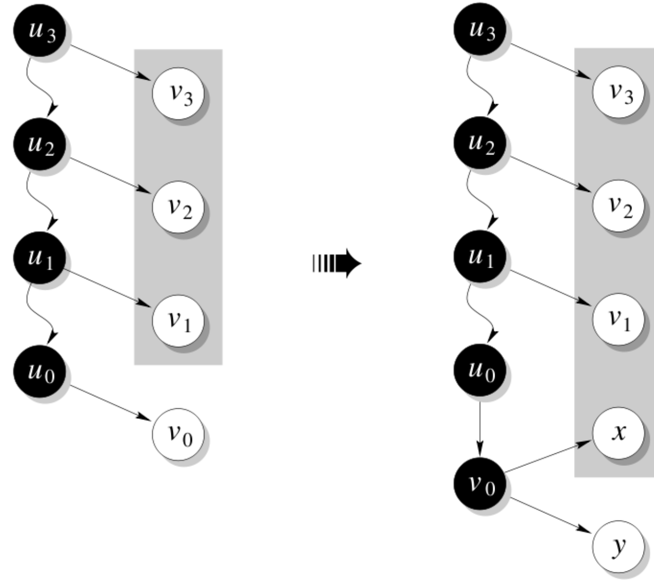


FIGURE 3.6 – Modification d'une pile de tâches lors de l'exécution d'une tâche v_0 qui active 2 tâches x et y. Lorsque la tâche v_0 termine son exécution, la tâche x est insérée en bas de la pile et la tâche y commence son exécution. La pile est grisée, les tâches en noir sont sur un chemin racine feuille dans l'arbre d'activation. Figure de [ABP98].

profondeur plus faible dans l'arbre d'activation. Le potentiel décroit de

$$\begin{aligned}\phi(i) - \phi(j_1) - \phi(j_2) &= 3^{2w(i)-1} - 3^{2w(j_1)} - 3^{2w(j_2)-1} \\ &= 3^{2w(i)-1} - 3^{2(w(i)-1)} - 3^{2(w(i)-1)-1} \\ &= \frac{5}{9} \cdot \phi(i)\end{aligned}$$

Si la tâche i active moins de 2 tâches, le potentiel décroit encore plus.

2. Lors du vol d'une tâche i, celle-ci devient en cours d'exécution et le potentiel décroit de

$$3^{2w(i)} - 3^{2w(i)-1} = \frac{2}{3} \cdot \phi(i)$$

On analyse maintenant la diminution du potentiel d'une pile de tâches. On définit le potentiel $\Phi(q)$ de la pile Q du processeur q par la somme du potentiel de toutes les tâches contenues dans Q, y compris la tâche en cours d'exécution. On a $\Phi(q) = \sum_{i \in Q} \phi(i)$.

1. Si le processeur q a une pile vide et aucune tâche en cours d'exécution $\Phi(q) = 0$.
2. Si le processeur q a une pile vide et une tâche i en cours d'exécution, le potentiel décroit d'au moins

$$\frac{5}{9} \cdot \phi(i) = \frac{5}{9} \cdot \Phi(q)$$

3. Dans le dernier cas, le processeur q a une pile non vide. Comme la profondeur des tâches augmente strictement dans la pile, le potentiel décroit géométriquement et le potentiel de la tâche i en haut de la pile du processeur q vaut au moins

$\phi(i) \geq \frac{3}{4} \cdot \Phi(q)$. Donc si le processeur q est volé, le potentiel décroit d'au moins

$$\frac{2}{3} \cdot \phi(i) \geq \frac{2}{3} \cdot \frac{3}{4} \cdot \Phi(q) = \frac{1}{2} \cdot \Phi(q)$$

On peut conclure de ces 3 cas que, quelque soit l'état de la pile du processeur q, si celui ci est volé, le potentiel décroit d'au moins $\frac{1}{2} \cdot \Phi(q)$.

On considère m tentatives de vols successives à partir de l'étape t jusqu'à l'étape t'. Si un processeur q reçoit au moins un vol, le potentiel décroit d'au moins $\frac{1}{2} \cdot \Phi(q)$. Si chaque processeur recevait exactement un vol, le potentiel diminuerait d'au moins $\frac{1}{2} \cdot \Phi_t$ et donc $\Phi_{t'} \leq \frac{1}{2} \cdot \Phi_t$. Comme les vols sont aléatoires, une telle répartition des vols n'est pas toujours réalisée. Cependant, on peut montrer avec une analyse probabiliste que le potentiel décroit d'au moins $\frac{1}{4}$ avec une probabilité supérieure à $\frac{1}{4}$

$$\mathbb{P}\left\{\Phi_{t'} \leq \frac{3}{4} \cdot \Phi_t\right\} > \frac{1}{4}$$

On dit qu'une phase de m tentatives de vols successives est réussie si le potentiel diminue d'au moins $\frac{1}{4}$. On a vu qu'une phase est réussie avec probabilité au moins $\frac{1}{4}$. Comme le potentiel initial est de $\Phi_0 = 3^{2D-1}$, il faut au moins $(2D-1)\log_{\frac{4}{3}} 3 < 8D$ phases réussies pour terminer l'exécution. On a donc besoin en moyenne de $32D$ phases. Comme chaque phase génère m vols, on a

$$\mathbb{E}[S] \leq 32 \cdot m \cdot D = O(m \cdot D)$$

La borne avec haute probabilité s'obtient en utilisant la borne de Chernoff [DP09]. □

La borne du théorème 3.2 sur le nombre de vols donne une borne sur le temps d'exécution en utilisant l'équation (3.1).

$$\mathbb{E}[T_m] \leq \frac{W}{m} + O(D) \tag{3.2}$$

Cette borne est similaire à celle du théorème 3.1 à un facteur constant près sur la profondeur. Si l'application a suffisamment de parallélisme, on a $\frac{W}{m} \gg D$ et cette perte d'un facteur constant influe peu sur les performances.

3.4 Programmation par tâches efficace

Dans la section précédente, nous avons vu comment ordonnancer un programme parallèle à base de tâches avec une faible contention sur les listes de tâches. Nous étudions ici l'implémentation d'un tel ordonnancement et analysons son coût.

3.4.1 Surcoûts par rapport au programme séquentiel

Pour avoir un programme parallèle à base de tâches efficace, il faut limiter les surcoûts par rapport au programme séquentiel. L'analyse de l'ordonnancement par vol de travail de la partie précédente prend uniquement en compte le surcoût de la contention lors de l'accès concurrent aux listes de tâches. Les autres coûts sont négligés. Il est néanmoins important de les considérer pour obtenir une implémentation efficace. Les sources potentielles de surcoût sont les suivantes.

La création des tâches. Le programme parallèle crée des tâches pour exprimer le parallélisme.

La gestion des dépendances. L'ordonnanceur a besoin de résoudre les dépendances pour déterminer les tâches prêtes.

La gestion des listes de tâches. L'ajout ou la suppression des tâches dans les listes lors de la création, l'exécution et le vol.

La contention de l'accès concurrent aux listes de tâches. Si plusieurs processeurs accèdent en même temps à une même liste de tâches, certains devront attendre si la liste est protégée par un verrou par exemple.

L'algorithme. Le programme séquentiel peut utiliser un algorithme de coût moindre que l'algorithme parallèle.

Soit F le nombre de tâches du programme parallèle [1]. Comme précédemment, W est le travail total de toutes les tâches sans prendre en compte les coûts de création de tâches et de gestion des listes de tâches. Soit S_m le nombre de vols lors d'une exécution sur m processeurs. On a vu dans la partie précédente que $S_m = O(m \cdot D)$ sous certaines hypothèses (théorème 3.2). Soit R le coût de résolution des dépendances. Le temps d'exécution du programme parallèle T_m vérifie

$$m \cdot T_m = W + c_f \cdot F + c_s \cdot S_m + R \tag{3.3}$$

La constante c_f représente le coût proportionnel au nombre de tâches du programme parallèle. La constante c_s représente le coût proportionnel au nombre de vols. Le coût de résolution des dépendances R n'est pas exprimé en fonction de F et S car il dépend aussi du nombre d'arcs du DAG. La constante c_f prend en compte les coûts suivants :

- création de la tâche
- insertion de la tâche dans une liste de tâches
- suppression de la tâche d'une liste de tâches lors de l'exécution locale ou au moment d'un vol (on considère que le coût de retirer une tâche de la pile en local est le même qu'au moment du vol)

La constante c_s prend en compte les coûts suivants :

- choix de la victime (génération d'un nombre aléatoire)
- coût de la prise de verrou ou du mécanisme de synchronisation sans contention

Le temps d'attente ou le surcoût de la contention au moment de l'accès à la liste de tâches est pris en compte dans le nombre de vols. On considère que si il y a de la contention sur la liste de tâches, une tentative réussit et les autres échouent et recommencent à voler.

On appelle grain g la quantité de travail moyenne d'une tâche

$$g = \frac{W}{F}$$

On peut réécrire l'équation (3.3) en

$$m \cdot T_m = \left(1 + \frac{c_f}{g}\right) \cdot W + c_s \cdot S_m + R \tag{3.4}$$

1. On note F le nombre de tâches en référence à l'opérateur FORK du langage ATHAPASCAN qui crée les tâches.

Le principe du *travail d'abord* [FLH98] indique qu'il vaut mieux réduire le surcoût des tâches, c'est-à-dire c_f, plutôt que réduire le surcoût des vols, c'est-à-dire c_s. En effet, dans l'hypothèse où $\frac{W}{m} \gg D$, le surcoût des tâches a plus d'importance que le surcoût des vols.

Dans la suite, nous décrivons plusieurs implémentations possibles pour le moteur d'exécution du vol de travail et nous analysons les impacts sur les différents coûts.

3.4.2 La gestion des listes de tâches

Il existe plusieurs implémentations de liste décentralisée. L'implémentation utilisée influence les coûts d'insertion, suppression et vol de tâches. L'efficacité de cette liste est donc critique pour obtenir une implémentation performante du vol de travail. L'insertion et la suppression de tâches en local s'ajoute au surcoût de tâches c_f alors que le vol s'ajoute au surcoût de vol c_s.

Dans [FLH98], Frigo *et al.* décrivent une implémentation basée sur le protocole THE. Suivant le principe du *travail d'abord*, Frigo *et al.* donnent une implémentation qui est rapide pour l'insertion et la suppression de tâches mais plus coûteuse pour le vol de tâches. Les différents voleurs se synchronisent en utilisant un verrou. Par contre la victime n'a besoin de prendre un verrou que dans un seul le cas : si la pile de tâches ne contient qu'une seule tâche et qu'un voleur essaye de récupérer celle-ci.

Arora *et al.* améliorent l'implémentation précédente en adaptant le protocole THE pour utiliser des CAS (instructions "compare and swap" atomiques) [ABP98]. Cette nouvelle implémentation est non bloquante, c'est-à-dire qu'un thread qui n'est plus ordonnancé par le système d'exploitation n'empêche pas la progression des autres threads. Cette implémentation non bloquante est un avantage lorsque les processeurs ne sont pas dédiés à une seule application. L'implémentation décrite dans [HS02a] permet le vol de plusieurs tâches en même temps ce qui peut améliorer l'équilibrage de charge et réduire le nombre de vols.

Dans toutes les implémentations précédentes, même si la manipulation en local des tâches (l'insertion et la suppression mais pas le vol) ne nécessite pas d'instructions atomiques de type CAS ni de verrous, il faut quand même utiliser une barrière mémoire qui empêche le processeur de réordonnancer les lectures et les écritures mémoire. Bien que cette opération soit moins coûteuse, cela engendre un surcoût proportionnel au nombre de tâches. L'implémentation de [MVS09] permet de supprimer cette barrière mémoire au prix d'une sémantique légèrement différente : il est possible qu'une tâche soit extraite de la pile plusieurs fois. Si l'application supporte qu'une tâche soit exécutée plusieurs fois ou qu'elle peut vérifier efficacement si une tâche a déjà été exécutée, cette implémentation peut apporter un gain de performance.

3.4.3 La création des tâches

Comparé au programme séquentiel, le programme parallèle exprime le calcul sous forme de tâches. Il faut donc créer ces tâches, c'est-à-dire leur allouer de la mémoire et y stocker la description du travail à effectuer. Une tâche contient en général du code (une fonction) et des données. Créer une tâche engendre un surcoût proportionnel au nombre de tâches qu'il est important de minimiser d'après le principe du *travail d'abord*.

Le surcoût de tâche c_f doit être amorti par la quantité de travail à faire pour exécuter la tâche. Plus le travail interne de la tâche est grand, plus le surcoût de création est faible. Des tâches de gros grain limitent le surcoût de création et aussi le surcoût de gestion des listes car il y a moins d'insertion/suppression de tâches. Par contre cela augmente la profondeur D et diminue l'efficacité de l'équilibrage de charge.

Une méthode pour choisir la taille du grain est proposée dans [DGG+07] pour la parallélisation de boucles. Le grain choisi permet de rendre le surcout de création de tâches asymptotiquement négligeable en augmentant la profondeur d'un facteur constant. Cette méthode est étendue dans [TRM+08] au cas des traitements de séquences de la STL dont les algorithmes parallèles peuvent présenter un surcout de travail comparé aux algorithmes séquentiels.

Dans TBB [RVK08], un mécanisme appelé `auto_partitioner` permet de choisir automatiquement le grain des tâches lors du traitement d'un ensemble de tâches (un `range` dans TBB). L'ensemble de tâches est initialement divisé en $4 \cdot m$ tâches, 4 par processeurs. Lors du vol, la moitié des tâches prêtes sont transférées. Lorsqu'un vol ne transfère qu'une seule tâche, celle-ci est redécoupée en 4 tâches. L'`auto_partitioner` permet d'adapter automatiquement le grain à l'exécution et donne de bonnes performances. Contrairement à la méthode de choix du grain de [DGG+07], la méthode proposée dans TBB utilise le nombre de processeurs.

Dans [HYY09], les tâches ne sont créées qu'au moment du vol. Lorsqu'un processeur reçoit une requête de vol, il remonte jusqu'au moment où il aurait dû créer la première tâche, il crée la tâche et reprend l'exécution de sa propre tâche. Le grain est donc parfait car on ne crée une tâche que dans le cas où elle est nécessaire, au moment du vol. Le surcoût des tâches est fortement diminué. Par contre, l'utilisation de cette technique force la victime à s'arrêter au moment d'une requête de vol pour créer la tâche, ce qui augmente le coût d'un vol c_s et peut à terme limiter le passage à l'échelle. Un tel mécanisme avait été préalablement proposé dans [Str98].

Dans [TCBV10], une tâche n'est créée que si la pile de tâches est vide. Pendant le traitement d'une tâche, on vérifie à intervalles réguliers que la pile est toujours non vide. Si jamais la pile est vide, une nouvelle tâche est créée. D'après [TCBV10], cette technique donne de meilleures performances que TBB `auto_partitioner`.

3.4.4 La gestion des dépendances entre les tâches

Lorsque le programme parallèle crée une tâche qui dépend d'autres tâches, il doit déterminer à quel moment la tâche devient prête, c'est-à-dire quand toutes les tâches en précédence ont été exécutées.

Dans Cilk [FLH98], toutes les tâches créées sont déjà prêtes. Pour déclarer des précédences entre les tâches, on utilise le mot clé `sync` qui permet d'attendre que toutes les tâches préalablement créées soient terminées. Si le programme parallèle crée des tâches après le `sync` elles seront déjà prêtes. Cette stratégie est intéressante car le surcoût de gestion des dépendances ne se retrouve qu'au moment du vol. Si une fonction qui crée des tâches n'a pas été volée, le `sync` n'exécute aucune instruction car il n'y pas besoin d'attendre de tâches, elles ont toutes été exécutées avant le `sync`.

Dans Kaapi [GBP07, GRW07], il est possible de créer des tâches non prêtes. Cela permet d'utiliser des DAG généraux contrairement à Cilk qui se limite aux DAG

séries parallèles. Chaque tâche maintient son statut (créée, en cours d'exécution, volée, terminée) ce qui augmente légèrement le coût c_f. Le coût d'un vol c_s est également impacté car un voleur doit chercher une tâche prête dans la pile de tâche. La résolution des dépendances est, comme dans Cilk, effectuée uniquement si une des tâches en précédence a été volée. Même si la possibilité de créer des tâches non prêtes augmente théoriquement les coûts c_f et c_s, des comparaisons expériementales montrent que la différence est faible en pratique [Gau10].

NABBIT [ALS10] est une bibliothèque portée au dessus de Cilk pour ordonnancer des DAG généraux. Dans NABBIT, le calcul des dépendances est toujours effectué. Chaque tâche a un compteur initialisé au nombre de dépendances. Lorsqu'une tâche termine son exécution, elle décrémente le compteur de toutes les tâches en dépendance. Lorsqu'un compteur arrive à 0, la tâche est déclarée prête. Le coût de la gestion des dépendances est proportionnel au nombre d'arcs du DAG. Ce mécanisme n'est pas très efficace si le grain est faible. NABBIT est plutôt destiné aux calculs à gros grains.

3.4.5 Le surcoût algorithmique

Le programme parallèle exécuté sur un seul processeur a un temps de $T_1 = (1 + c_f/g) \cdot W$. Si le grain des tâches est bien choisi on peut supposer que $c_f/g \approx 0$ et donc $T_1 \approx W$. Soit T_{seq} le temps d'exécution du programme séquentiel. Même dans ce cas idéal, on n'a pas obligatoirement $T_1 \approx T_{seq}$ car le programme séquentiel peut utiliser un meilleur algorithme que le programme parallèle. Dans certains cas, l'algorithme séquentiel ne peut pas être parallélisé sans rajouter des opérations. C'est ce qu'on appelle le surcoût algorithmique. Ce surcoût est particulièrement préjudiciable quand le programme parallèle est exécuté sur un faible nombre de processeurs. Dans certains cas, le programme parallèle est plus lent que le programme séquentiel.

L'algorithme de fusion parallèle présenté dans la partie 3.2.3 a aussi un surcoût algorithmique. A chaque division du travail en 2, il faut effectuer une recherche dichotomique. L'algorithme de fusion séquentiel n'effectue pas ces recherches dichotomiques. L'algorithme séquentiel a un coût de $W_{seq}(n) = n - 1$. Si on utilise l'algorithme parallèle tant qu'il reste au moins t éléments à fusionner pour ensuite passer sur l'algorithme séquentiel, le surcoût des recherches dichotomiques atteint $n \cdot \frac{\log_2 t}{t}$. En effet supposons que les deux tableaux à fusionner sont de tailles égales et que la recherche dichotomique divise toujours le deuxième tableau en deux parties égales. On a

$$W_{par}(n) = \begin{cases} 2 \cdot W_{par}\left(\frac{n}{2}\right) + \log_2 \frac{n}{2} \\ t - 1 \text{ si } n \leq t \end{cases}$$

Si on développe la récurrence, on obtient

$$\begin{aligned} W_{par}(n) &= \sum_{i=0}^{\log_2 \frac{n}{t} - 1} 2^i \cdot \log_2 \frac{n}{2^{i+1}} + \frac{n}{t} \cdot W_{seq}(t) \\ &= \frac{n}{t} \cdot \log_2 t - 1 - \log_2 n + n \cdot \left(1 - \frac{1}{t}\right) \\ &\sim n \cdot \left(1 + \frac{\log_2 t}{t}\right) \end{aligned}$$

L'algorithme de fusion parallèle effectue $n \cdot \frac{\log_2 t}{t}$ opérations de plus que l'algorithme séquentiel. On peut choisir $t = \log_2 n$ par exemple, ce qui rend le surcoût négligeable pour n grand. De plus la profondeur reste inchangée (à un facteur près). En effet, si $t = O(\log_2^2 n)$, la profondeur est toujours de $D(n) = O(\log_2^2 n)$.

Il est possible qu'un autre algorithme de fusion parallèle existe sans surcoût de travail mais il y a des problèmes pour lesquels un algorithme parallèle fera toujours plus d'opérations qu'un algorithme séquentiel. C'est le cas du calcul des préfixes.

Le problème des préfixes consiste à calculer pour n éléments $x_1, \ldots, x_n$ en entrée les n produits $\pi_1, \ldots, \pi_n$ avec

$$\pi_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k$$

sachant que l'opération $\otimes$ est associative. L'algorithme séquentiel pour résoudre ce problème est trivial et effectue $n - 1$ multiplications. Calculer les préfixes en parallèle a un surcoût algorithmique. Un algorithme de calcul des préfixes de n éléments de travail W et de profondeur D vérifie $W \geq 2(n-1) - D$ [Fic83]. Un algorithme parallèle avec la plus faible profondeur exécute presque 2 fois plus d'opérations que l'algorithme séquentiel.

L'opération `parallel_reduce` de TBB [RVK08] calcule en parallèle $\sum x_i$ en supposant que l'opération + est associative. Ce calcul est fait en parallèle en divisant en 2 le calcul de la somme puis en fusionnant les deux résultats obtenus. Lorsque la tâche calculant la première moitié est terminée, si la tâche calculant la deuxième moitié n'a pas été volée, la deuxième moitié est accumulée directement dans le résultat de la première moitié ce qui économise une fusion. Ce type d'optimisations permet de réduire le surcoût algorithmique.

3.5 Programmation parallèle adaptative

Nous présentons ici une méthode pour réduire le surcoût algorithmique et les coûts de création de tâches.

3.5.1 Algorithmes parallèles adaptatifs

Les programmes parallèles à base de tâches ne sont pas conscients du type d'ordonnancement qui est utilisé. Ils peuvent être ordonnancés par vol de travail mais aussi par un ordonnancement de liste centralisée tel que celui présenté dans la partie 3.2. La programmation parallèle adaptative propose d'écrire des programmes et des algorithmes parallèles qui sont conscients que leur exécution s'effectue par vol de travail. Ils peuvent donc réagir spécifiquement aux requêtes de vols, c'est-à-dire décider au moment du vol et non pas à priori quel sera le travail qui devra être effectué en parallèle. Il ne s'agit pas de créer une tâche uniquement lors d'un vol comme dans [HYY09] mais également de choisir dynamiquement le travail contenu dans la tâche.

Jusqu'ici on a supposé que le travail de l'application parallèle W était constant et ne dépendait pas de l'ordonnancement. Ce n'est plus le cas pour un programme adaptatif. Le travail dépend des requêtes de vols. Dans le cas où le programme parallèle s'exécute sur un seul processeur, il n'y a pas de vol et on voudrait $W = W_{seq}$. Lorsque le programme s'exécute sur m processeurs, le travail va augmenter à chaque requête de vol.

On voudrait que le travail W soit proche du travail W_m^* du meilleur algorithme parallèle statique sur m processeurs. Le programme adaptatif, par rapport au programme statique, a l'avantage de rester indépendant du nombre de processeurs comme les programmes à base de tâches et peut réagir à un environnement dynamique (vitesse des processeurs variable).

Les algorithmes adaptatifs ont été introduits dans [RTB06] pour résoudre le problème des préfixes. Un algorithme adaptatif a été donné pour les calculs de flux [BRT08]. Un schéma générique de construction d'algorithmes adaptatifs a été donné dans [DGK+05, TRM+08]. Ce schéma a été utilisé pour construire des algorithmes adaptatifs pour une grande partie de la STL [TRM+08]. Pour plus de détails sur les algorithmes adaptatifs, on peut consulter [Tra09].

Le schéma générique de [DGK+05, TRM+08] réalise un couplage entre deux algorithmes : un algorithme séquentiel réalisant peu d'opérations et un algorithme parallèle ayant une bonne profondeur. On suppose qu'à tout instant l'algorithme séquentiel peut être interrompu et que le travail restant peut être réalisé par l'algorithme parallèle. Lors d'une requête de vol, l'algorithme parallèle est utilisé pour diviser le travail restant en sous-parties qui pourront être exécutées en parallèle. Le traitement de ces sous-parties est effectué par l'algorithme séquentiel. Le nombre d'opérations peut augmenter au moment du vol à cause, par exemple, des étapes de fusion des résultats.

Une implémentation d'un programme adaptatif suivant le schéma générique sur un moteur de vol de tâches a été proposé dans [Tra09, DGK+05].

3.5.2 Algorithme adaptatif de tri fusion

Dans le cas du tri fusion parallèle, le surcoût algorithmique vient de la recherche dichotomique utilisée pour diviser le travail lors de la fusion parallèle de deux sous-tableaux triés. On propose donc de remplacer l'algorithme de fusion parallèle par un algorithme adaptatif suivant le schéma générique. La fusion commence en séquentiel. Lors d'une requête de vol, on divise le travail restant en utilisant une recherche dichotomique comme dans l'algorithme de fusion parallèle. Soit $A[1..p]$ et $B[1..q]$ les deux tableaux à fusionner. La fusion séquentielle a progressé dans A jusqu'à l'élément i et dans B jusqu'à l'élément j. Les tableaux $A[1..i-1]$ et $B[1..j-1]$ ont été fusionnés dans $C[1..i+j-2]$. On fait donc une recherche dichotomique de l'élément $A[\frac{i+n}{2}]$ dans B qui donne la position k. On exécute en parallèle la fusion des tableaux $A[i..\frac{i+n}{2}]$ et $B[j..k]$ et des tableaux $A[\frac{i+n}{2}+1..p]$ et $B[k+1..q]$.

Dans l'algorithme parallèle standard, on effectue toujours la recherche dichotomique même si il n'y pas de vol. Dans l'algorithme parallèle adaptatif, on n'effectue la recherche dichotomique uniquement si nécessaire, c'est-à-dire au moment d'un vol. On peut donc borner le surcoût algorithmique du programme adaptatif par

$$S_m \cdot O(\log n) = O(m \cdot D \cdot \log n) = O(m \cdot \log^3 n)$$

Le travail total du programme adaptatif est

$$W_{adapt} = W_{seq} + O(m \cdot \log^3 n)$$

On obtient la borne suivante pour le temps parallèle du programme adaptatif sur m

processeurs

$$T_m^{adapt} = \frac{W_{adapt}}{m} + O(\log^2 n) = \frac{W_{seq}}{m} + O(\log^3 n) = \frac{n}{m} + O(\log^3 n)$$

Pour plus de détails sur la fusion adaptative, on pourra consulter [Tra09].

Remarque. On peut borner plus finement le surcoût algorithmique dû aux vols en considérant les vols réussis et les vols ratés séparément. En effet, il n'y a de recherche dichotomique que lors d'un vol réussi. On peut montrer que le nombre de vols réussis est de $O(m \cdot \log n)$. Le travail total du programme adaptatif est donc $W_{adapt} = W_{seq} + O(m \cdot \log^2 n)$.

3.5.3 Moteur adaptatif à vol concurrent

La programmation adaptative a été motivée principalement par la réduction du surcoût algorithmique, mais elle permet aussi de limiter les autres surcoûts présentés dans la partie 3.4.1 si le moteur d'exécution propose une interface spécifique.

Interface du moteur exécutif

On suppose que le moteur de vol de travail fonctionne comme suit :
- la description du travail restant à réaliser est stockée dans un objet `Work` : c'est l'équivalent d'une tâche ;
- lorsque l'algorithme séquentiel s'exécute, il extrait une petite partie du travail contenu dans `Work` avec l'opération `extract_seq` ;
- lors d'un vol, une grosse partie du travail (usuellement la moitié) contenu dans `Work` est extraite avec l'opération `extract_par` et est stocké dans un nouvel objet de type `Work` à destination du voleur ;
- le voleur exécute l'algorithme adaptatif à partir du nouvel objet `Work` ;
- on suppose que les accès aux objets de type `Work` n'entrent pas en concurrence (par exemple avec un simple verrou ou un protocole similaire à THE).

Implémentation d'une boucle parallèle

Pour implémenter un algorithme adaptatif de traitement d'une boucle parallèle (par exemple le `parallel_for` de TBB), l'objet `Work` contient simplement l'intervalle d'indices à traiter et un pointeur vers la fonction qui traite un élément. La fonction `extract_seq` extrait de l'objet `Work` un petit intervalle à traiter, la fonction `extract_par` divise l'intervalle en deux et place la deuxième moitié dans un nouvel objet `Work` (voir l'algorithme 2).

Surcoûts du programme adaptatif

Le programme adaptatif limite le surcoût algorithmique mais aussi le coût de création des tâches. En effet un nouvel objet `Work` n'est créé qu'au moment du vol (comme dans [HYY09]). De plus créer un objet `Work` est en général moins coûteux que la création d'une tâche. La gestion de la concurrence sur les objets `Work` est similaire au protocole THE donc cette implémentation n'apporte pas de gain sur ce facteur.

Algorithme 2 Algorithme adaptatif pour une boucle parallèle

```
function EXTRACT_SEQ(w)
    [i, j] ← w
    k ← i + g
    if k > j then
        k ← j
    end if
    w ← [k + 1, j]
    return [i, k]
end function

function EXTRACT_PAR(w)
    [i, j] ← w
    if i > j then
        return failed
    end if
    w ← [i, (i+j)/2]
    w2 ← [(i+j)/2 + 1, j]
    return w2
end function

function PARALLEL_FOR(w)
    while w ≠ ∅ do
        [a, b] ← EXTRACT_SEQ(w)
        for i ∈ [a, b] do
            F(i)
        end for
    end while
end function

function ON_STEAL(w)
    Work w2 ← EXTRACT_PAR(w)
    PARALLEL_FOR(w2)
end function

function MAIN
    Work w_init ← [1, n]
    PARALLEL_FOR(w_init)
end function
```

Le seul surcoût proportionnel au travail est le coût des appels à la fonction `extract_seq`. Les appels à `extract_seq` sont moins coûteux que la création d'une tâche, mais lorsque le temps de traitement d'une unité de travail est faible il faut tout de même amortir le coût de ces appels avec un grain, la valeur g dans l'algorithme 2. Dans l'idéal, on aimerait supprimer complètement ce coût mais dans l'implémentation actuelle le travail en cours doit être maintenu pour permettre le vol.

3.5.4 Moteur adaptatif à vol coopératif

On peut supprimer les appels à `extract_seq` si le vol ne s'effectue plus en concurrence par le voleur pendant que la victime continue à travailler. Dans [BLTG09, Bes10], un vol en coopération entre la victime et le voleur est proposé. Lors d'un vol, la victime interrompt son travail pour traiter la requête de vol du voleur. L'avantage du vol coopératif est que la victime ne maintient pas le travail restant à tout instant mais seulement au moment du vol. On a donc déplacé un surcoût de travail en surcoût de vol ce qui est bénéfique d'après le principe du *travail d'abord*. Cependant le coût du vol augmente. En effet le voleur doit attendre la réponse de la victime et la victime interrompt son travail pour répondre au voleur.

Même avec le vol coopératif, il reste un faible surcoût de travail. La victime doit tester régulièrement qu'elle n'a pas reçu de requête de vol. Mais ce test est très peu coûteux, c'est un simple test sur un entier.

3.5.5 La préemption

Lors de l'exécution d'un algorithme avec surcoût de parallélisme, il peut être intéressant de revenir sur le travail donné à faire en parallèle car son traitement par l'algorithme parallèle est plus coûteux que son traitement par l'algorithme séquentiel. Par exemple, si après un vol, la victime termine sa part du travail avant le voleur, on voudrait reprendre le travail restant à traiter par le voleur pour le rendre à la victime. Le mécanisme de préemption permet d'arrêter le dernier voleur et de récupérer le travail restant.

La préemption permet de réduire le surcoût algorithmique mais elle est aussi utile dans d'autres cas, par exemple pour les algorithmes à terminaison anticipée [Tra09]. Un exemple d'un tel algorithme est la fonction `find_if` de la STL qui retourne le premier élément d'une séquence satisfaisant une condition. Si la victime trouve l'élément recherché, elle peut arrêter tous ses voleurs qui recherchent inutilement dans d'autres parties de la séquence, en utilisant la préemption.

Dans le cas du moteur à vol coopératif, il est facile d'implémenter la préemption. Chaque processeur doit vérifier régulièrement si il n'a pas reçu de requêtes de préemption. Cela peut être fait au même moment que le test des requêtes de vols.

Algorithmes parallèles efficaces en cache 4

Sommaire

4.1 Impact du parallélisme sur les caches 61
4.1.1 Caches privés et caches partagés 62
4.1.2 La cohérence de cache 62
4.1.3 Modéliser les accès mémoire d'un algorithme parallèle 63
4.2 Ordonnancement pour caches privés 64
4.2.1 Modéliser les caches privés 64
4.2.2 Ordonnancement par vol de travail pour caches privés 65
4.2.3 Raffiner l'analyse pour les algorithmes CO 67
4.2.4 Algorithmes parallèles pour caches privés 68
4.3 Ordonnancement pour un cache partagé 68
4.3.1 Modéliser un cache partagé 68
4.3.2 Ordonnancement avec liste centralisée pour cache partagé . . 69
4.3.3 Algorithmes parallèles pour cache partagé 70
4.4 Vers un ordonnanceur pour le cas général 71
4.4.1 Modèle combinant caches privés et caches partagés 71
4.4.2 Approches pour traiter le cas général 71
4.4.3 Algorithmes parallèles pour multicœurs 72
4.4.4 Ordonnancements basés sur l'affinité 73

Nous avons vu dans les deux chapitres précédents comment concevoir des algorithmes efficaces en cache, en particulier les algorithmes CO qui ne dépendent pas des paramètres de la hiérarchie mémoire, et comment ordonnancer efficacement des programmes parallèles en utilisant le vol de travail. Dans ce chapitre, nous étudions comment combiner ces deux approches pour concevoir des algorithmes parallèles efficaces en cache.

4.1 Impact du parallélisme sur les caches

Dans les chapitres 1 et 2, nous avons décrit et modélisé la hiérarchie mémoire vue par un programme séquentiel. Nous examinons dans cette section quels sont les phénomènes supplémentaires à prendre en compte dans le cadre d'un programme parallèle.

4.1.1 Caches privés et caches partagés

Une première distinction dans le cas d'une architecture parallèle est la présence de deux types de caches : les caches privés et les caches partagés. Un cache privé n'est connecté qu'à un seul cœur alors qu'un cache partagé est connecté à plusieurs cœurs (*cf.* figure 3.1 page 40).

Sur une architecture à caches privés, l'exécution de plusieurs applications séquentielles différentes ne perturbe pas le contenu des caches. Chaque application dispose de sa propre copie du cache. Sur une architecture à cache partagé disposant de la même quantité de cache, le comportement d'une application est fortement influencé par la présence d'autres applications utilisant le même cache [CGKS05]. Plusieurs travaux ont étudié comment allouer, de manière logicielle ou matérielle, une partie du cache partagé à chaque application [SRD04, TASS08]. Ces techniques permettent de réduire les perturbations entre plusieurs applications occasionnées par la présence de cette ressource partagée. L'architecture à cache partagé peut offrir de meilleures performances si la quantité de cache allouée à chaque application est adaptée à ses besoins et non pas fixée par la configuration matérielle, cas d'une architecture à caches privés [CS06].

Dans le cadre de l'exécution d'une seule application parallèle composée de plusieurs threads, une architecture à cache partagé peut offrir de meilleures performances si les threads utilisent des données partagées. En effet, ces données partagées doivent être dupliquées dans tous les caches si ceux si sont privés ce qui réduit la capacité effective du cache. Des études récentes ont montré que certaines applications parallèles contiennent une part importante de partage de données et qu'il est possible d'en tirer parti pour obtenir de meilleures performances [JMJ06, ZJS10].

4.1.2 La cohérence de cache

Un autre phénomène important qui n'apparait pas dans le cadre d'une exécution séquentielle est la cohérence des données stockées dans des caches privés. Si plusieurs copies d'une même donnée sont chargées dans différents caches privés et qu'une de ces copies est modifiée par une écriture, les autres copies deviennent invalides (*cf.* figure 6 page 7). C'est le rôle du protocole de cohérence de cache d'invalider les copies contenues dans les autres caches. Lorsqu'un thread accède à une copie invalidée, il doit aller chercher une copie valide en mémoire et produit donc un défaut de cache. Ce défaut de cache n'aurait pas été présent lors d'une exécution séquentielle. On peut qualifier ce défaut de cache de "défaut de cache de cohérence" ou *coherency miss*. Ce type de défauts de cache s'ajoutent aux trois autres types identifiés dans la section 1.1.4.

Les défauts de cache de cohérence peuvent se produire même quand les threads ne partagent pas de données mais que les adresses de ces données se trouvent dans le même segment mémoire de la taille d'une ligne de cache. Comme le protocole de cohérence de cache fonctionne à la granularité d'une ligne de cache, il ne peut pas différencier des accès à l'intérieur d'une ligne. On appelle ce phénomène le *false sharing*.

Il existe plusieurs protocoles de cohérence de cache, les deux plus classiques sont le *snooping* et le *directory based* détaillés dans [HP06]. La conception de protocoles de cohérence de cache qui passent à l'échelle est un domaine de recherche très actif.

4.1.3 Modéliser les accès mémoire d'un algorithme parallèle

La dernière distinction entre les programmes séquentiels et les programmes parallèles est le modèle de description des accès mémoire. Pour analyser les défauts de cache d'un algorithme parallèle, on a besoin d'un modèle qui enrichisse le modèle du DAG pour prendre en compte les accès mémoire de l'algorithme.

Le DAG utilisé dans le chapitre 3 pour modéliser les algorithmes parallèles décrit les tâches à exécuter et les précédences à respecter pour obtenir un ordre d'exécution valide. Une tâche n'est caractérisée que par la quantité de travail (nombre d'instructions) p_i à exécuter. Ce modèle est à la fois simple à utiliser et représente assez fidèlement un programme parallèle réel.

A notre connaissance, il n'existe pas de modèle d'accès mémoire en parallèle qui combine précision et simplicité d'utilisation. Pour comprendre quelle est l'origine de cette difficulté, on peut prendre l'exemple de deux tâches indépendantes (dans le DAG) qui s'exécutent en parallèle sur deux cœurs différents. Pour calculer l'état des caches privés de ces deux cœurs (ou d'un cache partagé), on doit savoir comment s'entrelacent les accès mémoire de ces deux tâches. Il faut donc descendre à un grain très fin au niveau de la description de l'algorithme, chaque tâche n'exécutant qu'un faible nombre d'accès mémoire. De tels modèles existent mais ils sont en général difficiles à résoudre [GMM97]. Le plus souvent, on ne connait pas d'algorithmes d'approximation avec un facteur d'approximation constant [JL95]. En comparaison, pour un DAG sans accès mémoire l'algorithme de liste de Graham [Gra69] présenté dans la section 3.2.2 est simple et donne un ordonnancement à un facteur 2 de l'optimal (*cf.* section 3.2.2).

Une des solutions qui a été utilisée par Blelloch *et al.* dans [BG04] est de se comparer à une exécution séquentielle. Au lieu d'analyser directement les défauts de cache de l'algorithme parallèle, on commence par analyser les défauts de cache sur une exécution séquentielle comme dans le chapitre 2, puis on compte les défauts de cache supplémentaires qui sont liés à l'exécution parallèle. Cette approche est plus simple car elle permet de découpler l'analyse des défauts de cache à grain fin lors de l'exécution séquentielle de l'analyse des défauts de cache à gros grain lors de l'exécution parallèle. Mais elle a deux limitations principales.

– Pour obtenir une exécution parallèle réalisant peu de défauts de cache supplémentaires sur un cache partagé, l'algorithme parallèle doit suivre l'exécution séquentielle au plus près ce qui conduit à des algorithmes potentiellement peu efficaces. De plus, si l'exécution parallèle diverge fortement de l'exécution séquentielle alors l'évaluation du nombre de défauts de cache supplémentaire devient imprécise, les bornes supérieurs obtenues s'éloignent rapidement des valeurs réelles.
– Pour un cache partagé, l'exécution séquentielle optimale en cache est toujours meilleure que l'exécution parallèle optimale en cache. En effet, il est toujours possible de prendre comme exécution séquentielle, une modification de l'exécution parallèle dans laquelle on exécute une instruction de chaque flot d'exécution à tour de rôle [1]. Dans le cas de caches privés, l'exécution séquentielle optimale peut être moins bonne que l'exécution parallèle optimale. En effet, l'exécution parallèle sur p cœurs dispose globalement de p fois plus de cache que l'exécution

1. En inversant ce procédé, on ne peut pas transformer une exécution séquentielle en exécution parallèle valide à cause des contraintes de précédence.

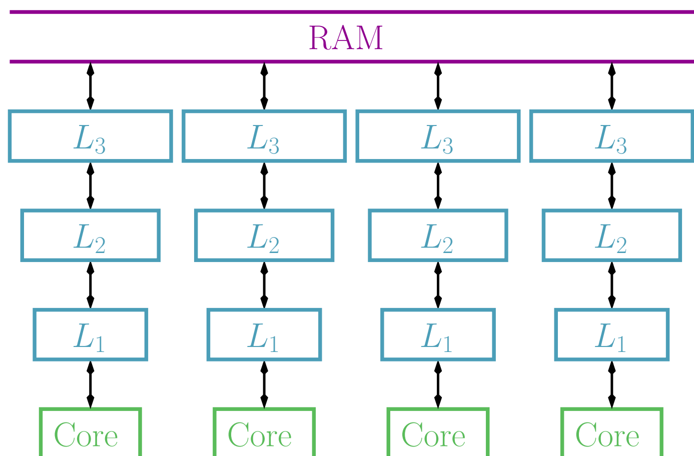


FIGURE 4.1 – Modèle d'une architecture à caches privés

séquentielle et peut donc générer moins de défauts de cache. Les accélérations super-linéaires observées avec certains algorithmes sont classiquement le résultat de cette capacité de l'algorithme parallèle à cacher plus de données. Cet effet ne peut être modélisé en comptant les défauts de cache supplémentaires par rapport à l'exécution séquentielle.

4.2 Ordonnancement pour caches privés

Nous nous intéressons en premier lieu à l'ordonnancement d'une application parallèle dans le cas où tous les caches sont privés.

4.2.1 Modéliser les caches privés

On considère une hiérarchie mémoire dans laquelle chaque cœur possède un ou plusieurs niveaux de cache privé, tous ces cœurs accédant à la même mémoire centrale (*cf.* figure 4.1). Les cœurs ne partagent aucun niveau de cache.

Dans une telle architecture, la difficulté pour analyser les défauts de cache d'un algorithme parallèle vient principalement des défauts de cache de cohérence. Ce sont ces défauts de cache qui empêchent d'analyser l'état des caches de façon indépendante du comportement des autres cœurs. Les travaux de la littérature se concentrent dans le cas où on peut obtenir cette indépendance [BFJ+96b, BFJ+96a, ABB02, FS09]. Quand cette hypothèse est satisfaite, on qualifie les caches de *non interfering*.

Une des méthodes pour obtenir des caches *non interfering* est d'utiliser un protocole de cohérence de cache basé sur le DAG comme dans [BFJ+96b, BFJ+96a]. Dans un tel protocole, la cohérence des données n'est réalisée que pour une relation de précédence $u \to v$ entre deux tâches qui ont été exécutées sur deux cœurs différents P_1 et P_2. Après l'exécution de u mais avant l'exécution de v, le protocole met en cohérence les données chargées dans les caches des cœurs P_1 et P_2. Il ne peut donc y avoir de défaut de cache de cohérence que lorsqu'un cœur exécute une tâche qui était en précédence avec une tâche exécutée sur une autre cœur. Il est par contre nécessaire de s'assurer que le programme est correct lorsque l'on utilise ce protocole basé sur le DAG. Deux

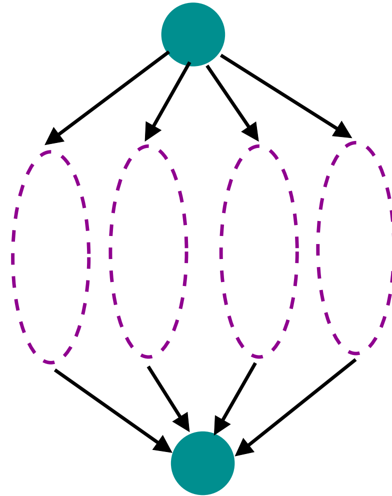


FIGURE 4.2 – DAG série parallèle composé d'une tâche *fork* en haut, une tâche *join* en bas, reliant des DAGs en pointillés qui sont eux aussi série parallèle.

tâches qui ne sont pas reliées par un chemin de précédence dans le DAG peuvent voir des copies différentes d'une même donnée. Potentiellement très efficace, ce protocole n'est cependant pas supporté par les architectures actuelles.

L'autre méthode pour obtenir cette hypothèse d'indépendance consiste à limiter l'étude à une classe plus restreinte de programmes parallèles comme dans [ABB02]. Un programme parallèle est *fully strict* ou *fork join* si sa représentation sous forme de DAG est série parallèle (*cf.* figure 4.2). Dans ce cas, le DAG a une structure spécifique qui permet de reconnaitre des sous-ensembles de tâches sans relation de précédence entre eux. On demande également que le programme parallèle soit *race free*, c'est-à-dire qu'une tâche ne peut lire une donnée écrite par une autre tâche que si elles sont reliées par un chemin dans le DAG. En pratique, on dit qu'un programme parallèle contient une *race* lorsqu'un cœur lit une donnée écrite par un autre cœur sans précautions. On peut utiliser des barrières mémoire pour garantir que l'écriture de la donnée soit visible au moment de sa lecture par un cœur. Ici, on suppose qu'une telle opération sera réalisée en utilisant une synchronisation. Enfin, on suppose également qu'il n'y a pas de *false sharing*, c'est-à-dire qu'il n'y a pas de données différentes dans la même ligne de cache.

On pourrait croire que la première méthode utilisée par [BFJ+96b, BFJ+96a] est plus générale car elle ne fait pas d'hypothèse sur la structure du DAG. En fait, même si le protocole de cohérence est valide pour un DAG général, les analyses présentées dans [BFJ+96a] ne considèrent que les programmes parallèles *fully strict*. En effet, il parait difficile de borner le nombre de défauts de cache de cohérence lorsqu'un sous-ensemble de tâches peut se synchroniser arbitrairement avec une autre sous-ensemble. Dans la suite, nous utiliserons la deuxième méthode, celle de [ABB02]. Les bornes obtenues par la première méthode sont similaires.

4.2.2 Ordonnancement par vol de travail pour caches privés

Nous nous intéressons maintenant au nombre de défauts de cache supplémentaires par rapport à l'exécution séquentielle d'un programme parallèle dans le cas où les caches

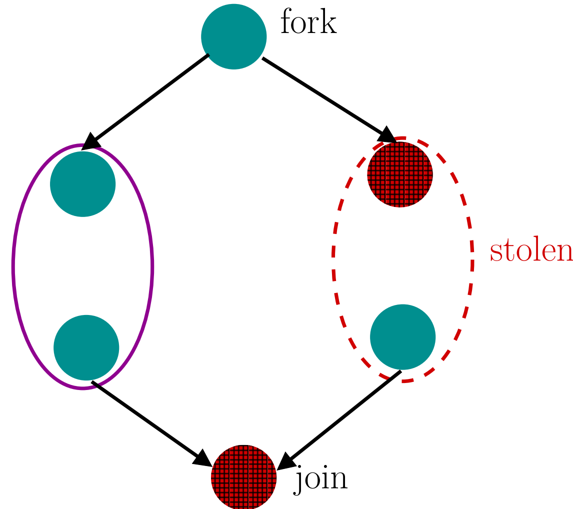


FIGURE 4.3 – Lors d'un vol, deux tâches (hachurées) sont potentiellement exécutées dans le désordre par rapport à l'exécution séquentielle en profondeur d'abord.

peuvent être considérés comme *non interfering*. Comme dans le chapitre 3, l'exécution séquentielle correspond à l'ordre en profondeur d'abord.

Théorème 4.1 (Acar *et al.* [ABB02])**.** *Un programme parallèle fully strict de profondeur D avec un degré sortant d'au plus 2 ordonnancé par vol de travail génère*

$$O\Big(\frac{M}{B} \cdot m \cdot D\Big)$$

défauts de cache supplémentaires par rapport à l'exécution séquentielle sur une architecture avec m cœurs, chacun ayant un cache privé de taille M et des lignes de taille B.

Démonstration. L'approche de [ABB02] fonctionne en deux étapes :

1. Montrer que le nombre de défauts de cache supplémentaires est majoré par le produit du nombre de lignes dans le cache M/B et du nombre de tâches exécutées "dans le désordre" par rapport à l'exécution séquentielle.
2. Montrer que le nombre de tâches exécutées "dans le désordre" est au plus 2 fois le nombre de vols S.

Il est ainsi facile de conclure en utilisant la borne sur le nombre de vols de [ABP98] présentée dans la section 3.3.2 : $S = O(m \cdot D)$.

Pour montrer la première propriété, Acar *et al.* considèrent le résultat d'une même suite d'accès mémoire en partant de deux caches chargés initialement avec des données différentes. Sous certaines conditions sur la politique de remplacement, les auteurs montrent que le nombre de défauts de cache diffère d'au plus M/B, c'est-à-dire un défaut pour chaque ligne de cache. En effet, après M/B défauts de cache supplémentaires, le contenu du cache est le même quel que soit le contenu initial. Ces conditions sont vérifiées pour les caches gérés par la politique LRU ou la politique FIF [2]. Quand une

2. On pourra consulter l'article original [ABB02] pour une description des politiques de remplacement qui vérifient cette propriété.

tâche est exécutée dans le désordre, l'état du cache n'est pas le même que dans le programme séquentiel et donc l'exécution de cette tâche et des tâches suivantes activées par cette tâche génère au plus M/B défauts de cache supplémentaires par rapport à l'exécution séquentielle. On a besoin ici de l'hypothèse des caches *non interfering* pour garantir qu'aucun autre cœur ne vienne perturber le contenu du cache.

Pour montrer la deuxième propriété, Acar *et al.* remarquent que lors d'un vol, au plus deux tâches sont exécutées dans le désordre par rapport à l'exécution séquentielle (*cf.* figure 4.3). La première tâche exécutée dans le désordre est la tâche volée. Une tâche volée est forcément la fille d'une tâche de type *fork*. Cette tâche ne sera exécutée dans l'ordre séquentiel que lorsque toutes les autres tâches générées par le fils gauche de la tâche de *fork* seront exécutées sauf pour la tâche de *join* correspondant au *fork* (l'ensemble de tâches entourées en trait plein sur la figure). La deuxième tâche potentiellement exécutée dans le désordre est la tâche de *join*. Si le sous-ensemble des tâches volées (entouré par un pointillé) termine avant le sous-ensemble restant sur le processeur victime (entouré en train plein), la tâche de *join* sera exécutée juste après l'exécution du sous-ensemble gauche. Dans l'exécution séquentielle, la tâche de *join* est exécutée après le sous-ensemble droit. □

On peut énoncer une version informelle du théorème 4.1 : à chaque vol, un cœur doit "chauffer son cache".

4.2.3 Raffiner l'analyse pour les algorithmes CO

La borne donnée par le théorème 4.1 est atteinte dans le pire des cas mais sur des exemples dont les accès mémoire n'ont pas une bonne localité. Si les programmes parallèles ont une bonne localité, comme les algorithmes CO, on peut obtenir une meilleure garantie sur le nombre de défauts de caches.

Dans [FS09], Frigo *et al.* considèrent une architecture à caches privés dans laquelle chaque cache est géré par la politique de remplacement optimale FIF comme pour le modèle CO séquentiel. Ils supposent qu'on peut majorer le nombre de défauts de cache générés par l'exécution séquentielle d'une séquence consécutive d'instructions $\mathcal{A}$ du programme par une fonction concave f de son nombre d'instructions $|\mathcal{A}|$. Dans ce cas, la garantie sur le nombre de défauts de cache total est donnée par le théorème suivant.

Théorème 4.2 (Frigo *et al.* [FS09]). *Soit un programme parallèle de travail W et tel que toute exécution d'une séquence consécutive de ces instructions $\mathcal{A}$ génère au plus $Q(\mathcal{A}) \leq f(|\mathcal{A}|)$ défauts de cache avec f une fonction concave. Une exécution parallèle ordonnancée par vol de travail avec S vols cause au plus*

$$O\Big(S \cdot f\Big(\frac{W}{S}\Big)\Big)$$

défauts de cache.

Démonstration. La preuve de ce théorème est similaire à la preuve du théorème 4.1. Il suffit de montrer que le nombre de séquences consécutives d'instructions augmente de 3 après chaque vol. Avant un vol, la victime travaille sur une séquence. Après le vol, la victime travaille sur une nouvelle séquence, le voleur travaille sur une nouvelle

séquence et la dernière séquence correspond au *join*. En majorant le nombre de défauts de cache des $O(S)$ séquences lors de l'exécution parallèle en utilisant la fonction f et en appliquant l'inégalité de Jensen, on obtient la borne annoncée. □

4.2.4 Algorithmes parallèles pour caches privés

Une autre approche pour développer des algorithmes parallèles efficaces pour caches privés consiste à concevoir directement un algorithme parallèle sans utiliser un ordonnanceur générique comme par exemple le vol de travail.

On peut citer les travaux de Arge *et al.* [AGNS08, AGS10] qui suivent cette approche. Dans [AGNS08], ils introduisent le modèle PEM pour *Parallel External Memory* qui est une extension du modèle CA présenté dans la section 2.1.1 pour une architecture parallèle à caches privés. Les algorithmes parallèles développés dans ce modèle sont donc *cache-aware*, ils utilisent les paramètres du cache.

Dans [AGNS08], Arge *et al.* donnent deux algorithmes de tri optimaux en temps et en nombre de défauts de cache dans le modèle PEM. L'un est basé sur le tri par distribution, l'autre sur le tri fusion. Dans [AGS10], ils présentent un algorithme de *list ranking* optimal dans le modèle PEM et utilisent cet algorithme pour résoudre divers problèmes sur les graphes. Dans [ASZ10], Ajwani *et al.* s'intéressent aux algorithmes géométriques et proposent en particulier un algorithme dans le modèle PEM pour la technique de *distribution sweeping*.

4.3 Ordonnancement pour un cache partagé

Nous nous intéressons maintenant à l'ordonnancement d'une application parallèle dans le cas où tous les caches sont partagés.

4.3.1 Modéliser un cache partagé

On considère une hiérarchie mémoire dans laquelle tous les cœurs sont connectés à un même cache (*cf.* figure 4.4). Il peut y avoir plusieurs niveaux de cache partagés mais il n'y a aucun cache privé.

Dans une telle architecture, il n'y a pas de problème de cohérence mémoire car aucun cœur n'a de copie privée des données. Par contre, même en se restreignant à une sous-classe de programmes parallèles comme les programmes *fully strict* dans la section précédente, on ne peut pas se placer dans un cas où les défauts de cache générés par un cœur sont indépendants de ceux des autres cœurs. Dans le cas d'un cache partagé, la dépendance entre les données chargées dans le cache par chaque cœur est plus forte : chaque accès mémoire d'un cœur peut déclencher un défaut de cache qui entraîne l'algorithme de remplacement à évincer une ligne du cache qui a pu être chargée dans le cache par n'importe quel autre cœur.

Pour définir rigoureusement une politique de remplacement sur un cache accédé par plusieurs cœurs en parallèle, il faut définir un ordre total sur les accès mémoire. Pour déterminer l'état du cache après les p accès mémoire effectués en parallèle par les p cœurs, on considère que ces p accès sont ordonnés de façon arbitraire par un adversaire. On peut donc utiliser une politique de remplacement séquentielle comme LRU ou FIF.

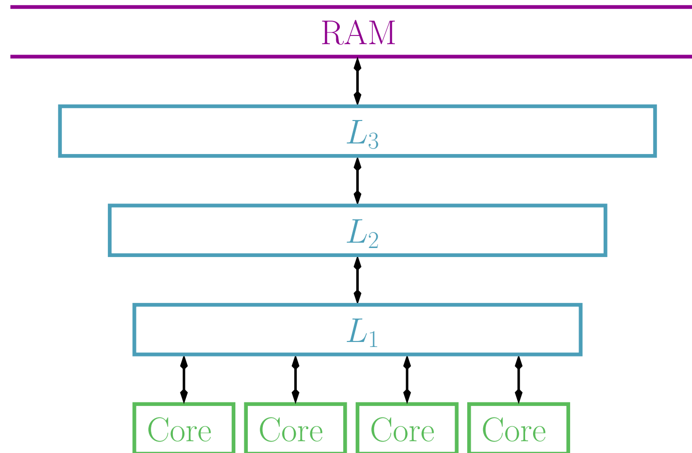


FIGURE 4.4 – Modèle d'une architecture à caches partagés

Cet ordonnancement séquentiel des accès mémoire n'est utilisé que pour calculer l'état du cache, il n'altère pas le temps de traitement. On suppose que le cache est capable de répondre à ces p accès mémoire en une unité de temps. Répondre à un accès mémoire signifie ici donner accès à la donnée si elle est dans le cache ou déclencher un défaut de cache si elle n'est pas présente. Un défaut de cache est ensuite traité comme dans le cas séquentiel.

4.3.2 Ordonnancement avec liste centralisée pour cache partagé

Nous présentons dans cette partie le travail de Blelloch *et al.* sur l'ordonnanceur pour cache partagé PDF [BGM99, BG04, CGK+07].

Contrairement à la partie précédente sur les caches privés, l'ordonnancement PDF n'est pas basé sur du vol de travail mais sur un ordonnancement de type liste centralisée comme présenté dans la partie 3.2.2. C'est un ordonnancement glouton, c'est-à-dire que dès qu'un cœur est inactif, il va chercher une tâche dans la liste et l'exécute. Les tâches activées sont placées dans la liste.

Afin de minimiser le nombre de défauts de cache supplémentaires, l'ordonnancement PDF produit une exécution parallèle qui reste proche de l'exécution séquentielle. Pour cela, chaque tâche du DAG reçoit une priorité qui correspond à sa position dans l'exécution séquentielle. Quand un processeur choisit une tâche dans la liste, il choisit la tâche ayant la priorité la plus faible.

Le théorème suivant garantit que l'exécution parallèle produite par PDF n'engendre pas de défauts de cache supplémentaires par rapport à l'exécution séquentielle si l'exécution parallèle utilise un cache légèrement plus grand. On suppose ici que les caches sont idéaux comme dans le modèle CO, c'est-à-dire gérés par la politique FIF [3].

Théorème 4.3 (Blelloch *et al.* [BG04])**.** *Soit un DAG de profondeur D dont l'exécution séquentielle génère Q_1 défauts de cache sur un cache de taille M. L'ordonnancement*

3. On pourra consulter l'article original [BG04] pour une analyse similaire pour la politique LRU. Cependant, la borne obtenue est moins bonne que dans le cas de la politique FIF.

PDF génère $Q_m \leq Q_1$ défauts de cache sur m processeurs partageant un cache de taille $M' \geq M + B \cdot (m-1) \cdot D$.

Démonstration. Comme la politique de gestion du cache est la politique optimale, il suffit de trouver une politique qui génère au plus Q_1 défauts de cache pour l'exécution parallèle pour obtenir la borne voulue. La politique proposée sépare le cache disponible en deux parties :

- une partie de taille M contient les mêmes lignes que lors de l'exécution séquentielle,
- l'autre partie de taille $B \cdot (m-1) \cdot D$ contient les lignes de cache accédées par les tâches exécutées prématurément par rapport à l'ordre séquentiel.

Un résultat obtenu dans [BGM99] permet de borner le nombre d'instructions exécutées prématurément à un instant donné par $(m-1) \cdot D$. Comme chaque instruction accède au plus une ligne de cache de taille B, la taille de cache supplémentaire nécessaire est de $B \cdot (m-1) \cdot D$. Avec une telle politique, le cache contient à tout instant les mêmes lignes que lors de l'exécution séquentielle, plus les lignes accédées par des tâches exécutées en avance par rapport à l'exécution séquentielle.

Cependant, il est possible que les tâches qui déclenchent des défauts de cache dans l'exécution parallèle ne soient pas les mêmes que celles qui déclenchent des défauts de cache dans l'exécution séquentielle. On peut montrer qu'à toute tâche causant un défaut de cache dans l'exécution parallèle mais pas dans l'exécution séquentielle, on peut associer une tâche qui cause un défaut de cache dans l'exécution séquentielle mais pas dans l'exécution parallèle. On pourra consulter l'article original [BG04] pour les détails permettant d'établir cette correspondance. □

Dans l'article [CGK+07], Chen *et al.* comparent l'ordonnancement PDF avec un ordonnancement par vol de travail sur une machine (simulée) possédant jusqu'à 32 cœurs avec un niveau de cache privé suivi d'un niveau de cache partagé sur 3 noyaux de calculs. PDF réduit le nombre de défauts de cache partagé comparé au vol de travail et obtient de meilleures accélérations (de 0% à 50%) lorsque le grain des tâches est bien réglé. En pratique, il est difficile de régler ce grain et les performances sont fortement diminuées.

L'ordonnancement PDF est efficace pour minimiser le nombre de défauts de cache sur une architecture à cache partagé mais il nécessite une liste centralisée qui peut nuire au passage à l'échelle si le grain des tâches est faible.

4.3.3 Algorithmes parallèles pour cache partagé

Quelques travaux considèrent des algorithmes spécifiquement optimisés pour une architecture à cache partagé.

Dans [CQ09], Cade *et al.* introduisent un algorithme parallèle pour un traitement de type *stencil*. Les éléments du tableau à traiter sont répartis entre les threads de manière à ce que le thread $i+1$ puissent réutiliser les données chargées en cache par le thread i. Les threads se synchronisent en utilisant une relation de type producteur-consommateur.

Dans [KMN+09], Kandemir *et al.* proposent un algorithme de parallélisation de boucles optimisé pour un cache partagé. Une première phase de l'algorithme associe à chaque itération l'ensemble des blocs de données qui seront accédés lors du calcul de l'itération. Dans une deuxième phase, les itérations sont distribuées entre les threads de manière à ce que les threads utilisent les mêmes données au même moment.

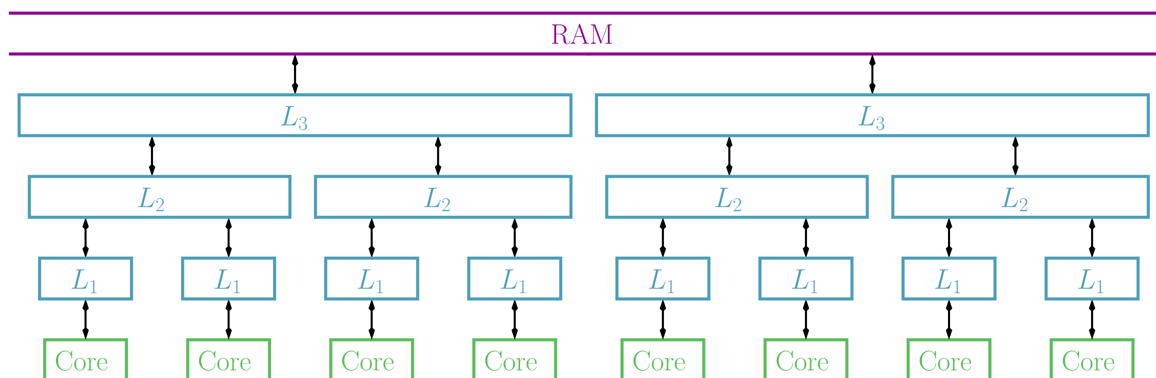


FIGURE 4.5 – Modèle *Tree-of-Caches* combinant caches privés et caches partagés

4.4 Vers un ordonnanceur pour le cas général

Nous avons vu dans les deux sections précédentes comment ordonnancer un programme parallèle dans les deux cas particuliers des caches privés et des caches partagés. Les architectures actuelles mêlent ces deux types de caches, il faut donc un ordonnanceur capable de gérer le cas général.

4.4.1 Modèle combinant caches privés et caches partagés

Le modèle *Tree-of-Caches* (ToC) combinant caches privés et caches partagés a été introduit dans [BGS09, FGBS10, Gib10]. Dans ce modèle, les cœurs sont reliés à la mémoire centrale par une hiérarchie de caches en forme d'arbre (*cf.* figure 4.5). Les feuilles de l'arbre sont les cœurs, la racine est la mémoire centrale. Chaque nœud de l'arbre représente un cache qui est partagé par tous les cœurs du sous-arbre enraciné en ce nœud. De plus, un cache de niveau i a une capacité supérieure à la somme des capacités de tous ces fils de niveau $i-1$. Ce modèle comprend comme cas particuliers les deux modèles des sections précédentes contenant exclusivement des caches privés ou des caches partagés. Il est suffisamment général pour modéliser les machines à mémoire partagée actuelles. Un modèle moins général (le premier niveau de cache est privé et le dernier niveau de cache est partagé par tous les cœurs) a été introduit dans [CSBR10].

4.4.2 Approches pour traiter le cas général

Les approches pour caches privés ou partagés des section précédentes peuvent se résumer ainsi :
- un ordonnanceur pour caches privés doit allouer aux cœurs des tâches à gros grain et accédant des données disjointes,
- alors qu'un ordonnanceur pour cache partagé doit allouer aux cœurs des tâches à grain fin et accédant les mêmes données.

Un ordonnanceur pour le modèle général doit trouver un compromis entre ces deux tendances opposées.

Dans [BCG$^+$08], Blelloch *et al.* proposent l'ordonnanceur Controlled-PDF qui permet d'obtenir de bonnes performances pour les algorithmes diviser pour régner hiérarchiques sur un modèle restreint comprenant un niveau de cache privé suivi d'un niveau de cache

partagé. Un algorithme diviser pour régner hiérarchique est un algorithme diviser pour régner dont les étapes de division et combinaison sont elles-mêmes des algorithmes diviser pour régner. Cette classe comprend par exemple la multiplication de matrice ou le tri fusion. L'ordonnanceur Controlled-PDF fonctionne en cinq étapes.

1. Toutes les tâches correspondant à un même appel récursif sur un ensemble de données de taille inférieure à la capacité du cache de niveau 1 sont regroupées en des super-tâches de type 1.
2. Toutes les super-tâches de niveau 1 correspondant à un même appel récursif sur un ensemble de données de taille inférieure à la capacité du cache de niveau 2 sont regroupées en des super-tâches de type 2.
3. Les super-tâches de type 2 sont exécutées en séquentiel en suivant l'ordre de l'exécution séquentielle.
4. Les super-tâches de type 1 appartenant à une même super-tâche de type 2 sont exécutées en parallèle sur tous les cœurs.
5. Les tâches d'une même super-tâche de type 1 sont exécutées en séquentiel sur le même cœur.

Un tel ordonnancement garantit un nombre de défauts de cache à un facteur constant de l'exécution séquentielle pour les caches privés et le cache partagé. De plus, l'accélération est linéaire en fonction du nombre de cœurs si l'algorithme diviser pour régner comprend suffisamment de parallélisme, c'est-à-dire qu'une super-tâche de niveau 2 contient suffisamment de super-tâches de niveau 1. On pourra consulter l'article original [BCG+08] pour la condition exacte.

La technique utilisée par Controlled-PDF permet d'obtenir des performances garanties pour les algorithmes diviser pour régner hiérarchiques sur un modèle de cache à deux niveaux. Cette technique a été étendue pour le modèle ToC pour des algorithmes parallèles *fork-join* mais sans garantie de performance sur le temps d'exécution [CSBR10, FGBS10]. On suppose que l'on connait pour chaque tâche la quantité de données utilisées par cette tâche et toutes ses descendantes. On alloue cette tâche et toutes ses descendantes sur un ensemble de cœurs qui partagent un cache de capacité supérieure à la quantité de données nécessaires.

A notre connaissance, il n'existe pas d'implémentation efficace en pratique de tels ordonnancements. Les ordonnancements PDF et Controlled-PDF sont centralisés et nécessitent des tâches à grain fin. Ils sont donc moins efficaces que les ordonnancements décentralisés basés sur le vol de travail [KR04, HKR04].

4.4.3 Algorithmes parallèles pour multicœurs

Différents travaux ont été publiés, présentant des algorithmes parallèles optimisés pour les caches des multicœurs. Les auteurs considèrent un modèle simplifié avec un niveau de cache privé suivi d'un niveau de cache partagé.

Dans [JMR09], Jacquelin *et al.* présentent deux algorithmes de multiplication de matrices *cache-aware* qui minimisent le nombre de défauts de cache pour des caches privés ou un cache partagé. Ils montrent comment combiner ces deux algorithmes pour minimiser le temps d'accès aux données, mesure qui prend en compte à la fois le nombre de défauts de cache pour les caches privés et le cache partagé.

4.4.4 Ordonnancements basés sur l'affinité

Une autre approche pour minimiser le nombre de défauts de cache d'un algorithme parallèle consiste à enrichir la description du programme avec des informations d'affinité. On dit qu'une tâche a de l'affinité pour un cœur si le cache associé à ce cœur contient des données qui seront accédées par cette tâche. Ces informations d'affinité peuvent être données par le programmeur ou déduites du placement des données en mémoire ou encore d'une exécution précédente du même calcul.

Dans [ABB02], Acar *et al.* proposent une modification de l'ordonnancement par vol de travail pour gérer ces informations d'affinité. En plus de la liste de tâches prêtes utilisée dans le vol de travail standard, on associe à chaque cœur une deuxième liste de tâches appelée *boîte aux lettres*. Cette liste supplémentaire contient des pointeurs vers les tâches qui ont de l'affinité pour ce cœur. A chaque activation de tâche, un pointeur est placé dans la *boîte aux lettres* du cœur qui a de l'affinité pour cette tâche. Lorsqu'un cœur devient inactif, il va essayer de voler en priorité les tâches pour lesquelles il a de l'affinité en suivant les pointeurs de sa *boîte aux lettres*.

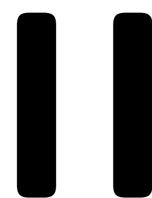
Un technique similaire est implémentée dans TBB [RVK08] sous le nom d'*affinity partitioner*. Utiliser un *affinity partitioner* lors de l'exécution de plusieurs boucles parallèles sur un même ensemble d'itérations favorise l'exécution de la même itération sur le même cœur. Cela permet de diminuer les défauts de cache lorsqu'une itération accède aux mêmes données pour chaque boucle parallèle.

Dans [HRF09, HRF$^+$10], Hermann *et al.* proposent un mécanisme de gestion de l'affinité pour l'exécution d'un graphe de flot de données représentant une simulation physique de plusieurs objets. Ce mécanisme est implémenté dans le logiciel KAAPI qui utilise un ordonnancement basé sur le vol de travail. La gestion de l'affinité intervient à trois niveaux.

- Les tâches associées au même objet physique sont ordonnancées sur le même cœur.
- Lorsque le même calcul est itéré plusieurs fois, l'ensemble de tâches associés à un objet est initialement placé sur le cœur qui a exécuté ces tâches lors de l'itération précédente. Le mécanisme de vol de tâches est ensuite libre de voler ces tâches pour assurer l'équilibrage de charge.
- Lors d'un vol, un cœur inactif vole en priorité des tâches correspondant à des objets qui interagissent dans la simulation avec ceux qu'il a déjà exécutés.

Dans [HS02b], Hendler *et al.* proposent une structure de données concurrente permettant de placer une tâche nouvellement activée directement sur le cœur pour lequel elle a de l'affinité. Ce mécanisme respecte mieux l'affinité que la méthode de Acar *et al.* [ABB02] qui ne récupère une tâche avec de l'affinité qu'au moment du vol.

Les ordonnancements basés sur l'affinité ne permettent pas d'obtenir en général de garanties de performances sur le nombre de défauts de cache. En effet, ils ne garantissent pas qu'une tâche va bien s'exécuter sur le cœur pour lequel elle a de l'affinité. Cependant, ils permettent d'obtenir de bonnes performances en pratique. De plus, ils peuvent s'intégrer dans un ordonnancement à base de vol de travail avec un faible surcoût.



Contributions

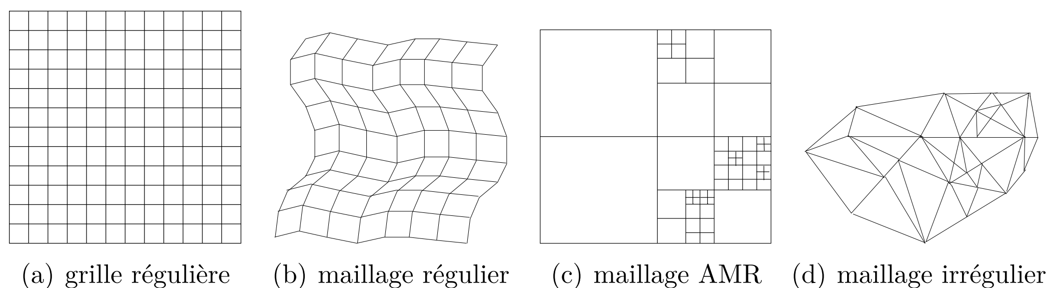
Maillage cache-oblivious pour la visualisation scientifique 5

Sommaire

5.1 Résumé des contributions 78
5.2 Discussion et perspectives 80
5.3 Binary Mesh Partitioning for Cache-Efficient Visualization 81
1 *Introduction* 81
2 *Related Work* 82
3 *Framework* 83
4 *Overlap Graphs Partitioning* 86
5 *Recursive Mesh Layout* 86
6 *Experiments* 89
7 *Conclusion* 93
References 94

Les filtres de visualisation scientifique tels que l'extraction d'isosurface ou le rendu volumique sont très gourmands en accès mémoires et nécessitent en général peu de calculs par donnée accédée. Ils sont souvent limités par les accès mémoires (au sens de la définition de la section 1.2.1). Améliorer la localité des accès aux données est donc une bonne manière d'accélérer l'exécution de ces filtres.

La structure de données principalement utilisée par un filtre de visualisation est le maillage. Il contient les points, les cellules et les données à visualiser associées à chaque point et à chaque cellule. Les points et les cellules ne sont pas toujours stockés explicitement dans la structure mais peuvent être retrouvés par un calcul simple lorsque le maillage est régulier. On peut distinguer trois types de maillages (*cf.* figure 5.1) :



(a) grille régulière (b) maillage régulier (c) maillage AMR (d) maillage irrégulier

FIGURE 5.1 – Quatre types de maillages

Grille régulière : découpage de l'espace régulier, points et cellules implicites ;

Maillage régulier : déformation géométrique d'une grille régulière, les points sont stockées explicitement mais les cellules sont implicites ;

Maillage irrégulier : le cas le plus général, les points et les cellules sont stockés explicitement.

On peut aussi distinguer les maillages AMR (*Adaptive Mesh Refinement*) qui sont réguliers mais avec une structure hiérarchique.

La plupart des accès mémoires d'un filtre de visualisation scientifique concernent le maillage. Nous nous intéresserons dans ce chapitre à l'optimisation des accès mémoires sur un maillage en réorganisant la façon dont les données sont stockées en mémoire. Pour cela, nous utilisons les techniques d'organisation mémoire CO présentées dans la section 2.3.1 :

- les organisations mémoires récursives,
- les organisations mémoires utilisant les *space filling curves.*

Le cas régulier (et par extension le cas AMR) peut être traité avec des *space filling curves* [PF01]. Le cas irrégulier est plus difficile. Dans [YLPM05], Yoon *et al.* proposent un algorithme de réorganisation de maillage irrégulier appelé OpenCCL. Cet algorithme donne de bons résultats expérimentaux mais sans garanties théoriques.

5.1 Résumé des contributions

Dans l'article *Binary Mesh Partitioning for Cache-Efficient Visualization* [TDR10a], nous proposons une alternative à OpenCCL appelée FastCOL qui permet d'obtenir un niveau de performance équivalent mais avec une garantie de performance. Nous résumons ici les contributions de cet article.

A partir d'un examen des filtres de la bibliothèque VTK [SML04], nous distinguons trois types de schéma d'accès globaux qui décrivent dans quel ordre sont parcourus les points et les cellules d'un maillage (*layout order*, *connectivity* et *external data structure traversals*) et deux types de schéma d'accès locaux qui décrivent les données accédées en chaque point et chaque cellule (*neighborhood* et *attributes operations*). A partir de cette classification, nous proposons une stratégie de réorganisation des points et des cellules du maillage qui vise à optimiser la localité des accès mémoires pour les schémas considérés (sauf pour le schéma *external data structure traversal*). L'idée est simple : stocker à proximité dans la mémoire les points et les cellules proches dans le maillage. La notion de proximité dans le maillage n'est pas géométrique mais topologique (relations de voisinage entre cellules).

Basé sur cette idée, l'algorithme FastCOL se déroule en deux étapes :

1. découper le maillage en deux récursivement pour obtenir un partitionnement sous forme d'arbre BSP (*Binary Space Partitioning*),
2. stocker les feuilles de l'arbre linéairement dans la mémoire.

Les points et les cellules correspondant à une même feuille de l'arbre sont proches dans le maillage et sont stockées proche dans la mémoire. Comme cette localité est hiérarchique, le maillage obtenu est *cache-oblivious*. Le séparateur utilisé cherche à minimiser le nombre de cellules découpées à chaque étape [MTTV98]. En effet, une

cellule découpée par un séparateur verra ses points éloignés dans les feuilles de l'arbre et donc dans la mémoire ce qui diminue la localité. Ce séparateur donne une garantie sur le nombre de cellules découpées. On peut déduire de cette borne une garantie sur le nombre de défauts de cache lorsque le schéma d'accès du filtre respecte la localité définie par la topologie du maillage. Pour un maillage de taille S sur un cache de taille M avec des lignes de cache de taille B, le nombre de défauts de cache est

$$\frac{S}{B} + O\Big(\frac{S}{M^{1/3}}\Big).$$

Le premier terme représente le nombre de défauts de cache nécessaire pour lire entièrement le maillage avec une localité spatiale parfaite. Le deuxième terme représente le surcoût dû à la localité temporelle qui ne peut être complètement optimisée. On peut remarquer que ce surcoût diminue quand la taille du cache augmente. Enfin, un séparateur étant obtenu en temps linéaire, le temps de calcul nécessaire pour FastCOL est de $(n \log n)$ avec n le nombre de points et de cellules du maillage. Ce temps est raisonnable sachant que n est en général très gros.

A travers une campagne d'expériences approfondies, nous avons comparé les performances du maillage original, du maillage réorganisé avec un tri géométrique, du maillage réorganisé avec OpenCCL et du maillage réorganisé avec notre algorithme FastCOL sur 10 filtres de visualisation et sur 50 maillages de tailles différentes. La plupart de ces filtres proviennent de VTK et ne sont pas modifiés. Nous avons mesuré à la fois le gain en temps comparé à l'organisation originale et la réduction du nombre de défauts de cache sur deux types de processeurs différents. Les maillages *cache-oblivious* OpenCCL et FastCOL ont des performances bien supérieures aux maillages originaux et géométriques. FastCOL et OpenCCL ont des performances comparables, OpenCCL étant légèrement meilleur. Cependant la réorganisation est bien plus rapide avec FastCOL et nécessite beaucoup moins d'espace mémoire. Nous avons aussi comparé ces maillages sur deux codes d'extraction d'isosurface sur GPU. Les maillages *cache-oblivious* améliorent nettement les performances grâce à un plus grand nombre d'accès mémoires fusionnés (*coalesced memory access*). C'est à notre connaissance la première fois que l'avantage des algorithmes *cache-oblivious* est mis en valeur sur GPU.

Nous proposons également une mesure de la qualité d'une organisation de maillage basée sur les longueurs d'arêtes. On dit qu'une arête reliant les points d'indices i et j a une longueur $|j - i|$ qui est proportionnelle à la distance en mémoire entre ces deux points. Un calcul de corrélation entre cette mesure et le nombre de défauts de cache observé expérimentalement montre que les longueurs d'arêtes mesurent bien la qualité de l'organisation mémoire. De plus, nous remarquons que les longueurs d'arêtes ne représentent pas complètement la qualité de l'organisation mémoire mais qu'il faut aussi prendre en compte la dispersion de ces arêtes.

Enfin, nous comparons les performances de FastCOL avec une approche basée sur les *space filling curves*. Pour adapter une *space filling curve* à un maillage irrégulier, on construit d'abord un *kd-tree* puis on organise les feuilles de l'arbre suivant l'ordre donné par la *space filling curve*. Cette approche donne des performances presque aussi bonnes que FastCOL sur la plupart des maillages de notre jeu de test. Cependant elle ne donne aucune garantie de performance. Le séparateur utilisé par le *kd-tree* est purement géométrique et ne donne aucune garantie sur la découpe du maillage. Dans le cas où

cette découpe se fait sur des zones à forte connectivité, les performances peuvent être médiocres comme nous le montrons sur un exemple.

5.2 Discussion et perspectives

Au niveau de l'analyse théorique, cet article offre principalement deux perspectives. La garantie sur le nombre de défauts de cache nécessite une condition sur le schéma d'accès mémoire du filtre qui n'est pas indépendante de l'organisation mémoire. Il serait intéressant de pouvoir garantir les performances d'un filtre en utilisant des conditions intrinsèques au maillage sans faire référence à son organisation mémoire. On peut distinguer deux types d'accès globaux qui ne font référence qu'à des caractéristiques intrinsèques du maillage :

Thin access l'ordre d'accès aux éléments du maillage suit les relations de voisinage,

Fat access k éléments du maillage accédés consécutivement forment un volume compact, c'est-à-dire le voisinage de ces k éléments est de taille $O(k^{2/3})$.

Il serait aussi intéressant d'établir une borne inférieure sur le nombre de défauts de cache de la meilleure organisation mémoire. Nous conjecturons que le nombre de défauts de cache est au moins de

$$\frac{S}{B} + \Omega\left(\frac{S}{B \cdot M^{1/3}}\right).$$

Au niveau des performances en pratique des réorganisations mémoires, les expériences réalisées dans cet article montrent que l'approche par *kd-tree* fonctionne bien dans la plupart des cas. Elle ne donne pas de garanties théoriques mais les maillages sur lesquelles elle fonctionne moins bien sont très spécifiques et ne devrait pas se retrouver en pratique. De plus cette approche est plus simple à mettre en place et la réorganisation est moins coûteuse. Il est à noter que lorsque le maillage est déjà décomposé en sous domaines suite à une simulation sur un supercalculateur par exemple, il est en général possible de réutiliser la même découpe pour l'organisation mémoire. En effet, les décompositions de domaine pour le calcul en mémoire distribuée minimisent le volume de communication nécessaire entre les processeurs et donc la surface de contact entre les sous domaines. Limiter la surface de contact étant bon pour le cache, on pourra se contenter de réorganiser chaque sous domaine séparément.

Nous avons également étudié des organisations mémoires spécifique aux GPUs qui maximisent le nombre d'accès mémoires fusionnés. Cependant les organisations mémoires proposées n'ont pas obtenu de meilleures performances que FastCOL. Ce travail a été mené lors du stage réalisé par Siméon Marijon.

Binary Mesh Partitioning for Cache-Efficient Visualization

Marc Tchiboukdjian, Vincent Danjean and Bruno Raffin

Abstract—One important bottleneck when visualizing large data sets is the data transfer between processor and memory. Cache-aware (CA) and cache-oblivious (CO) algorithms take into consideration the memory hierarchy to design cache efficient algorithms. CO approaches have the advantage to adapt to unknown and varying memory hierarchies. Recent CA and CO algorithms developed for 3D mesh layouts significantly improve performance of previous approaches, but they lack of theoretical performance guarantees. We present in this paper a $O(N \log N)$ algorithm to compute a CO layout for unstructured but well shaped meshes. We prove that a coherent traversal of a N-size mesh in dimension d induces less than $N/B + O(N/M^{1/d})$ cache-misses where B and M are the block size and the cache size, respectively. Experiments show that our layout computation is faster and significantly less memory consuming than the best known CO algorithm. Performance is comparable to this algorithm for classical visualization algorithm access patterns, or better when the BSP tree produced while computing the layout is used as an acceleration data structure adjusted to the layout. We also show that cache oblivious approaches lead to significant performance increases on recent GPU architectures.

Index Terms—Cache-aware, cache-oblivious, mesh layouts, data locality, unstructured mesh, isosurface extraction.

1 Introduction

Many visualization related processing steps, like isosurface extraction, rely on read-only and memory intensive algorithms. Adequately combining data layout and access patterns can significantly improve performance. Since classical processor architectures cache blocks of adjacent data, storing data accessed consecutively nearby in memory enables to reduce cache-misses. Enforcing locality is also relevant for some GPU architectures that coalesce parallel memory accesses to save clock cycles when the target data are close in memory. For instance the Nvidia G80 and G200 [1] can coalesce concurrent threads data accesses, while the Intel Larrabee supports vector level coalesced loads and stores for its VPUs.

For regular data structures, data layouts based on space filling curves, like the Z curve, are common [2]. They provide a cache-efficient layout for access patterns showing a strong spatial locality. For irregular data structures, computing cache-efficient layouts is significantly more difficult.

We can distinguish two classes of cache-efficient algorithms: Cache-Aware (CA) and Cache-Oblivious (CO) algorithms. CA algorithms are based on the external-memory (EM) model [3]. The memory hierarchy consists of two levels, a main memory of size M called cache and an infinite size secondary memory. The data are transferred between these two levels in blocks of B consecutive elements. CA algorithms can be very efficient but require the layouts to be recomputed

- *Marc Tchiboukdjian is with CNRS and CEA/DAM,DIF, E-mail: Marc.Tchiboukdjian@imag.fr.*
- *Vincent Danjean is with Grenoble Universités and LIG (UMR 5217), E-mail: Vincent.Danjean@imag.fr.*
- *Bruno Raffin is with INRIA and LIG (UMR 5217), E-mail: Bruno.Raffin@imag.fr.*

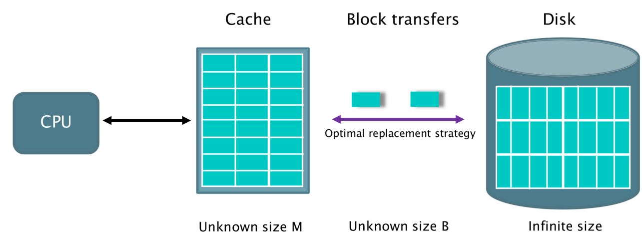


Fig. 1. The cache-oblivious memory model. The data are transfered by block of B consecutive elements into a cache of size M. Both parameters are unknown to the algorithm.

for each memory architecture. It makes it difficult to efficiently share the same layout between heterogeneous processing units mixing CPUs and GPUs for instance. CO approaches [4] intend to overcome this limitation by proposing layouts that are independent from the cache size M and the block size B (Fig. 1). The Z curve is an example of a CO layout (Fig. 2). For irregular data structures, the most significant and recent work is probably the CO mesh layout proposed by Yoon *et al.* [5] (OpenCCL algorithm). In comparaison to other layouts, experiments show speedups ranging from 2 for in-core computations, up to 20 for out-of-core computations. This algorithm is experimentaly efficient for a wide range of meshes. However this algorithm is based on heuristics, without theoretical performance guarantees, neither on the layout computation complexity nor on the quality of the resulting layout.

In this paper we introduce a new CO layout algorithm for irregular but well shaped meshes with a theoretical performance guarantee. It relies on a recursive mesh partitioning using a specific BSP (Binary Space Partitioning) algorithm introduced by Miller *et al.* [6]. This algorithm cuts the mesh

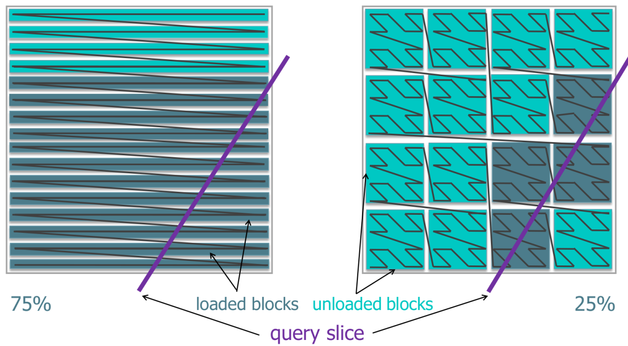


Fig. 2. Good layouts can significantly reduce the number of block transfers. On the left, 75% of the data must be loaded to access the queried slice (each line corresponds to a cache line), while the CO layout used on the right (Z curve) enables to reduce this amount to only 25% of the data (each block fits into a cache line).

guaranteeing a good tradeoff between minimizing the number of cut elements and having two partitions of similar size. When applied recursively it ensures that spatially close and strongly connected data tend to be partitioned deeper in the BSP tree. The CO layout is obtained by storing the data linearly in memory from the first leaf of the BSP tree to the last one. The data loaded in a cache block are thus contiguous leaves of the BSP tree. It is cache oblivious as to any block and cache size corresponds a BSP tree depth level ensuring a strong locality and connectivity.

Classical BSP algorithms or space filling curves could be used in a similar way for building layouts. But these space partitioning techniques only take into account geometric information and not connectivity. Performance is not guaranteed.

Our CO layout algorithm has several benefits. The layout computation has a $O(N\log N)$ complexity. It also guarantees that a coherent traversal of a N-size mesh in dimension d induces less than $N/B + O(N/M^{1/d})$ cache-misses where B and M are the block and cache size. Experiments show that the layout computation is about two to three times faster than for the OpenCCL algorithm while requiring significantly less memory (only 2% of the memory used by OpenCCL on the biggest meshes). At execution, perfomance is comparable with the OpenCCL algorithm for classical access patterns. The BSP tree computed for the layout can also be used as an internal, layout consistent, acceleration data structure. Experiments reveal that using it as a min-max tree for accelerating an isosurface extraction brings significant additional performance improvements (from 12% to 55% for in-core computations) compared to using an external kd-tree not necessarily consistent with the layout.

We also show that CO layouts can lead to significant performance improvements on recent NVIDIA GPUs (speedups ranging from 1.52 to 4.09), even if no cache mechanism is involved. Because CO algorithms enforce data locality, they favor coalesced data accesses. To our knowledge, this is the first time such benefits of CO layouts on GPUs are highlighted.

Related work is discussed in section 2. We introduce our

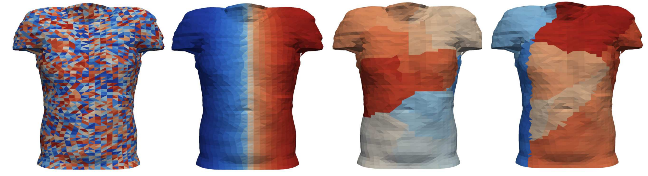


Fig. 3. Visual illustration of different layouts for the torso mesh. Successive cells in memory are colored from blue to red. From left to right, the original, geometric (sorted by x,y and z coordinates), OpenCCL (cache-coherent layout from [5]) and FastCOL (our approach) layouts. For spatially close tetrahedra, color discrepancy decreases from the left to right layouts. It denotes an improved memory locality, less likely to generate cache-misses for spatially coherent access patterns.

framework and review common mesh access patterns in section 3. Overlap graphs, the class of meshes our algorithm applies to, and the graph separator algorithm are presented in section 4. The CO algorithm and its implementation are described in section 5. Experimental results are presented in section 6 before the conclusions.

2 Related Work

2.1 Cache-Efficient Algorithms

Today, many algorithms have their CA or CO versions [7] where computations and data are reordered for an efficient cache use. A widely used technique is blocking or tiling: elements are mapped in memory and accessed by blocks of size B to fit in a cache line. For instance, regular search trees are made CA by grouping B keys in a single node. Such trees are called B-trees. Another example is matrix multiplication. Instead of rows or columns, the ATLAS library [8] traverses the matrix by blocks such that all involved blocks for a partial computation fit in the cache.

It is often possible to obtain the same result while being oblivious to the block size. For example, by carefully storing a binary tree in memory with the vam Emde Boas layout [7], a search can match the I/O bound of the CA B-tree. In this layout, the tree is divided at the middle of its height into a top tree and several bottom trees. The same layout is recursively applied to each of these trees, which are then stored sequentially in memory. The same recursive blocking technique is applied within the divide and conquer matrix multiplication. Indeed, this is a CO algorithm [7] with the same theoretical complexity as its CA counterpart.

Another CO alternative to blocking is the use of space filling curves. These layouts have been used efficiently for regular mesh traversals [2] and matrix multiplication algorithms [9].

CO algorithms for regular structures are not always as efficient as their CA counterparts. The access pattern to a CO layout is more complex, leading to a significant overhead that limits the benefit of a CO approach. For instance, the CO matrix multiplication is not competitive with the CA version [10]. We do not face this problem here as an unstructured mesh has already a complex access pattern.

2.2 Mesh Layouts

The problem of reordering mesh elements for efficient cache use was first encountered when a vertex cache was introduced in graphic cards. To maximize the efficiency of the hardware vertex cache, triangles needed to be reordered before being sent to the graphic card. The algorithm developed by [11] reorders triangles to form triangle strips. They assume the cache has a FIFO policy and the cache size is known to the algorithm. Algorithms not based on strips that work for all cache sizes have been introduced in [12]. The layout quality has been improved in [13], and the overdraw rate reduced in [14]. When the geometry and the topology of the mesh can be modified, the method of [15] generates a single strip representing all the mesh while improving the efficiency of back-face culling. In [16], the authors propose a mesh compression scheme that is also cache efficient. However, as they target graphic cards, all these approaches only reorder mesh cells and not points. They consider the graphics card cache model (no cache line, independent vertices fetching, etc.) which is very different from the CPU cache models. Only the temporal locality on mesh points is taken into account and not the spatial locality. Moreover, the application must access the mesh in the exact same order as given by the cell layout (especially for triangle strips). Finally, work in this area mostly deals with surface meshes.

Processing sequences [17] reorder the points and the cells of a mesh, but this approach focuses on streaming computations. The goal is to minimize the maximum amount of memory used during the computation. The application should again follow the mesh layout.

OpenCCL [18] presented in [5] casts the mesh layout problem as a graph optimization problem. To describe the access pattern of the application using the mesh, the user must provide a graph where vertices represent data and edges link data that are likely to be accessed in sequence at runtime. A good mesh layout is a permutation of the graph vertices that results in a more efficient layout of the mesh in memory. They developed a local metric to decide if a swap of some vertices improves the layout. They optimize this measure thanks to a multilevel optimization scheme. In [19], two global cache-oblivious metrics are introduced to quantify the quality of a mesh layout. These two metrics involve edge lengths. If two mesh elements i and j likely to be accessed sequentially are stored in the layout at position x_i and x_j then the edge length l_{ij} is $|x_i - x_j|$. The first metric proposed (COM_a) is the arithmetic mean of edge lengths and the second (COM_g) is the geometric mean of edge lengths. While both metrics yield a good correlation with measured cache misses, COM_g seems to perform better. All previously proposed mesh layout optimization algorithms [5], [19] are based on heuristics. No bound on the quality of the layout, i.e. number of cache-misses, is provided. Our algorithm, called FastCOL, guarantees an upper bound on the number of cache-misses for the class of meshes it applies to. This bound is closely related to edge lengths, like the COM_a and COM_g metrics introduced in [19]. FastCOL is based on the mesh geometry, a data often available for the meshes considered in scientific visualization. OpenCCL is more general on that aspect as it only uses the mesh topology and thus can be applied to graphs not embedded in space.

Aforementioned approaches mainly focus on optimizing mesh layouts when the application accesses the mesh without the help of any additional data structure. That is, the application only traverses the mesh with the help of the cells and points arrays or with the cell-to-points or point-to-cells pointers of the mesh. Another approach [20] optimizes the layout for applications accessing the mesh through bounding volume hierarchies (BVH) trees. To generate an efficient layout, they use the OpenCCL algorithm and provide two kind of links in the access graph: links representing spatial locality in the mesh and links representing parent-child locality in the BVH tree. Our algorithm also handles these two kinds of locality. During the layout computation, we build a BSP tree that is used to reorder the mesh. This tree is tailored to efficiently use our mesh layout. Contrary to the approach in [20] where the layout algorithm takes a mesh and a BVH tree as input to produce a layout, we only take the mesh as input and produce both a layout and a BSP tree consistent with this layout. This BSP tree can be used as an acceleration structure, for isosurface extraction for instance.

2.3 Isosurface Extraction

The marching tetrahedra algorithm can be accelerated with various data structures allowing to efficiently search for the cells intersected by the isosurface. One such data structure is the min-max tree [21]. An octree where each node stores the minimum and maximum values of its subtree permits to quickly discard parts of the mesh that do not contain any intersected cell. The search is thus improved from $O(n)$ to $O(k + k \log n/k)$ where n is the number of cells and k the size of the isosurface (usually $k \ll n$). If the scalar field is spatially coherent, the performance is actually improved on the theoretical bound.

An optimal data structure for this problem is the interval tree storing for each cell c the interval whose extremes are the minimum and maximum value of the points of c [22]. The query time is improved to $O(\log n + k)$ whatever the spatial repartition of the scalar field is. The interval tree has been made I/O-efficient allowing a query with complexity $O(\log_B n + k/B)$, where B is the block size. This bound is optimal [23]. However this approach is not space-efficient since the vertex information is duplicated many times. The 2-level indexing scheme based on the meta-cells technique introduced in [24], [25] is both practical and space-efficient as there is no duplicated information. Spatially close cells are grouped into meta-cells, which are then used in the I/O-efficient interval tree.

The approach we developed with the consistent BSP tree is a CO alternative to the meta-cells technique but we use the min-max tree instead of the interval tree, which may not be as efficient on a scalar field with high spatial variations.

3 Framework

3.1 Common Mesh Access Patterns

A mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates,

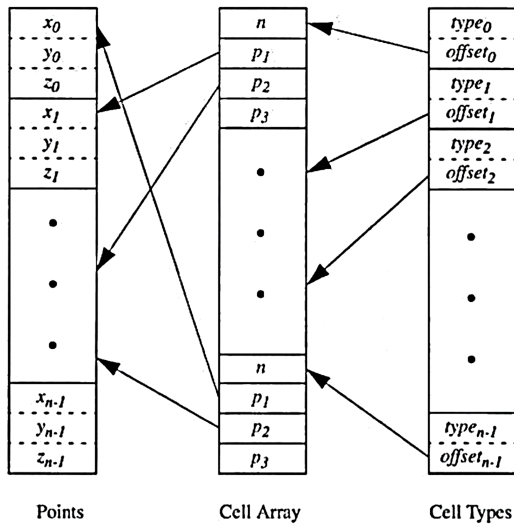


Fig. 4. The vtkUnstructuredGrid data structure (from the VTK Textbook [26]). The Points array contains points coordinates and the Cells array contains the indices of cell points. The Cell Types array contains the nature of each cell and provides $O(1)$ random access to cells.

scalar values, etc.) and an array storing for each cell its points and attributes (e.g. the nature of the cell, scalar values, etc.). When the mesh is composed of cells of different nature (using various number of points), an additional array allows random access to cells (Fig. 4). As many visualization filters also need to access neighbors of a point or a cell (e.g. the gradient filter), additional structures storing the connectivity permit efficient access to point and cell neighbors. Finally, accelerating structures can be used to efficiently select cells or points verifying a certain property (e.g. select cells intersecting an isosurface).

A mesh can be traversed using the following strategies:

- **layout order traversal:** traverse all points or all cells in the order given by the corresponding array (e.g. the marching tetrahedra algorithm);
- **connectivity traversal:** traverse all points or all cells using the connectivity information (e.g. the ray casting algorithm or the VTK connectivity filter);
- **data structure traversal:** traverse points or cells in the order given by an external data structure (e.g. isosurface extraction with a min-max tree).

While traversing the mesh, several local operations are commonly used to process a mesh element:

- **neighborhood operation:** get all points/cells connected to the current point/cell (e.g. the VTK gradient filter);
- **attributes operation:** get attributes from points composing the current cell or get attributes from cells using the current point (e.g. marching cube).

Multiple local operations can be used at the same time.

3.2 Layout Influence on Cache Performance

The cache performance of visualization filters is greatly influenced by the underlying mesh data structure and specifically the indices of points and cells: the mesh layout. These indices can be modified without affecting the intrinsic characteristics of the mesh like the geometry or the topology and without any modification on the visualization filters. Depending on the access pattern, the layout can impact the cache performance in various ways.

- Cache performance is optimal with **layout order traversals** as they lead to sequential memory accesses.
- A layout improves cache performance of **connectivity traversals** and **neighborhood operations** if the elements that are connected in the mesh topology are stored nearby. Spatial locality is increased as two consecutive cells in the traversal could be in the same memory block. It also enhances temporal locality since the current cell in the traversal has a good probability to still be in cache if its memory block has been accessed recently.
- A layout improves cache performance of **data structure traversals** if the elements that are accessed consecutively by the data structure are stored nearby. Two consecutive elements could be in the same memory block, increasing spatial locality.
- A layout improves cache performances of **attributes operations** if points corresponding to a same cell (or cells using a common point) are stored nearby. Points may share memory blocks, which increases spatial locality.

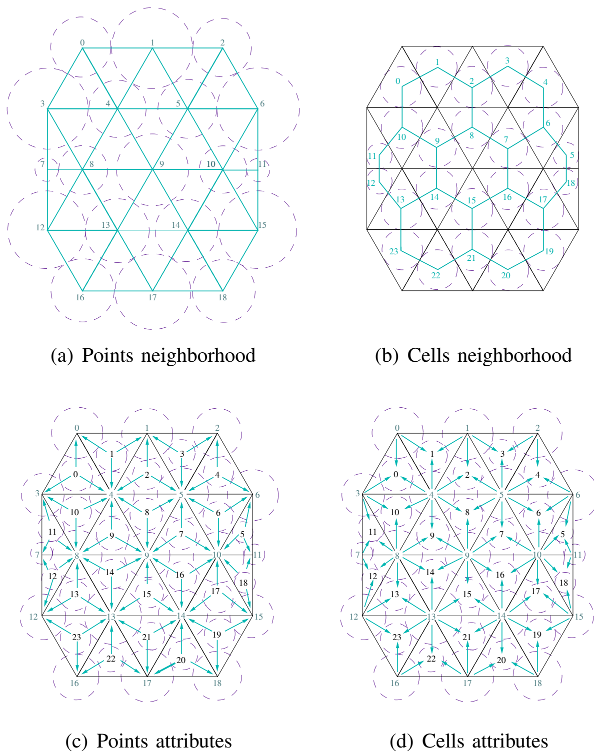
A layout order traversal with attributes operations can be further optimized if the layout stores nearby the cells that share common points. The memory blocks containing the common points have a higher chance to still be in cache, which enhances temporal locality. The marching tetrahedra algorithm is an example of such access pattern.

3.3 The Access Graph Model

To optimize a layout for a specific access pattern, we need to model the data accesses. As in [5], we use a graph $G=(V,E)$ where vertices are mesh elements (point or cell) and edges represent consecutive data accesses. However we constrain the topology of the access graph to forbid edges between elements that are 'far' from each other, as detailed in section 3.5.

We now model a visualization filter as a function f applied once to each element of the mesh. As we are interested in the cache performance, we do not consider the processing part of f. We restrict its memory accesses so that they are compatible with the access graph: only the neighbors of the element i in the access graph can be accessed to compute $f(i)$.

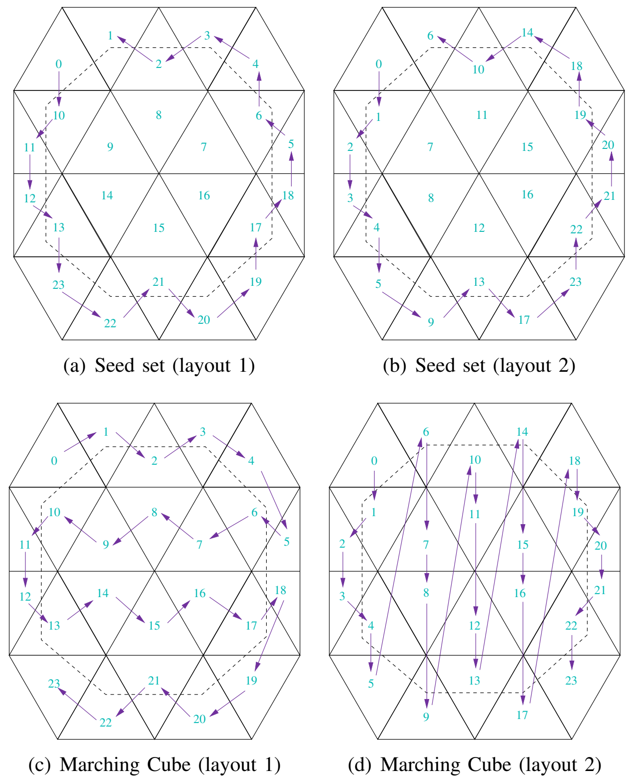
Figure 5 presents examples of access graphs for the neighborhood and attributes operations. A point neighborhood operation is represented in 5(a). This kind of local access scheme is used by the VTK gradient filter. To compute the gradient value at a point, values of the scalar field at the neighborhood points are needed, thus edges link neighbor points in the access graph. Likewise, a point attributes operation is represented in 5(c). The marching cube algorithm is an example of such a scheme. Indeed, to compute the isosurface within a cell, the coordinates and scalar value for each point composing the cell are needed. Therefore, edges link each cell to its points in the access graph.



(a) Points neighborhood (b) Cells neighborhood

(c) Points attributes (d) Cells attributes

Fig. 5. Example of access graphs for the neighborhood and attributes operations defined in section 3.1. Numbers represent points (a), cells (b) or points and cells (c,d) indices in the layout. Data accesses are represented in blue: neighbor points in (a), neighbor cells in (b), points of each cell in (c) and cells of each point in (d). Dashed circles show that all these graphs are overlap graphs (cf. section 4.1).



(a) Seed set (layout 1) (b) Seed set (layout 2)

(c) Marching Cube (layout 1) (d) Marching Cube (layout 2)

Fig. 6. Example of access patterns for two isosurface extraction algorithms: Marching Cube (layout dependent) and Seed Set (layout independent). Arrows represent the access pattern followed to extract the isosurface (dashed line). Numbers represent cell indices in the layout.

3.4 Access Graph for Layout Order Traversals

Visualization filters do not always rely on intrinsic mesh characteristics such as topology or geometry when accessing the mesh. They sometimes rely on the layout itself. For example, the Seed Set isosurface extraction algorithm processes the mesh with a connectivity traversal. The access graph does not change when the layout changes, provided that the initial seeds stay the same (Fig. 6(a) and 6(b)). On the contrary, the Marching Cube algorithm processes the mesh with a layout order traversal and thus the access graph depends on the layout (Fig. 6(c) and 6(d)).

The access graph of fig. 5(c) properly models the local operations of the marching cube algorithm. However, to optimize the global traversal strategy, the edges of the access graph of fig. 5(b) should be added instead of using the access graphs of fig. 6(c) or fig. 6(d). First, because the resulting access graph does not depend on the layout. Second, because it enables temporal locality optimization: cells that share common points should be stored nearby.

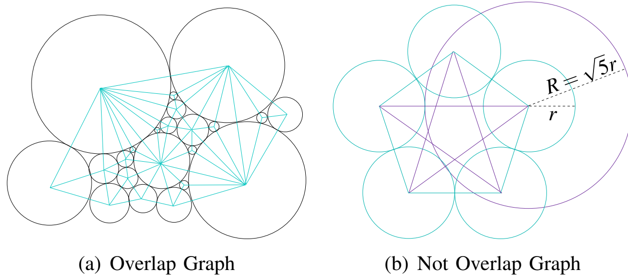
3.5 Restriction to Overlap Graphs

Access graphs can model a large range of access patterns, even ones with a weak spatial coherency where edges connect distant elements. To be able to build efficient CO layouts with a provable quality, we restrict access graphs to be overlap graphs. These graphs model spatially coherent access patterns, i.e. where edges connect spatially close elements. They are formally defined in the next section.

In contrast to OpenCCL [5], we add geometric information to the access graph and use it to constrain its topology. We add to each vertex the coordinates of the corresponding element of the mesh and we restrict the graph to be an overlap graph. This assumption forbids consecutive access of mesh elements that are too 'far' from each other. This restriction is satisfied by most visualization filters and allows us to devise an efficient separator-based algorithm with a theoretical guarantee on the quality of the mesh layout generated.

Meshes are often composed of elements that are well shaped in some sense, such as having a bounded aspect ratio or angles that are not too small or too large. Provided that the underlying mesh is constrained by such geometric features, the access graphs for connectivity traversals, neighborhood and attributes operations are overlap graphs [6] (Fig. 5). We have seen in the previous section that the layout order traversals should be handled differently depending on the visualization filter as they are based on the layout.

The only remaining mesh access pattern is data structure traversal. Unfortunately, access graphs for data structure



(a) Overlap Graph (b) Not Overlap Graph

Fig. 7. (a) Example of overlap graph for $\alpha = 1$. There is an edge (solid blue line) between two points if their corresponding circles are tangential.(b) Example of a non overlap graph for $\alpha = 1$ but overlap for $\alpha = \sqrt{5}$ (K_5 complete graph).

traversals may not always be overlap graphs. For instance, the interval tree used in the isosurface extraction of [22], does not traverse cells intersected by the isosurface in an order based on the geometry or topology of the mesh. However in this case, the access graph depends on the value of the isosurface. It is not practical to compute a layout and reorder the mesh for each isosurface extraction. We develop an alternative with our consistent BSP tree (cf. section 5.6). This tree is tailored to efficiently use our layout as the induced mesh traversal is a layout order traversal with good cache performance.

4 Overlap Graphs Partitioning

In this section, we review the work of Miller *et al.* [6] on overlap graphs, which we use to restrain the topology of the access graph.

4.1 Overlap Graphs

We associate to each vertex v_i of the access graph the coordinate p_i in $\mathbb{R}^d$ of the corresponding mesh element (point or cell). A graph is α-overlap if:

- It is possible to associate to each vertex v_i a ball B_i centered at p_i such that the two balls of any pair intersect in at most one point (Fig. 7(a)) ;
- Edges can connect two vertices only if expanding the smaller of their two balls by a factor of α make them intersect (Fig. 7(b)).

The α factor constrains the topology of the graph to follow the geometry of the mesh: two elements that are too far away from each other cannot be edge connected (Fig. 7(b)).

As detailed in section 3.5, most access patterns of visualization filters can be modeled with overlap graphs. Fig. 5 shows how balls can be added to the neighborhood and attributes access graphs so that all edges respect the overlap graph constraint.

4.2 Geometric Separator Algorithm

Given their geometrical properties, overlap graphs can be partitioned efficiently in two parts of approximately equal size, while minimizing the number of edges cut.

The following randomized algorithm introduced by Miller *et al.* [6] computes in linear time and with a high probability an optimal geometric separator (Algo. 1). It starts by randomly sampling a constant number of points V_s from the input graph. Next it projects these V_s points onto the surface of the unit sphere centered at the origin in $\mathbb{R}^{d+1}$, using a stereographic projection. It produces V_p points. Then it finds a centerpoint c of this random sample V_p in linear time relative to the sample size. A point is a centerpoint if every hyperplane passing through it divides the sample set V_p approximately evenly, at most in a ratio $d+1:1$. With good probability, this centerpoint is a centerpoint of the projection of the original set of points V [27]. Finally, we randomly choose a hyperplane $(c, \mathbf{n})$ passing through this centerpoint. This hyperplane splits the graph into two partitions, each one consisting of the points of V that project on the same side of the hyperplane in $\mathbb{R}^{d+1}$. Repeating this process and selecting the separator cutting the smallest number of edges gives a small separator with high probability.

Algorithm 1 Geometric separator algorithm

Input: Graph $G = ($Vertices V, Edges $E)$
Output: A separator ϕ
1: **repeat** n_c **times**
2: $V_s \leftarrow$ sample of $(d+3)^4$ points of V
3: $V_p \leftarrow$ project V_s to the unit sphere in $\mathbb{R}^{d+1}$
4: $c \leftarrow$ find a centerpoint of V_p
5: **repeat** n_h **times**
6: $\mathbf{n} \leftarrow$ random normal vector
7: $\phi \leftarrow$ separator defined by $(c, \mathbf{n})$
8: compute the number of edges cut by ϕ
9: **end**
10: **end**
11: **return** the best ϕ

The most time consuming part of the algorithm is the quality evaluation of the separator (Alg. 1 line 8). The other operations involve only a small number of points.

The quality of the obtained separator is guaranteed by the following theorem.

Theorem 1 (Geometric separator [6]): Let G be an n-vertex α-overlap graph in d dimensions. With high probability, the previous algorithm (Alg. 1) partitions the vertices of G into two sets A and B such that $|A|,|B| \leq \frac{d+1}{d+2}n$ and the number of edges between A and B is $O\left(\alpha n^{1-1/d}\right)$.

Such a separator is asymptotically optimal for the class of overlap graphs. Indeed we cannot find a smaller separator for a regular d dimensional grid [6].

5 Recursive Mesh Layout

Applying the separator algorithm recursively for a given overlap graph corresponding to the mesh access pattern enables us to define a CO layout. In this section we present the CO layout computation algorithm, prove its performance and discuss some implementation details.

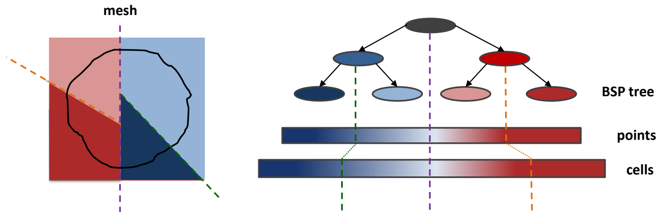


Fig. 8. Illustration of the correspondence between mesh regions, BSP tree branches and data arrays.

5.1 Mesh Layout Algorithm

The recursive application of the separator gives a BSP tree where each node is a separator (Fig. 8). Leaves of this tree correspond to small subparts of the mesh that are stored consecutively to provide the layout (Algo. 2).

Algorithm 2 Layout algorithm

```
1: function COLAYOUT(G,layout,i,j)
2:   if size(G) > 1 then
3:     FINDSEPARATOR(G,G_left,G_right)
4:     n_left ← size(G_left)
5:     COLAYOUT(G_left,layout,i,i+n_left)
6:     COLAYOUT(G_right,layout,i+n_left,j)
7:   end if
8: end function
```

Given the linear complexity of the geometric separator, our layout algorithm has a complexity of:

$$\begin{aligned} W(N) &= \max_{1/2\leq\lambda\leq\delta} [W(\lambda N) + W((1-\lambda)N)] + O(N) \\ &= O(N \log N) \end{aligned}$$

where $\delta = \frac{d+1}{d+2}$. The only requirement to obtain the claimed complexity is to have a point sampler of linear complexity and an iterator on edges of linear complexity too.

5.2 Layout Quality

The algorithm 2 generates a BSP tree that can be used to create a partition of the access graph such that each subpart fits in cache. When processing the mesh, the edges of the access graph linking elements in the same subpart do not generate extra cache misses as the whole subpart fits in cache. Conversely, edges of the access graph linking elements in different subparts (cut edges) may generate extra cache misses. Because the number of such cut edges is known, we can exhibit an upper bound on the number of cache misses for the layout (Th. 3). The number of cut edges is equal to the total number of edges cut by all separators down to the largest nodes fitting in cache (Lm. 2 and Fig. 9).

Lemma 2 (Cut Edges): Let G be a N-vertex α-overlap graph in d dimensions. Let T be the BSP tree obtained by recursively applying the geometric separator. Let T_m be the tree corresponding to T after removing all nodes that have a father node with less than m vertices. The leaves of T_m verify $size(father(x)) > m$ and $size(x) \leq m$ (Fig. 9). The total number of edges cut by all separators in T_m is bounded by $k_m = O\left(\frac{N}{m^{1/d}}\right)$.

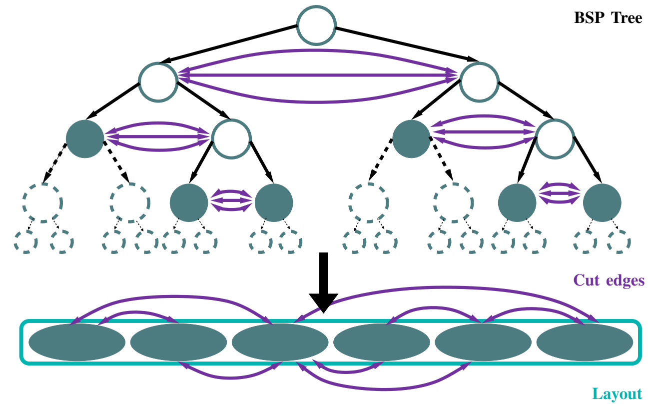


Fig. 9. A full tree generated by the algorithm 2. The sub-tree (solid lines) represents T_m and the purple arrows the edges cut for this sub-tree. The leaves of T_m (green filled) all have less than m vertices. Below, the green ellipses identify the leaves of T_m in the layout.

Proof: We sum the number of edges cut by all separators from the root of T_m to its leaves. The separator theorem (Th. 1) ensures that the number of cut edges is less than $\alpha c r^{1-1/d}$ for a r-vertex graph (c is a constant). It provides two subgraphs of size λr and $(1-\lambda)r$ with $1/2 \leq \lambda \leq \frac{d+1}{d+2}$.

The total number of cut edges in a subtree rooted at a node $v \in T_m$ representing an r-vertex graph is thus:

$$K(r) \leq \max_{1/2\leq\lambda\leq\frac{d+1}{d+2}} [K(\lambda r) + K((1-\lambda)r)] + c\alpha r^{1-1/d}.$$

The $K(\lambda r)$ and $K((1-\lambda r))$ terms are due to the edges cut by all the separators in the left and right subtrees. The $c\alpha r^{1-1/d}$ term corresponds to the edges cut by the separator of node v. By induction on r, we show that

$$K(r) \leq c'\left(\frac{r}{m^{1/d}} - r^{1-1/d}\right)$$

taking

$$c' \geq \frac{\alpha c}{2^{1/d}-1}.$$

And thus

$$k_m = K(N) = O\left(\frac{N}{m^{1/d}}\right).$$

□

We now assume that the mesh is traversed by chunks of m elements, i.e. each chunk contains m consecutive elements in the layout that should all be processed (in any order) before accessing another chunk anywhere in the layout. The size of a chunk m expresses how much the filter access pattern respects the layout locality. As spatially close elements in the mesh tend to be close in the layout, filters with spatially coherent access patterns use big chunks.

Theorem 3 (Chunk traversal): The CO layout guarantees that a traversal by chunks of size $m \leq M$ of an N-size mesh induces less than $N/B + O(N/m^{1/d})$ cache misses where B and M are the block and cache size, respectively.

Proof: Assume first that there is no edge cut in T_m, i.e. processing an element in a chunk only accesses elements in the same chunk. Processing a chunk would not induce any cache miss beside the m/B compulsory ones to read the chunk as the entire chunk fits in cache. This sums up to N/B cache misses for processing all the mesh. However, processing an element may require data that are not in the same chunk, causing $O(1)$ extra cache-misses per edges linking elements in different chunks: the cut edges. Accessing an element in another chunk induces one cache miss to read the element and may generate another one as it can evict a block of the current chunk that may still be needed. The total number of these extra cache misses is proportional to the number of cut edges: $O\left(\frac{N}{m^{1/d}}\right)$. We thus obtain the claimed bound. □

For the sake of simplicity, we assume in the proof that the chunks are perfectly aligned with the leaves of T_m. One can easily show that there are still $O\left(\frac{N}{m^{1/d}}\right)$ cut edges when the chunks are not aligned with the leaves. Consider leaves of size $2m$ and add edges within a leaf that link different chunks. This only modifies the number of cut edges by a constant factor.

Corollary 4 (Layout order traversal): The CO layout guarantees that a layout order traversal of an N-size mesh induces less than $N/B+O(N/M^{1/d})$ cache misses where B and M are the block and cache size, respectively.

Proof: A layout order traversal is a traversal by chunks of size M. □

With our layout, a visualization filter still needs to traverse the mesh in an order coherent with the layout, but the assumption is strongly relaxed compared to a layout order traversal. We believe that we could obtain the same performance guarantee slackening the traversal by chunks assumption to rely only on the characteristics of the mesh itself. We are however not able to prove it yet. Experiments (cf. section 6) use visualization filters that traverse the mesh in the layout order (e.g. gradient, vtkiso, cpuiso, etc.), filters that traverse the mesh by connectivity (e.g. connectivity, RC), and filters that traverse the mesh through another data structure (e.g. CpuTree). All of them yield speed up, which indicates that in practice the chunk traversal assumption is usually verified for some m.

At this point we cannot directly compare this algorithm with OpenCCL. OpenCCL is based on a meta-heuristic and no upper bound on the quality of the resulting layout is given.

5.3 Layout Computation

We implemented the geometric separator algorithm in C++. We first randomly generate all the $n_h n_c$ separators (Alg. 1). We then traverse all the cells of the mesh and for each of them we check that its points are on the same side of the separator. If not, the cell is cut by the separator and we increment the cut size by 1. Using cells instead of edges to select the best separator produces a very close result and allows us not to compute the edges of the graph, a task that can be computationally expensive. The bound of theorem 3 still applies as at most a constant number of edges correspond to a cell. All separators are checked against a cell before going to the next one. This allows us to dereference each cell index only once for the entire separator computation.

To keep the memory usage low, we do not project all the points before evaluating a separator but project them on the fly. This induces duplicate computation as a point is used in several cells but keep memory overhead close to zero. That way we do not need to store an entire copy of the points in memory.

Once we found the best separator, the points of the mesh are reordered according to this separator. All points laying to the left of the separator are moved to the left part of the array and points laying to the right are moved to the right part of the array. The same partitioning is done on cells. When a cell is cut by the separator we choose a side according to the center of gravity of the cell. We then recurse on the left and right mesh generated. This algorithm is very similar to a quicksort and could be efficiently parallelized.

We stop when a submesh has a size lower than 8. We choose $n_c = 2$ and $n_h = 30$ for the experiments (Alg. 1). As the randomized centerpoint algorithm is quite good we can keep n_c low. During our experiments we noticed that even substantial changes of all these parameters did not impact significantly the quality of the generated layout.

5.4 Choosing the Access Graph

The algorithm described in section 5 can be applied to any access pattern as long as the corresponding graph is an overlap graph (actually the algorithm still works if it is not the case but the bound on cache-misses does not hold). However to generate a new layout for each application is not practical. For instance to compute a volume rendering by ray casting of the mesh, one might want to optimize the mesh layout according to the rays direction. Both our algorithm and OpenCCL are too slow to generate a layout before each image generation.

In practice it is better to compute only once a layout that will be efficient in general. We choose in the implementation to only consider the graph where vertices are points of the mesh and edges link two points sharing a cell (Fig. 5(a)). Using this access graph produces an efficient layout for most access patterns as access graphs for connectivity traversal, neighborhood and attributes operations look alike (Fig. 5). Following on our volume rendering example, this layout will be reasonably good for any ray direction. One ray traversing c cells of the mesh induces $c/B^{1/3}$ cache-misses while a layout optimized for this specific direction may induce only c/B cache-misses (but as bad as 1 cache-miss per cell for an orthogonal direction). In this specific case, packing rays should also improve performance of the more general layout to c/B.

5.5 Cells Layout

A mesh layout tries to optimize both points and cells ordering. As points and cells are usually accessed in a similar way, a consistent ordering for points and cells is better. For instance in an isosurface extraction, points composing a cell are often accessed immediately after the cell itself. Thanks to our geometric approach, the same geometric separators can be applied for both points and cells. A separator cutting few edges

for the points graph (Fig. 5(a)) also cut few edges for the cells graph (Fig. 5(b)). It is not possible to do the same with OpenCCL for example as their separators are combinatorial and not geometric.

Computing points and cells layouts independently often leads to a larger computation time and a lower runtime efficiency. For OpenCCL, computing the cell layout is around 3 times slower than computing the points layout on our meshes. A consistent cells layout (min-vertex), can be deduced from the points layout by sorting the cells using the minimal index of their points.

Having consistent points and cells layouts can also improve runtime performance. The min-vertex approach enforces the consistency. For instance, min-vertex with OpenCCL leads to a 20% runtime performance increase compared to applying OpenCCL to both points and cells. The FastCOL layout further enforces this consistency. On large meshes the consistent layout for points and cells with the BSP tree is up to 10% faster than the min-vertex layout applied to FastCOL points layout on isosurface extraction.

5.6 Consistent BSP Tree

The BSP tree defining the partitioning of the mesh can be used as an acceleration structure that has the advantage of being consistent with the layout. In this article we show how it can be used as a min-max tree for isosurface extraction. At each node of the BSP tree we store the minimum and maximum value of the scalar field in the corresponding region of the mesh. A region that do not contain any cell intersected by the isosurface can be quickly discarded.

An interesting property of this BSP-tree is that each region corresponds to a small part of the mesh stored sequentially in memory (Fig. 8). When traversing the BSP-tree in prefix order and examining the mesh cells that might contain a part of the isosurface, mesh cells are accessed sequentially. The sequence of cells can jump part of the mesh but it never goes back. This leads to a layout order traversal of the mesh that induces fewer cache-misses (see experimental results in section 6.3).

6 Experiments

We compare the performance of the initial, geometric, OpenCCL and FastCOL layouts on various meshes and access patterns. For sake of conciseness, we present only some representative results. Full results are provided in our research report [28].

6.1 Architectures, Filters and Meshes

We took 9 different meshes[1], processed to generate several instances of various sizes. We used tetgen[2] to refine the meshes by adding a volume constraint to each tetrahedron[3]. For each mesh and each size (100 k, 1 M, 10 M and 50 M cells) we generated two finer meshes. In the first one, all tetrahedra have approximatively the same volume. In the second one, we used a volume constraint proportional to the inverse of the gradient of the scalar field to mimic an adaptive mesh refinement. It leads to a set of 50 meshes that can be divided by their size into 4 groups: 5 meshes of about 100 k cells, 10 meshes of about 1 M cells, 17 meshes of about 10 M cells, and 18 meshes of about 50 M cells.

The experiments were conducted on three different architectures, two classical CPU architectures with 2 cache levels (AMD Opteron875 @ 2.2Ghz, cache L1 8KB, cache L2 1MB and Intel Core2 E6750 @ 2.66Ghz, cache L1 32KB, cache L2 4MB), and one GPU architecture (NVIDIA GTX280 with 1GB of memory) tested to probe the influence of the layout on the number of coalesced memory accesses.

Ten filters were tested on each layout, using VTK filters [26], homemade CPU codes or Cuda (version 1.3) codes for the GPU tests:

- **Gradient.** The VTK gradient filter computes the gradient of the mesh scalar field. Each gradient value is computed from the local scalar value and the values of neighbor points. Data are processed in the order given by the point layout. Using the terms introduced in section 3 this is a point layout order traversal with point neighborhood operations.
- **Connect.** The VTK connectivity filter applies a breadth first search on the mesh to compute the connected region each cell lies in. This filter uses a connectivity traversal.
- **RC.** A mesh volume rendering computed by the VTK Bunyk Ray Cast filter [29]. Each ray traverses the mesh cell by cell and then accesses points attributes to compute the contribution of the cell to the pixel color. This is a connectivity traversal with point attributes operations.
- **PT.** A mesh volume rendering computed by the VTK Projected Tetrahedra filter [30]. Tetrahedra are sorted by their centroid according to the viewing direction and then sent to the GPU for projection. During the sorting phase, each tetrahedron accesses its points and tetrahedra are processed in the order given by the cell layout. This is a cell layout order traversal with points attributes operations. Both CPU and GPU computations are included in the time measure but only the CPU part is included in the number of cache misses.
- **HAVS.** A mesh volume rendering computed by the VTK HAVS filter [31]. Data accesses are similar to PT. Again, both CPU and GPU computations are included in the time measure but only the CPU part is included in the number of cache misses.
- **VtkIso.** The VTK isosurface extraction filter implements the marching tetrahedra algorithm. Each cell accesses to its points. Cells are processed in the cell layout order. This is a cell layout order traversal with points attributes operations.
- **CpuIso and GpuIso.** One CPU and one GPU homemade implementation of the marching tetrahedra isosurface extraction algorithm.
- **CpuTree and GpuTree**. The CpuIso and GpuIso code extended to include a min-max tree acceleration structure.

1. Blunt fin, buckyball, langley fighter, liquid oxygen post, plasma64, san fernando and spx models are provided by the AIM@SHAPE Shape Repository (http://shapes.aim-at-shape.net/). Torso is courtesy of SCI and the last one is not published.

2. Available at http://tetgen.berlios.de/.

3. We used the command `tetgen -raq`.

TABLE 1
Layout computation (on Opteron)

Mesh size		OpenCCL		FastCOL	
#cells	Bytes	Time	Mem. space	Time	Mem. space
100k	3.7 MB	4.5 s	123 MB	1.2 s	22 MB
1M	43 MB	54 s	1.24 GB	17 s	81 MB
10M	370 MB	9 min 24 s	9.96 GB	3 min 50 s	0.56 GB
50M	1.8 GB	NA	> 96 GB	26 min 44 s	2.72 GB

For the OpenCCL layout a kd-tree is used. For the FastCOL layout two versions are tested: one based on a kd-tree and one relying on the BSP tree built when computing the layout. Only some cells are processed, in a min-max tree driven order. This is a data structure traversal with points attributes operations. We only time the processing of the cells intersected by the isosurface and not the tree traversal (the kd-tree is the same for both layout and the code for its traversal is not optimized).

The bigger meshes (50 M cells) have not been tested with the volume rendering filters due to the very large execution time, nor on the GPU that has only 1GB of memory.

For our GPU implementation, we only measure the time to compute the kernel and not the memory transfers between CPU and GPU, which take the same amount of time for all layouts.

6.2 Layout Algorithm Performance

All layouts have been prepared on an Opteron875 @ 2.2Ghz with 32 GB of memory and 64 GB of swap. Table 1 shows the execution time and memory needs for computing the OpenCCL and FastCOL layouts. Our FastCOL program is about three times faster than the OpenCCL one. It requires far less memory. The bigger meshes with 50 M cells have not been processed with OpenCCL because their computation would have required more than 96 GB of memory. The multilevel heuristic used in OpenCCL may explain such memory consumption. Space is needed at each level to store the coarsened access graph and additional information to undo the coarsening operation.

Computing the geometric layout, a coordinate sort by the x, y and z axes, is very fast (less than 40 s for the biggest meshes) and compact in memory.

6.3 Mesh Layout Performance

We measured the execution time, the number of L1 and L2 cache-misses using the PAPI software [32] for the CPU tests, and the number of uncoalesced parallel accesses for the GPU ones. For each experiment (architecture, layout and algorithm fixed), the execution time, the numbers of cache-misses and uncoalesced accesses are very stable.

Tables 2 and 3 show the means of the speedup (“Speedup”), ratio of saved L2 cache-misses on CPU (“L2”) and ratio of coalesced memory accesses on GPU (“Coal.”) for the geometric, OpenCCL and FastCOL layouts. These ratios are relative to the performance obtained with the initial layout. In all cases, higher values are better. Table 2 gathers the results for the tests visiting the entire mesh, while Table 3 displays the performance results for the min-max tree accelerated isosurface extraction using a kd-tree for the OpenCCL and FastCOL layouts, and the BSP tree computed for the FastCOL layout (“FastCOL (bsp)”).

TABLE 2
CPU and GPU performance ratios relative to the original layout (on Core2)

	Mesh	Geometric		OpenCCL		FastCOL	
	size	Speedup	L2	Speedup	L2	Speedup	L2
Gradient	100k	1.02	1.51	1.01	1.49	1.02	1.52
	1M	1.06	3.53	1.07	4.03	1.08	3.94
	10M	1.07	2.36	1.15	8.22	1.15	7.81
	50M	1.1	1.34			1.36	10.53
Connect	100k	0.95	0.94	1.12	1.17	1.11	1.21
	1M	0.97	0.95	1.16	1.19	1.19	1.19
	10M	1.09	1.08	1.45	1.49	1.46	1.49
	50M	0.89	0.87			1.66	1.9
RC	100k	0.98	1.08	1.05	1.4	1.06	1.36
	1M	1.01	1.07	1.2	1.8	1.2	1.79
	10M	0.76	0.72	3.28	5.02	3.2	4.89
PT	100k	1.06	0.82	0.93	1.18	1.15	1.18
	1M	0.91	0.64	1.09	1.51	1.1	1.52
	10M	0.97	0.9	1.37	2.66	1.37	2.65
HAVS	100k	1.02	2.04	1.01	2	1.08	2
	1M	1.14	3.43	1.06	4.04	1.09	4.03
	10M	1.2	1.9	1.33	5.96	1.32	5.77
VtkIso	100k	0.99	1.04	1.04	1.06	1.04	1.09
	1M	1.1	1.22	1.15	1.24	1.15	1.24
	10M	1.32	1.71	1.44	1.79	1.44	1.78
CpuIso	100k	1.06	1	1.16	1.02	1.17	1.02
	1M	1.71	2.85	2.34	2.8	2.35	2.78
	10M	2.28	5.4	4.08	5.78	3.99	5.68
	50M	0.97	0.79			4.87	6.84
		Time	Coal.[4]	Time	Coal.[4]	Time	Coal.[4]
GpuIso	100k	0.96	1.18	1.56	3.08	1.52	2.97
	1M	1.26	1.04	2.2	2.59	2.11	2.38
	10M	1.83	1.25	4.09	3.85	3.8	3.39

6.3.1 CO Layouts on CPU

Table 2 shows higher performance ratios with larger meshes where cache effects are predictably more important. Indeed, with smaller meshes, a bigger part of the mesh can be loaded in the cache, whatever the layout is. Both CO layouts, OpenCCL and FastCOL, lead to speedup ratios from 1.01 to 4.87, all tests being in-core. It shows the benefits of CO layouts that can bring significant performance increases without any change to the application. The geometric layout is significantly less efficient for most of the tests, a result analyzed in section 6.3.2.

The FastCOL layout reaches similar performance compared to OpenCCL, while providing a theoretical performance guarantee.

Some important differences are observed between the L2 cache-miss ratio and the speedups for the Gradient and HAVS

4. Ratio of coalesced parallel memory accesses on GPU.

TABLE 3
CPU and GPU performance ratios relative to the original layout for tree accelerated isosurface extraction (on Core2)

	Mesh size	OpenCCL Speedup	OpenCCL L2	FastCOL Speedup	FastCOL L2	FastCOL (bsp) Speedup	FastCOL (bsp) L2
CpuTree	100k	1.23	1.31	1.21	1.30	1.37	1.23
	1M	1.46	1.8	1.45	1.74	1.75	1.72
	10M	2.37	3.14	2.31	3.02	2.75	3.06
	50M			2.92	3.47	4.55	4.85
			Coal.[4]		Coal.[4]		Coal.[4]
GpuTree	100k	1.20	1.85	1.18	1.75	1.33	1.90
	1M	1.63	1.81	1.57	1.67	1.84	1.89
	10M	2.50	2.14	2.34	1.86	2.79	2.14

tests. Gradient is computationally intensive and HAVS extensively uses the GPU, making the cache-miss overhead a small fraction of the overall computation time.

We can also observe that measured speedups are generally smaller with VTK filters than with homemade ones. This clearly appears for the isosurface filter that is implemented with VTK (VtkIso) compared to the homemade code (CpuIso). The VTK implementation shows a maximum speedup of 1.44 whereas our implementation goes up to 4 (with a smaller global execution time). The VTK library is not fully optimized and performs several other computations. For instance, after the extraction of the isosurface, the VtkIso filter merges the identical points to provide a mesh (instead of a triangle soup) as a result.

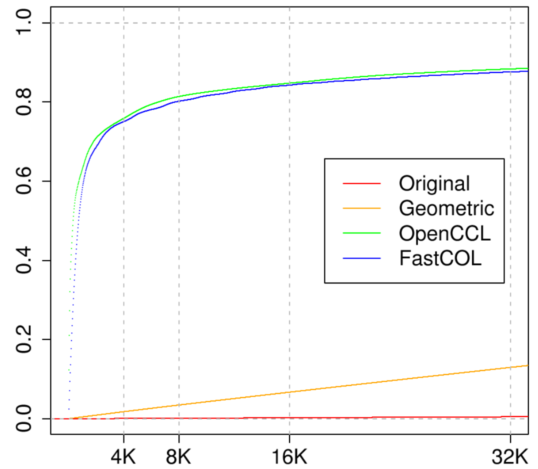
6.3.2 Edge Lengths and Layout Quality

To better analyze the properties of the different layouts, we analytically relate the performance improvements to the better data locality in memory. We call "edge length" the memory gap between two vertices of the same edge in the vertex array loaded in memory. If a mesh has shorter edges, more of them will fit in cache and a better performance should be observed. Other analysis could also be conducted with similar metrics. For example, instead of considering the length of edges, we can consider the "size of a cell", which would be either the maximum memory gap between all vertices of the cell in the vertex array, or the maximum memory gap between all adjacent cells in the cell array.

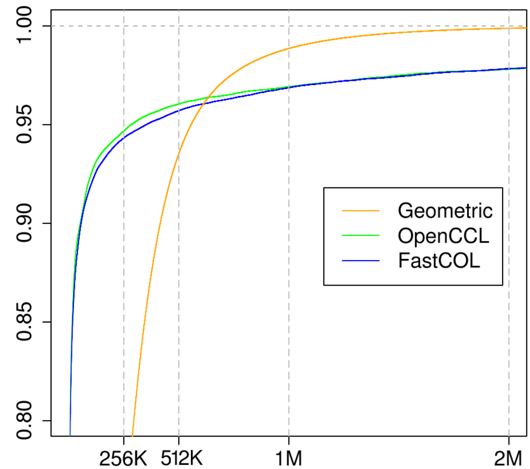
Figure 10 shows the cumulative distribution of edge lengths for the 10 M cells torso mesh[5]. The two graphs are focused around the L1 and L2 cache sizes of the tested architectures. CO layouts appear to favor small edge lengths.

The geometric layout behaves differently. The amount of small edges is reduced compared to CO layouts, but almost all edges have a length shorter than 2M. Actually, by construction, the edge lengths are shorter than the size of two entire slices of the mesh in the x direction. The layout thus leads to a good performance when two slices in the x direction can fit in the L2 caches. This is visible in the results where the geometric

5. The other meshes produce similar graphs.



(a) Zoom on L1 cache sizes



(b) Zoom on L2 cache sizes

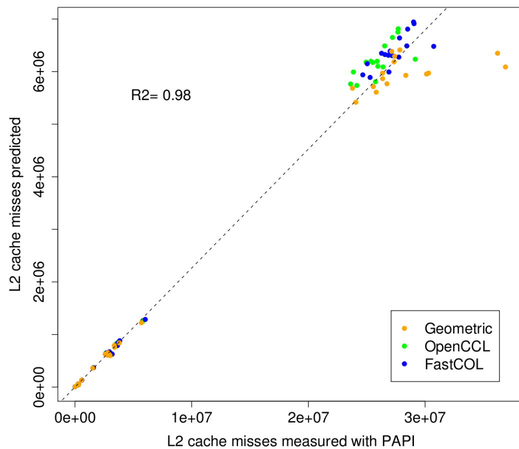
Fig. 10. Cumulative distribution function of edge lengths for various layouts applied to the torso mesh (10M resolution). The CO layouts (OpenCCL and FastCOL) favor small edges: 80% of their edges have a length below 8K (a) and 95% below 256K (b). The original layout does not appear on graphic (b) as the cumulative distribution is too small: only 40% of its edges have a length smaller than 2M.

layout performs well for small meshes while it is outperformed by the CO layouts for the bigger ones. For small meshes, the geometric layout is often slightly less efficient due to its low efficiency with respect to the L1 cache (L1 cache-miss ratios omitted for sake of conciseness).

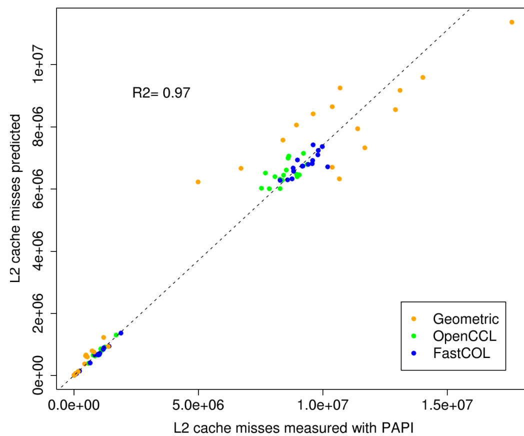
We now estimate the number of cache misses using an edge length based metric, and show that there is a strong correlation with the actual number of observed cache misses. Let N be the size of a mesh (in bytes), E the set of all edges of the mesh, B the cache line size and M the cache size, we estimate the number of cache-misses by:

$$ExpectedCM \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M}$$

where λ_e is the length of the edge e. We count the number of



(a) Core2



(b) Opteron

Fig. 11. Correlation between edge lengths and measured L2 cache-misses on CpuIso. Each point corresponds to a mesh with geometric, OpenCCL or FastCOL layouts.

cache-misses for a linear full read of the data arrays and we add one cache-miss per edge whose length is bigger than the cache size M.

The theoretical upper bound $\frac{N}{B} + O(k_M)$ for our FastCOL algorithm is larger because we count one cache miss for each cut edge in the T_M BSP tree (Th. 3). Some cut edges counted in our theoretical bound can in fact have an edge length shorter than M.

In figure 11, we display the correlation between the expected cache-misses for the considered mesh layouts and the cache misses observed on both CPU architectures. The $\frac{N}{B}$ factor has been subtracted from this measure as it does not depend on the layout. The correlation between expected cache misses and actual ones is very high with a calculated r^2 of 0.98.

Notice that the layout quality is not only influenced by the edge lengths (directly linked with the number of edges cut), but also by the dispersion of cut edges. The number of cache misses is smaller than the number of edges cut by a separator if successive cut edges point toward the same memory block (Fig. 12).

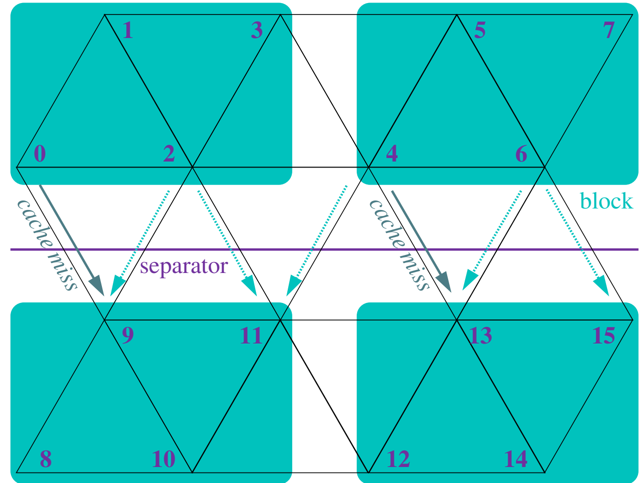


Fig. 12. Mesh layout using the solid purple line as a separator. We assume that neighbor points are needed to process each mesh point (e.g. the VTK gradient filter). Even if the separator cuts seven edges of the access graph, only two induce a cache miss if the cache can hold at least 3 blocks (in light green).

6.3.3 CO Layouts on GPU

The last test of Table 2 (GpuIso) evaluates the benefits of CO layouts on a Nvidia GPU. We measure the execution time and the number of coalesced accesses. All data are stored in the global GPU memory. There is no cache mechanism involved. The only block-based data transfer that occurs is related to coalesced parallel memory accesses. The concurrent global memory access performed by all threads of a half-warp (16 threads) is coalesced into a single memory block transfer as soon as the data accessed lie in the same 128 Bytes segment for 32, 64 and 128 bit data. The context is very different from cache based CPUs. We only have a single small block $M = B = 128$ Bytes. CO layouts lead to speedups ranging from 1.52 to 4.09, which is significant knowing that only the layout is modified. It shows they efficiently minimize the edge lengths even for very small sizes (128 Bytes). OpenCCL slightly outperforms the FastCOL layout. The geometric layout suffers from too long edges.

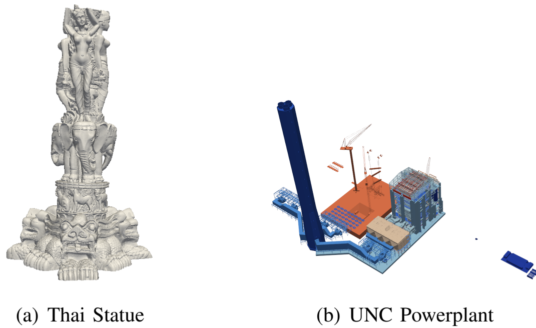
Various applications can share work between the CPU and the GPU. The same CO layout can thus be shared between the CPU and the GPU to reduce both cache-misses and non-coalesced accesses.

6.3.4 Layout Consistent Min-Max Tree

In all tests the OpenCCL and FastCOL layouts show similar results. However the FastCOL layout is computed from a BSP tree that can be used as an internal, layout consistent, acceleration data structure to further take advantage of this layout. Experiments of Table 3 reveal that using it as a min-max tree for accelerating an isosurface extraction brings significant additional performance improvements. Compared to OpenCCL or the FastCOL layout that both use an external min-max kd-tree, the min-max BSP tree provides a performance improvement of 11% to 55% on CPU and of 11% on

TABLE 4
Comparison of OpenCCL and FastCOL on two triangle meshes (Thai statue and UNC powerplant) with the VTK connectivity filter and the VTK depth sort filter (Median of 30 runs on Core2).

Mesh	Filter	OpenCCL Speedup	OpenCCL L2	FastCOL Speedup	FastCOL L2
Thai Statue	Connectivity	1.09	1.26	1.08	1.23
	Depth Sort	1.19	1.40	1.15	1.31
Power plant	Connectivity	0.84	0.71	0.89	0.90
	Depth Sort	0.88	0.76	0.94	0.92



(a) Thai Statue (b) UNC Powerplant

Fig. 13. Two triangle meshes

GPU. The leaves to be processed being M-size leaves of the BSP tree, they are less likely to trigger cache-misses than the leaves of the kd-tree computed independently from the layout. The speedup is smaller on the GPU because we could not use the biggest meshes (50M cells) due to memory constraints.

6.3.5 Comparison on a Scanned Model

We compare OpenCCL and FastCOL on the Thai statue[6]. This is a triangle mesh with 5M vertices and 10M triangles (Fig. 13(a)). To build the layout, OpenCCL needs 912s and FastCOL 311s, which is comparable with the tetrahedral meshes. We compared these two layouts on two VTK filters, the connectivity filter previously used and the depth sort filter that sorts triangles with respect to a view direction. We cannot use all previous filters as they require a tetrahedral mesh. On these two filters, the performances of both layouts are comparable, around $10-20\%$ faster than the original layout, OpenCCL being slightly better (Tab. 4).

6.3.6 Comparison on a CAD Mesh

We now compare OpenCCL and FastCOL on the UNC Powerplant mesh[7]. This is a triangle mesh with 12.7M triangles and 11M points and a complex geometry and topology (Fig. 13(b)). It consists of several totally disconnected parts (1,083,733). We reorder each of those parts independently with OpenCCL and FastCOL. OpenCCL reorder points and then use min-vertex to find the cell order. For the whole mesh, OpenCCL needs 671s and FastCOL only 223s.

6. available at http://graphics.stanford.edu/data/3Dscanrep/
7. available at http://www.cs.unc.edu/ geom/Powerplant/

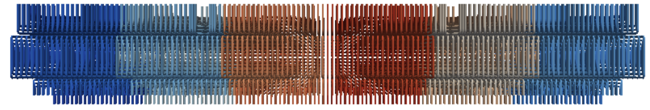


Fig. 14. Visual illustration of the original cell layout of the section 01 part a of the UNC powerplant model. Successive cells in memory are colored from blue to red.

We again compare those two layouts on the connectivity filter and the depth sort filter. Both layouts perform worse than the original that is already well optimized (Tab. 4). In the original layout, each connected part is stored contiguously and each of those parts is then well organized. No big improvement was expected due to this already good layout (Fig. 14). Previous work on this mesh led to improvements over the original layout using experiments much more based on the geometry than the filters we used: view dependent rendering in [19] and collision detection in [20].

6.3.7 Comparison with a Space Filling Curve Approach

We now compare our layout to a space filling curve approach. We use the Z-curve as in [2]. To compute the layout efficiently, we decompose the space using a kd-tree[8] until there is only one point for each leaf and then order the leaves in the order of the Z-curve. This algorithm is very similar to FastCOL except that, instead of looking for an efficient separator at each step of the recursion, we use planes parallel to the x,y,z axes cutting exactly in half the set of points.

The space-filling curve approach is faster but does not take into account the topology of the mesh. The kd-tree does not provide an upper bound on the number of cells cut by the plane separator. However this approach performs almost as well as FastCOL and OpenCCL on most of our meshes and very well on regular meshes.

As the space-filling curve does not take into account the topology of the mesh, it can perform badly on specific meshes. We created a mesh with a high density of points and cells where the kd-tree cut the mesh. To do so, we first generated a set of points in $[-1,1]^3$ with a high density around the planes $x=0$, $y=0$, $z=0$, $x=\pm0.5$, $y=\pm0.5$, $z=\pm0.5$. We tetrahedralized them with tetgen. We mapped the scalar field of one of our meshes using a linear interpolation. On CpuIso the FastCOL layout is about 1.4 time faster than the space-filling curve layout.

7 Conclusion

We introduced FastCOL, an algorithm relying on Miller *et al.* [6] geometric separator for computing CO mesh layouts. To our knowledge this is the first CO layout algorithm for unstructured meshes with a guaranteed theoretical upper bound of $N/B+O(N/M^{1/d})$ cache-misses.

Experiments show that this algorithm requires significantly less computation time and memory than OpenCCL, the best

8. We used the VTK implementation of the kd-tree.

known CO mesh layout algorithm [5]. Without modifying the visualization algorithms, both CO layouts can bring comparable performance improvements on CPUs where they reduce the number of cache-misses, as well as on GPU architectures where they favor parallel coalesced data accesses. FastCOL improves its performance by more than 10% when using the layout consistent BSP tree produced by the algorithm as an acceleration data structure instead of an external one.

ACKNOWLEDGMENTS

This work is partly funded by CEA/DIF/DSSI, Bruyères le Châtel, France and by the Agence Nationale de la Recherche contract ANR-07-CIS7-003.

REFERENCES

[1] NVIDIA, "Nvidia cuda programming guide 2.3.1," 2009.
[2] V. Pascucci and R. Frank, "Global Static Indexing for Real-Time Exploration of Very Large Regular Grids," in *Proc. of Supercomputing '01*, 2001, p. 45.
[3] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, p. 1116, 1988.
[4] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," in *Proc. of FOCS '99*, 1999, p. 285.
[5] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," in *Proc. of SIGGRAPH '05*, 2005, p. 886.
[6] G. Miller, S.-H. Teng, W. Thurston, and S. Vavasis, "Geometric Separators for Finite-Element Meshes," *J. Scientific Computing*, vol. 19, no. 2, p. 364, 1998.
[7] U. Meyer, P. Sanders, and J. Sibeyn, Eds., *Algorithms for Memory Hierarchies, Advanced Lectures*, ser. Lecture Notes in Computer Science, vol. 2625. Springer, 2003.
[8] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, p. 101, 2005.
[9] M. Bader and C. Zenger, "Cache oblivious matrix multiplication using an element ordering based on a Peano curve," *Linear Algebra and Its Applications*, vol. 417, no. 2–3, p. 301, 2006.
[10] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proc. of SPAA '07*, 2007, p. 93.
[11] H. Hoppe, "Optimization of mesh locality for transparent vertex caching," in *Proc. of SIGGRAPH '99*, 1999, p. 269.
[12] A. Bogomjakov and C. Gotsman, "Universal rendering sequences for transparent vertex caching of progressive meshes," in *Proc. of GRIN '01*, 2001, p. 81.
[13] G. Lin and T. P.-Y. Yu, "An Improved Vertex Caching Scheme for 3D Mesh Rendering," *Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, p. 640, 2006.
[14] P. Sander, D. Nehab, and J. Barczak, "Fast triangle reordering for vertex locality and reduced overdraw," *Transaction on Graphics*, vol. 26, no. 3, p. 89, 2007.
[15] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola, "Single-strips for fast interactive rendering," *The Visual Computer*, vol. 22, no. 6, p. 372, 2006.
[16] J. Chhugani and S. Kumar, "Geometry engine optimization: cache friendly compressed representation of geometry," in *Proc. of I3D '07*, 2007, p. 9.
[17] M. Isenburg and P. Lindstrom, "Streaming meshes," in *Proc. of Visualization '05*, 2005, p. 231.
[18] "OpenCCL: Cache-Coherent Layouts," http://www.cs.unc.edu/ geom/COL/OpenCCL/.
[19] S.-E. Yoon and P. Lindstrom, "Mesh Layouts for Block-Based Caches," *Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, p. 1213, 2006.
[20] S.-E. Yoon and D. Manocha, "Cache-Efficient Layouts of Bounding Volume Hierarchies," *Computer Graphics Forum*, vol. 25, no. 3, p. 507, 2006.
[21] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *Transactions on Graphics*, vol. 11, no. 3, p. 201, 1992.
[22] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Optimal isosurface extraction from irregular volume data," in *Proc. of VVS '96*, 1996, p. 31.
[23] Y.-J. Chiang and C. Silva, "I/O optimal isosurface extraction," in *Proc. of Visualization '97*, 1997, p. 293.
[24] Y.-J. Chiang, C. Silva, and W. Schroeder, "Interactive out-of-core isosurface extraction," in *Proc. of Visualization '98*, 1998, p. 167.
[25] Y.-J. Chiang and C. Silva, "External memory techniques for isosurface extraction in scientific visualization," in *External memory algorithms*, 1999, p. 247.
[26] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed.* Kitware Inc., 2004.
[27] K. Clarkson, D. Eppstein, G. Miller, C. Sturtivant, and S.-H. Teng, "Approximating center points with iterated radon points," in *Proc. of SoCG '93*, 1993, p. 91.
[28] M. Tchiboukdjian, V. Danjean, and B. Raffin, "Binary Mesh Partitioning for Cache-Efficient Processing," INRIA, Tech. Rep., 2009.
[29] P. Bunyk, A. Kaufman, and C. Silva, "Simple, Fast, and Robust Ray Casting of Irregular Grids," in *Proc. of Visualization '97*, 1997, p. 30.
[30] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," in *Proc. of SIGGRAPH '90*, vol. 24, no. 5, 1990, p. 63.
[31] S. Callahan, M. Ikits, J. Comba, and C. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, p. 285, 2005.
[32] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, p. 189, 2000.

Marc Tchiboukdjian received an engineering degree in Computer Science at ENSIMAG in France in 2007. He is currently a PhD student in the MOAIS team of the Laboratoire d'Informatique de Grenoble where he is studying efficient scientific visualization algorithms. His research interests include scientific visualization, cache-aware and cache-oblivious algorithms, parallel computing and gpu programming.

Vincent Danjean is research scientist at University Joseph-Fourier at Grenoble. He received a Ph.D. on parallel computing from the École normale supérieure de Lyon in 2004. After a 1 year postdoc in French CEA institute working on large parallel computers, he has been hired in the MOAIS INRIA team to work on middleware for large and efficient parallel computing. Today, his research activity focuses on high performance parallel computing on large scale machine, multicore machines and on embedded hardware such as GPU. He develops the KAAPI software that wins several times the GRIDS@WORK international challenge, being able to deploy and efficiently run applications on several thousands of cores.

Bruno Raffin is research scientist at INRIA Rhône-Alpes Grenoble. He received a Ph.D. on parallel computing from the Université d'Orléans in 1997. After a 2 year postdoc in USA working on large parallel computers, he returned to France to work on the association of virtual reality, scientific visualization and parallel computing. Today his research activity focuses on high performance interactive computing. He develops the FlowVR software suite and manages the real-time multi-camera 3D modeling platform called Grimage. He has co-chaired the 2004 and 2006 Eurographics Symposium on Parallel Graphics and Visualization and participated to the program committee of several international conferences.

Extraction d'isosurfaces parallèle et efficace en cache 6

Sommaire

6.1 Résumé des contributions **95**
6.2 Discussion et perspectives **101**
6.3 Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores **103**
1 *Introduction* 103
2 *Marching Tetrahedra Review* 104
3 *Cache-Efficient Isosurface Extraction* 105
4 *Parallel Cache-Efficiency* 106
5 *Implementation and Experiments* 107
6 *Related Work* 111
7 *Conclusion* 111
References 112

Dans le chapitre précédent, nous nous sommes intéressés à l'optimisation de l'organisation mémoire d'un maillage accédé par un filtre de visualisation séquentiel et utilisant un schéma d'accès global directement basé sur le maillage (*layout order* ou *connectivity traversals*). Dans ce chapitre, nous nous focalisons sur un filtre de visualisation classique, l'extraction d'isosurfaces, et nous examinons les deux cas laissés en suspens dans le chapitre précédent :

- Comment améliorer les performances lorsqu'un filtre n'accède pas le maillage directement mais à travers une autre structure de données (*external data structure traversal*) ?
- Quel est l'impact sur les défauts de cache lorsqu'un filtre s'exécute en parallèle ?

6.1 Résumé des contributions

Nous résumons ici les contributions de l'article *Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores* [TDR10b] qui s'attèle à répondre à ces deux questions.

L'extraction d'isosurface est un filtre de visualisation classique qui consiste à extraire une surface à valeur scalaire constante à partir d'un maillage tridimensionnel (*cf.* figure 8 page 8). L'algorithme standard pour extraire une isosurface est *Marching Tetrahedra* (MT) [HJ04]. Pour chaque cellule du maillage, on construit une approximation de

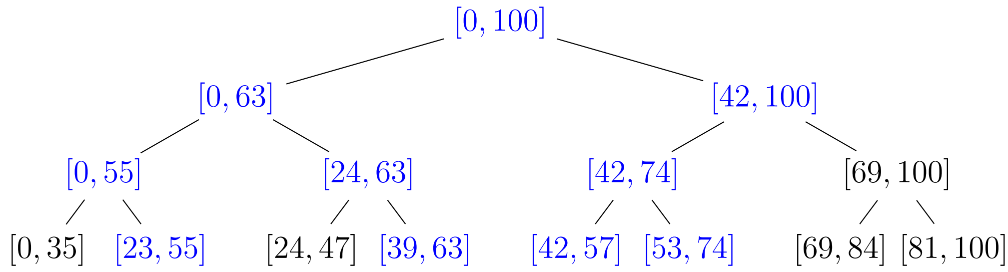


FIGURE 6.1 – Arbre min-max. Chaque nœud de l'arbre correspond à une région du maillage dont les valeurs du champ scalaire sont comprises dans l'intervalle associé au nœud. Ici, si la valeur cherchée est 54, seuls les nœuds en bleu sont examinés ce qui réduit la complexité de l'algorithme.

l'isosurface à l'intérieur de la cellule en interpolant linéairement le champ scalaire à partir des valeurs données aux points de la cellule. Le résultat de l'algorithme est une soupe de triangles que l'on peut afficher directement ou transformer en maillage pour des traitements ultérieurs [1].

Arbre min-max adapté à FastCOL

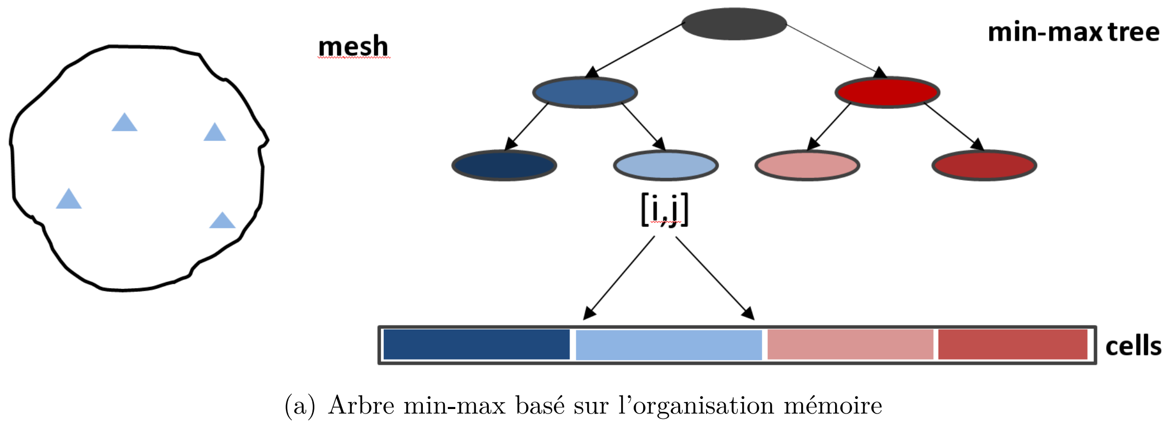
L'inconvénient de l'algorithme MT est qu'il doit parcourir toutes les cellules du maillage, même celles qui ne sont pas intersectées par l'isosurface. Ces dernières ne génèrent pas de triangles et il est donc inutile de les traiter. Il existe plusieurs structures de données accélératrices qui permettent à l'algorithme MT de ne parcourir que les cellules utiles [HJ04]. Nous avons choisi d'examiner la structure qui est implémentée dans la bibliothèque VTK : l'arbre min-max. Un arbre min-max est un arbre de décomposition du maillage (*octree*, *kd-tree* ou *BSP tree*) enrichi avec des intervalles de valeurs scalaires. Pour chaque nœud de l'arbre, on stocke les valeurs minimales et maximales du champ scalaire dans la région du maillage correspondant au nœud. Cette structure permet d'accélérer la recherche des cellules intersectées par l'isosurface en éliminant des régions entières du maillage en un simple test. Si la valeur de l'isosurface cherchée est v, il est inutile de parcourir une région dont les valeurs scalaires sont entre m et M si $v \notin [m, M]$.

L'algorithme de MT accéléré par un arbre min-max est un exemple de filtre de visualisation dont le schéma d'accès global est de type *external data structure*. En effet, les cellules du maillage sont accédées dans l'ordre des feuilles de l'arbre min-max. Suivant la façon dont l'arbre a été construit, il est possible que l'ordre donné sur les cellules n'ait pas une bonne localité dans le maillage (les cellules accédées consécutivement ne sont pas proches dans la topologie définie par le maillage *cf.* figure 6.2(b)). L'organisation mémoire produite par FastCOL n'aura dans ce cas que peu d'effets sur les performances.

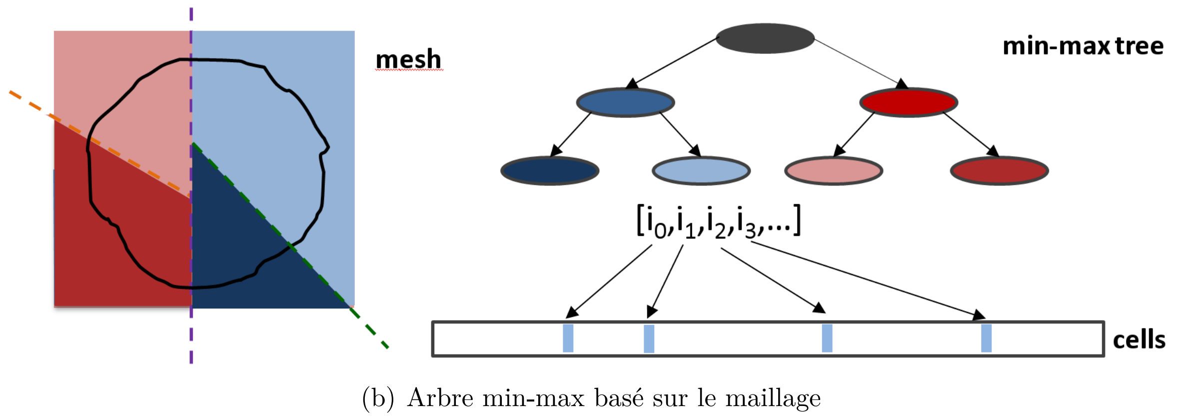
Il existe deux possibilités pour choisir la structure de l'arbre min-max.

– On peut choisir la structure de l'arbre min-max en se basant sur le maillage (*cf.* figure 6.2(b)(b)). On découpe le maillage en régions basées sur la géométrie (par exemple avec un *octree*). Comme les champs scalaires ont en général une

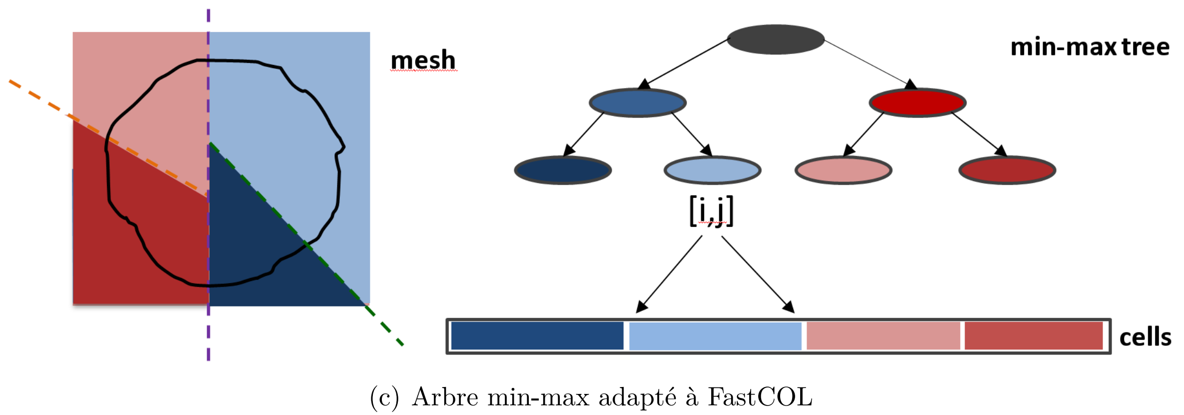
1. C'est l'option retenue par la bibliothèque VTK [SML04].



(a) Arbre min-max basé sur l'organisation mémoire



(b) Arbre min-max basé sur le maillage



(c) Arbre min-max adapté à FastCOL

FIGURE 6.2 – Trois possibilités de choix de l'arbre min-max. Notre proposition (c) combine les avantages de (a) et (b).

cohérence spatiale, c'est-à-dire le champ varie peu localement, les intervalles de valeurs scalaires associés aux nœuds sont de longueur réduite ce qui permet d'éliminer davantage de cellules. Cependant, comme les régions n'ont pas de lien avec l'organisation mémoire, les accès mémoires ont une mauvaise localité. De plus, il faut stocker pour chaque feuille de l'arbre, la liste des cellules correspondant à cette feuille ce qui augmente fortement l'espace mémoire nécessaire.
– On peut choisir la structure de l'arbre min-max en se basant sur l'organisation mémoire (*cf.* figure 6.2(a)(a)). On découpe le maillage en régions en se basant sur les indices des cellules. Dans ce cas, les cellules correspondant à une feuille de l'arbre sont un intervalle du tableau des cellules. Les accès mémoires sont donc locaux et la consommation mémoire est très fortement réduite. Cependant, les régions n'ont pas de lien avec la géométrie du maillage ce qui agrandit les intervalles de valeurs scalaires et réduit le nombre de cellules éliminées.

Nous proposons d'utiliser comme arbre min-max, l'arbre BSP construit par FastCOL lors du calcul de l'organisation mémoire (*cf.* figure 6.2(c)). De cette façon, les régions sont basées à la fois sur la géométrie du maillage et sur l'organisation mémoire ce qui permet de combiner les avantages des deux approches précédentes.

Parallélisation efficace en cache

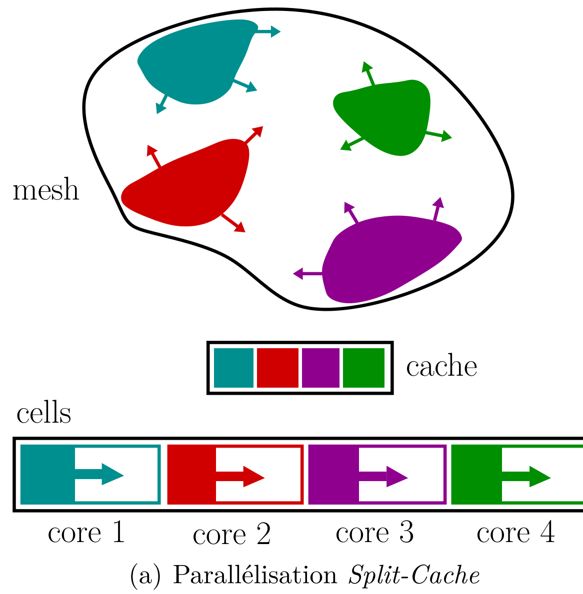
Le deuxième point que nous étudions dans cet article est l'impact de la parallélisation d'un filtre sur la localité de ses accès mémoires. Nous considérons un modèle multicœur avec un niveau de cache privé suivi d'un niveau de cache partagé (*cf.* figure 3.1 page 40).

Les filtres d'extraction d'isosurface MT et sa variante accélérée par un min-max tree sont assez faciles à paralléliser car le traitement de chaque cellule est indépendant des autres. Nous comparons deux parallélisations de ces filtres : *Split-Cache* et *Shared-Cache*. Ces algorithmes parallèles sont programmés avec une parallélisation statique en utilisant les threads.

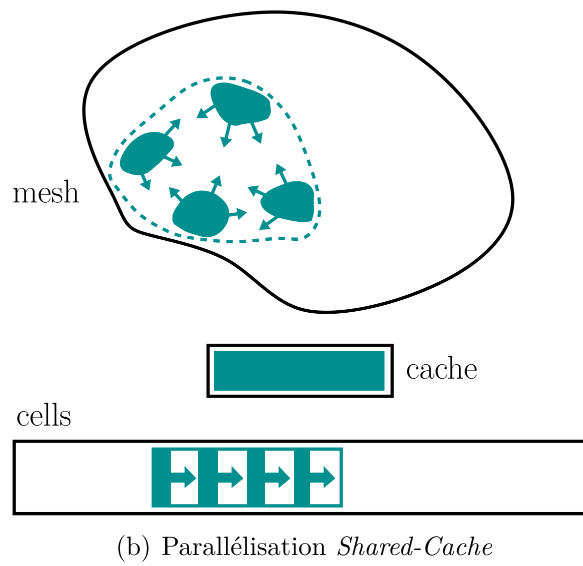
La parallélisation *Split-Cache* est une parallélisation standard (*cf.* figure 6.3(a)). Pour un maillage contenant n cellules à traiter sur p cœurs, on alloue à chaque thread n/p cellules. Cette parallélisation est efficace pour les caches privés car elle donne à chaque thread un sous ensemble contigu de données à traiter.

La parallélisation *Shared-Cache* est optimisée pour profiter à la fois des caches privés et du cache partagé (*cf.* figure 6.3(b)). Elle s'inspire de l'ordonnanceur Controlled-PDF présenté dans la section 4.4.2. On découpe implicitement les n cellules à traiter en n/m blocs de m cellules de manière à ce que les m cellules d'un bloc plus les points associés rentrent dans le cache partagé. On traite ces blocs de m cellules les uns après les autres (comme les super-tâches de type 2 dans Controlled-PDF). Chaque bloc est traité en parallèle par les p cœurs en allouant m/p cellules à chaque thread. Les threads se synchronisent à la fin du traitement de chaque bloc en utilisant une barrière `pthread_barrier`. Cette parallélisation est à la fois efficace pour les caches privés, les threads travaillant sur des sous ensembles contigus de données, et pour le cache partagé, l'exécution parallèle se comportant comme l'exécution séquentielle du point de vue du cache partagé. Par contre cette parallélisation nécessite plus de synchronisations que la parallélisation *Split-Cache*. Il faut en effet n/m barrières au lieu d'une seule.

Si la parallélisation *Split-Cache* est associée à l'organisation mémoire de FastCOL,



(a) Parallélisation *Split-Cache*



(b) Parallélisation *Shared-Cache*

FIGURE 6.3 – Deux schémas de parallélisations différents

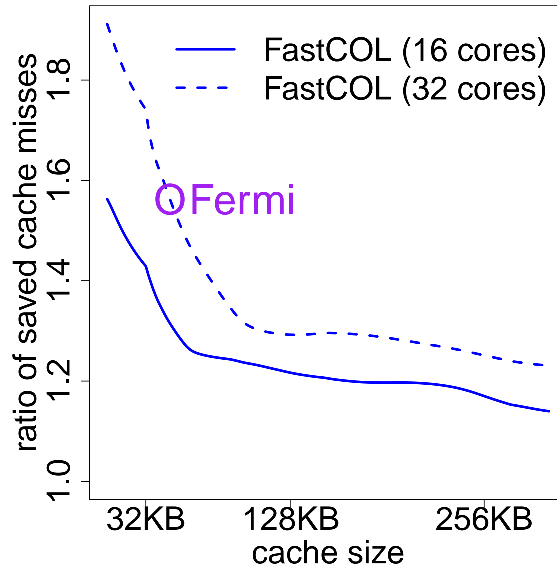


FIGURE 6.4 – Prédiction du gain de *shared-cache* par rapport à *split-cache* avec les paramètres de l'architecture du GPU Fermi de NVIDIA.

on peut obtenir une garantie sur le nombre de défauts de cache. En effet, la condition sur les accès mémoires du théorème 3 (*Chunk traversal*) de [TDR10a] est respectée. Le maillage est accédé par morceaux de taille M (avec M la taille du cache partagé), chaque morceau pouvant être traité dans un ordre quelconque. La parallélisation *Split-Cache* a la même garantie sur le nombre de défauts de cache que l'algorithme séquentiel.

Nous comparons les performances en pratique de *Split-Cache* et de *Shared-Cache* sur les deux filtres d'extraction d'isosurface MT et MT accéléré par arbre min-max sur 3 types de processeurs différents : l'un n'ayant que des caches privés (Opteron), les deux autres ayant un niveau (ou deux niveaux) de caches privés puis un cache partagé (Core2 et Nehalem). Sur une architecture à caches privés, le nombre de défauts de cache est similaire pour *Split-Cache* et *Shared-Cache*. *Split-Cache* est donc légèrement plus rapide grâce à un nombre plus faible de synchronisations. Sur les architectures à cache partagé, *Split-Cache* engendre beaucoup plus de défauts de cache que l'exécution séquentielle alors que *Shared-Cache* a un nombre de défauts de cache similaire à l'exécution séquentielle. Dans ce cas, la parallélisation *Shared-Cache* obtient de meilleures accélérations que *Split-Cache*. Cependant l'avantage de *Shared-Cache* est plus faible sur une organisation mémoire optimisée (que ce soit avec FastCOL ou un autre algorithme). En effet, avec une organisation mémoire optimisée le nombre de défauts de cache est déjà très faible.

Enfin, nous proposons un modèle basé sur les longueurs d'arêtes qui permet d'évaluer le gain de *Shared-Cache* par rapport à *Split-Cache*. Nous vérifions que ce modèle est fidèle au nombre de défauts de cache mesuré expérimentalement. En faisant varier les paramètres du modèle, nous montrons que le gain de *Shared-Cache* par rapport à *Split-Cache* augmente avec le nombre de cœurs partageant le même cache. La parallélisation *Shared-Cache* est donc très intéressante pour les futurs processeurs multicœurs qui auront de plus en plus de cœurs. Par exemple, on peut prédire l'avantage de *Shared-Cache* sur une architecture telle que le GPU Fermi de NVIDIA (*cf.* figure 6.4). Dans ce cadre, utiliser en combinaison *Shared-Cache* et FastCOL devrait apporter un gain de

performance important.

6.2 Discussion et perspectives

Optimisation fine des accès mémoires de MT

Nous avons continué à travailler sur l'algorithme MT pour encore améliorer ses performances en séquentiel. On peut d'abord remarquer que la soupe de triangles produite en sortie de l'algorithme n'est pas réutilisée dans l'algorithme et prend donc inutilement de la place dans le cache. En utilisant des instructions spéciales [2], on peut écrire cette soupe de triangles directement en mémoire sans passer par le cache. On peut aussi remarquer qu'il n'y a pas de réutilisation sur l'accès au tableau de cellules : chaque cellule n'est utilisée qu'une fois. On aimerait donc qu'une cellule soit supprimée du cache juste après avoir été utilisée. Il n'est pas possible de modifier l'algorithme de remplacement du cache d'un processeur. Pour réaliser cette optimisation, nous avons été amené à développer un outil qui modifie la façon dont les adresses virtuelles sont associées aux adresses physiques dans le but d'isoler dans une petite partie du cache le tableau de cellules. Cela permet de laisser une plus grande partie du cache au tableau des points qui est réutilisé plusieurs fois. Cette méthode d'optimisation du cache, c'est-à-dire adapter la taille du cache allouée à une structure de données en fonction de son histogramme de distances de réutilisation (présenté en section 1.2.4), est générale et a été appliquée à d'autres applications. Ce travail a été réalisé en collaboration avec Swann Perarnau et Guillaume Huard de l'équipe MOAIS et est actuellement en soumission. En utilisant ces optimisations, on accélère l'algorithme de MT d'un facteur 1.33 grâce à une réduction du nombre de défauts de cache d'un facteur 1.57.

Adaptation d'autres structures de données

L'arbre min-max que nous proposons dans ce chapitre est adapté à l'organisation mémoire du maillage et permet donc une amélioration de la localité des filtres ayant un schéma d'accès global de type *external data structure traversal*. La méthode que nous proposons ici, c'est-à-dire utiliser la structure de l'arbre de BSP produit par FastCOL, ne s'adapte a priori qu'aux structures d'arbres.

Une autre structure de données qui pourrait être adaptée à l'organisation mémoire du maillage est la pile. Une structure de pile est souvent utilisée lors d'un parcours de graphe. Quoique moins fréquent que les structures d'arbres, ce schéma d'accès se retrouve dans deux filtres de visualisation importants :

- l'extraction d'isosurfaces en utilisant la méthode *seed set and propagation* [HB94, IK95, vKvOB+97],
- le tri de visibilité [Wil92] (utilisé, par exemple, pour un rendu volumique par projection de tétraèdres [ST90]).

Dans ces deux filtres, on définit une relation binaire sur les cellules du maillage à partir des relations de voisinages entre cellules et d'autres conditions (valeurs scalaires pour la méthode *seed set and propagation*, relation de visibilité pour le tri de visibilité).

2. Instructions *non temporal write* du jeu d'instructions SSE

On parcourt ensuite le graphe dirigé défini par les cellules du maillage et les arcs de la relation binaire. Dans ces deux cas on pourrait adapter la pile en modifiant sa sémantique habituelle (dernier arrivé premier sorti ou LIFO), pour que la séquence des cellules extraites de la pile offre une bonne localité mémoire. La difficulté ici est d'arriver à trouver une implémentation de cette pile adaptée au maillage qui nécessite peu de calculs pour ne pas contrebalancer le gain en défauts de cache.

Une autre possibilité d'une telle adaptation est le cas des maillages multi résolutions. Si l'on dispose de plusieurs résolutions du même maillage, par exemple comme dans [SB09], on peut utiliser le même arbre de BSP pour toutes les résolutions. Outre le gain en temps lors du calcul de la réorganisation, on peut observer un gain en localité lorsque les accès mémoires passent de résolution en résolution. Une telle approche a été expérimentée dans le code de rendu volumique de Sébastien Barbier [SB09] mais les coûts en calcul étaient trop importants et masquaient les gains en défauts de cache.

De manière générale, il est intéressant d'étudier la cohérence en mémoire de plusieurs structures de données entre elles.

Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores

M. Tchiboukdjian†[1] and V. Danjean‡[2] and B. Raffin§[3]

[1] CNRS - CEA/DAM,DIF
[2] Grenoble Universites
[3] INRIA

Abstract

This paper proposes to revisit isosurface extraction algorithms taking into consideration two specific aspects of recent multicore architectures: their intrinsic parallelism associated with the presence of multiple computing cores and their cache hierarchy that often includes private caches as well as caches shared between all cores. Taking advantage of these shared caches require adapting the parallelization scheme to make the core collaborate on cache usage and not compete for it, which can impair performance. We propose to have cores working on independent but close data sets that can all fit in the shared cache. We propose two shared cache aware parallel isosurface algorithms, one based on marching tetrahedra, and one using a min-max tree as acceleration data structure. We theoretically prove that in both cases the number of cache misses is the same as for the sequential algorithm for the same cache size. The algorithms are based on the FastCOL cache-oblivious data layout for irregular meshes. The CO layout also enables to build a very compact min-max tree that leads to a reduced number of cache misses. Experiments confirm the interest of these shared cache aware isosurface algorithms, the performance gain increasing as the shared cache size to core number ratio decreases.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Software]: Concurrent Programming—Parallel Programming I.3.3 [Computer Graphics]: Picture/Image Generation—Isosurface computation I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Isosurface extraction is one of the most classical filters for scientific visualization. It has been intensively studied and various algorithms exist with different acceleration data structures and parallelizations.

In this paper, we focus on one specific aspect of its parallelization that has not been addressed so far: how to adapt the algorithm to take advantage of the shared caches often present on multicore processors. The goal is to propose an algorithm that saves cache misses, thus improving performance, compared to parallel algorithms that do not take into account the shared cache amongst several cores.

Multicore architectures usually have their last cache level shared between cores. For instance the L3 cache of the Intel Nehalem, the L2 cache of the Intel Larrabee or the L1 cache of NVIDIA Fermi processors are shared. Compared to private caches, this shared cache architecture can bring performance benefits if managed adequately. It allows fast communication between cores. If some cores work on the same data, these data are not duplicated into several caches. A core can potentially use more than its fraction of the cache if necessary. But this requires the algorithms to be adapted to make the cores collaborate on cache usage. Classical parallelization approaches usually favor tasks working on independent data sets to reduce communication and synchronization overhead. It results in competition rather than collaboration between cores for shared cache usage. Performance is at most equivalent to a private cache configuration. Indeed, [Has10] shows that this is actually worse than with pri-

† marc.tchiboukdjian@imag.fr
‡ vincent.danjean@imag.fr
§ bruno.raffin@imag.fr

© The Eurographics Association 2010.

vate caches as the LRU replacement policy performs poorly in this context.

Many scientific visualization filters, like isosurface extraction, are memory bounded. Favoring the locality of access patterns through an adapted data layout can bring significant performance benefits. In this paper we propose to have cores working on independent but close (regarding the memory layout and spatial locality) data sets that can all fit in the shared cache. If a core needs a data that is not in its data set, there is a good chance it will find it in the data set loaded in the cache by one of its neighbors, thus saving cache misses. We propose two versions of this isosurface parallel algorithm, one based on the marching tetrahedra (MT), and one using a min-max tree as acceleration data structure. We theoretically prove that in both cases the number of cache misses is the same as for the sequential algorithm using a cache of the same size. This is like if each core would benefit from a full size private cache, at the price of a few extra synchronizations required to ensure a proper collaboration between cores. The algorithm is based on the cache oblivious (CO) data layout for irregular meshes proposed in [TDR10]. Not only it ensures a strong data locality, but, in opposite to other layouts, it also provides a theoretical bound on the number of cache misses. Our proof relies on an extension of this result to our parallel isosurface extraction algorithms.

Experiments confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs. It also show that a CO layout is not necessarily required, as other classical layouts can lead to a high enough data locality given the high cache to core ratio available on the tested processors.

We also detail a compact and cache-efficient tree structure for accelerating the MT algorithm. This tree is a min-max tree using a decomposition of the mesh in regions adapted to the CO mesh layout. It allows a compact storage as regions correspond to intervals of the cell array. It also induces very few caches misses as the active cells respect the order of the CO layout.

The paper is organized as follow. The MT algorithm is reviewed in section 2. The sequential CO algorithm is presented and proved in section 3 before to be extended to the parallel context in section 4. Experiments are detailed in section 5. Related works are discussed in section 6 before the conclusions.

2. Marching Tetrahedra Review

We review the data access patterns associated with the MT isosurface extraction algorithm and its tree accelerated versions.

Mesh Data Structure. A mesh data structure usually consists of two multidimensional arrays: an array storing point

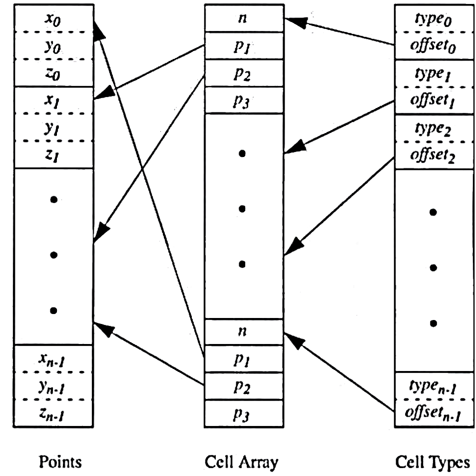


Figure 1: *The vtkUnstructuredGrid data structure (from the VTK Textbook [SML04]). The Points array contains point coordinates and the Cells array contains the indices of cell points. The Cell Types array contains the type of each cell and provides $O(1)$ random access to cells.*

attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). When the mesh is composed of cells of different types (using various number of points), an additional array allows random access to cells (Fig. 1). As the cache performance for meshes having identical or not type cells is similar, we focus on homogeneous meshes in this paper.

MT Algorithm. For one cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a linear approximation of the isosurface going through this cell. Applied on all mesh cells sequentially, it leads to a cost linear in the number of cells.

Tree Accelerated MT. The MT algorithm can be accelerated with various data structures allowing to efficiently search for the cells intersected by the isosurface. One such data structure is the min-max tree [WVG92]. An octree where each node stores the minimum and maximum values of its subtrees allows to quickly discard parts of the mesh that do not contain any intersected cell. The search is thus improved from $O(n)$ to $O(k + k\log n/k)$ where n is the number of cells and k the size of the isosurface (usually $k \ll n$). If the scalar field is spatially coherent, the performance is actually improved over this theoretical bound as large subtrees can be pruned.

Several kinds of min-max trees can be used. Octrees, kd-trees or more generally Binary Space Partitioning (BSP) trees recursively decompose the mesh into regions. The idea is that the scalar field does not vary too much in each re-

© The Eurographics Association 2010.

gion and thus the extreme of the scalar field form a small interval, less likely to contain the isovalue. Contrary to these geometric decompositions, the vtkSimpleTree of the VTK library [SML04] uses a layout decomposition. The regions consist of intervals of indices in the cell array. This tree is faster to compute and less memory consuming as regions are implicitly defined. However, depending on the cell layout, the scalar field may vary a lot in each region as they are not based on the geometry.

There exists an optimal data structure which is not based on the min-max tree. The interval tree [CMPS96] stores for each cell c the interval whose extremes are the minimum and maximum value of the points of c. The query time is improved to $O(\log n + k)$ whatever the spatial repartition of the scalar field is. The interval tree has been made I/O-efficient allowing a query with complexity $O(\log_B n + k/B)$, where B is the block size. This is optimal [CS97]. However this approach is not space-efficient since vertex information is duplicated many times. The 2-level indexing scheme based on the meta-cells technique introduced in [CSS98, CS99] improves over the interval tree in term of space usage but some data remain duplicated. Spatially close cells are grouped into meta-cells, which are then used in the I/O-efficient interval tree.

The tree accelerated MT, based on the min-max tree or the interval tree, improves performance over the regular MT at the cost of a higher memory requirement [SHwSS00]. Moreover, if the tree structure depends on the scalar field, the tree has to be stored for each scalar field. In this paper, we propose a min-max tree that is very compact and whose structure is the same for all scalar fields.

3. Cache-Efficient Isosurface Extraction

We now look at cache misses induced by sequential MT algorithms. After a general discussion, we focus on meshes stored according to the CO layout introduced in [TDR10] and give a theoretical guarantee of cache performance for a MT algorithm and a min-max tree accelerated.

3.1. Source of Cache Misses in MT

The cache misses in the MT algorithm come from accessing the cell array, accessing the point array, and accessing the min-max tree for the tree accelerated variant. In the regular MT algorithm, the cell array is traversed once in order, i.e. from low indices to high indices. Thus it induces only compulsory cache misses, corresponding to a first access. The point array is not accessed in a regular order like the cell array. Points are accessed by following a reference from the cell array, e.g. read coordinates of a point. This can induce capacity misses if the same point is needed by cells far away in the cell array. It is likely that the cache line containing the point will be evicted from the cache between these two accesses. In this case, the layout does not exhibit a good temporal locality. Moreover, not all data stored in a cache line will be use before being evicted; for example when a cache line stores data for two different points that are needed by cells far away in the cell array. In this case the layout lacks of spatial locality. For the tree accelerated MT, the cell array is not necessarily accessed in order like for the regular MT, which could lead to extra cache misses.

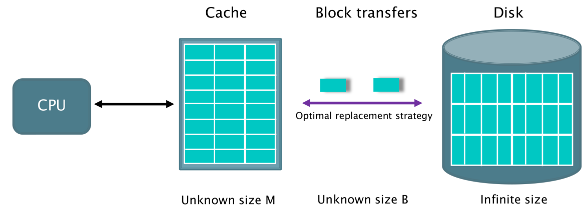


Figure 2: *The cache-oblivious memory model. The data are transferred by block of B consecutive elements into a cache of size M. Both parameters are unknown to the algorithm.*

The number of cache misses heavily depends on the mesh layout, i.e. how cells and points are sorted and stored in memory. For example, a point layout can improve cache performances if points corresponding to the same cell are stored nearby. These points may share cache lines, which increases spatial locality. Also, a cell layout can improve cache performance if the cells that are accessed consecutively by the min-max tree are stored nearby. Two consecutive active cells could be in the same cache line, increasing spatial locality.

As a mesh often includes several scalar fields, we consider here only optimizations of the layout that do not depend on the scalar field value. Thus these optimizations are efficient for all scalar fields.

3.2. Cache-Oblivious Model (CO)

We now introduce the cache-oblivious model from [FLPR99] we rely on to theoretically measure the number of cache misses. The memory hierarchy consists of two levels, a fast memory of size M called cache and an infinite size slow memory. The data are transferred between these two levels in blocks of B consecutive elements (Fig. 2). The cache performance of an algorithm is the number of block transfers needed to complete the computation. Parameters B and M are unknown to the algorithm to forbid tuning for a specific architecture. A good CO algorithm, i.e. one that performs well in the CO model, is thus expected to be cache-efficient whatever the cache and block size are.

3.3. CO Mesh Layout

In [TDR10], the authors introduce a CO layout algorithm for irregular meshes with a theoretical performance guarantee. It relies on a recursive mesh partitioning using a specific

© The Eurographics Association 2010.

BSP algorithm. This algorithm cuts the mesh guaranteeing a good tradeoff between minimizing the number of cut elements and having two partitions of similar size. When applied recursively, it ensures that spatially close and strongly connected data tend to be partitioned deeper in the BSP tree. The CO layout is obtained by storing the data linearly in memory from the first leaf of the BSP tree to the last one. The data loaded in a cache block are thus contiguous leaves of the BSP tree. It is cache-oblivious as to any block and cache size corresponds a BSP tree depth level. This ensures a strong locality and connectivity.

This CO layout algorithm has several benefits. Computing the layout is fast (complexity of $O(n \log n)$). When traversing the layout the cache-complexity is guaranteed not only for a strict layout consistent access order but also for a chunk based access. Data can be accessed by chunks of m consecutive elements in the layout, to be processed (in any order) before accessing another chunk anywhere in the layout.

Theorem 1 (Chunk traversal from [TDR10])
The CO layout guarantees that a traversal by chunks of size $m \leq M$ of an N-size mesh induces less than $N/B + O(N/m^{1/3})$ cache misses where B and M are the block and cache size, respectively.

3.4. MT Cache Performance

Using the cache-oblivious layout of the previous section, one can guarantee that the marching tetrahedra algorithm induces less that $O(n/B + n/M^{1/3})$ cache misses where B and M are the block size and the cache size. Indeed, the MT algorithm processes the mesh in order and thus by chunks of size M.

3.5. Tree Accelerated MT Cache Performance

We consider here a specific min-max tree, the one based on the BSP tree partitioning the mesh for computing the CO layout. We use this BSP tree because each node corresponds to a sub part of the mesh stored sequentially in memory. We thus get a min-max tree that is layout friendly. When traversing the BSP tree in prefix order and examining the mesh cells that might contain a part of the isosurface, mesh cells are accessed sequentially. Contiguous mesh cells can be skipped (pruned by the min-max tree), but we will never go backward. We can expect to save cache misses. This tree accelerated MT algorithm has been introduced in [TDR10]. Here, we also state its performance given the additional hypothesis that the scalar field is spatially coherent, i.e. for regions of n cells in the mesh the isosurface intersects on average at least $n^{2/3}$ cells for each region. A similar hypothesis is used in [CS99]. Using theorem 1, one can show that the cache performance of this tree accelerated MT algorithm is $O(k/B^{2/3} + k/M^{1/3})$ where k is the number of active cells.

4. Parallel Cache-Efficiency

We now introduce a parallel MT algorithm that is equivalent to the sequential one regarding cache performance.

4.1. Shared Cache Multicore

Most last generation multicores share a similar design for the cache hierarchy. Each core has its own private caches while the last cache level is shared between all cores. For instance the Intel Nehalem, the AMD Phenom and the IBM Power7 all have a shared L3 cache. Coming GPU architectures also adopt this cache design. The Intel Larrabee has a shared L2 cache. The NVIDIA Fermi has a L1 cache that is shared for all stream processors in the same multiprocessors and a L2 cache that is shared across all multiprocessors. They are many advantages of the shared cache compared to private caches. It allows fast communication between cores. If some cores work on the same data, these data are not duplicated into several caches. A core can potentially use more than its fraction of the cache if necessary. But this requires the algorithms to be adapted to make the cores collaborate on cache usage. Classical parallelization approaches that are not shared cache aware lead to competition for shared cache. Performance, at most equivalent to a private cache configuration, is actually impaired as the LRU replacement policy performs poorly in this context [Has10].

4.2. Shared Cache Aware Parallelization

We now present a new parallelization scheme that guarantees that the cache performance of the underlying sequential application is not reduced. The idea is to have cores working on close data so that each core has the impression to own the totality of the shared cache. Cache misses of one core profit to other cores as they are likely to also need these data in a near future. But cores should not work on data that are too close either because this could cause bad private cache behaviors.

The parallel algorithm is based on the sequential execution order $i_0, i_1, \ldots, i_p$. We assume that the sequential algorithm has good locality and thus data that are processed closely in the sequential execution are also close in memory. Informally let i_m be the first instruction whose processing would need to evict from the shared cache data needed by instruction i_0. To keep the cache performance of the sequential algorithm, the parallel scheme will deviate from the sequential order at most for m instructions. That is, instruction i_k can be processed only when instructions i_1 to i_{k-m} have been completed. This way, data evicted when processing i_k do not affect the processing of the other instructions. Moreover, as the cores work on instructions close in the sequential order, they work on close data and thus can profit of other cores cache misses.

© The Eurographics Association 2010.

4.3. Parallel MT

We now apply this parallel scheme to the MT algorithm. Let m be the maximal size, in number of cells, such that the corresponding region of the mesh fits entirely in the shared cache. It corresponds to the cells of the largest subtree of the CO layout BSP tree that fit in the last level of cache. To process all cells in parallel, we divide the n cells into n/m chunks of size m and then process each chunk in parallel. Processing a new chunk is started only when the previous one has been entirely processed. To process one chunk, we divide the m cells into m/p groups, one for each core. From theorem 1, we know that a chunk can be processed in any order without affecting the cache performance, thus the number of cache misses of the sequential algorithm 3.4 is still valid for this parallel algorithm.

Let compare this shared cache aware parallelization with a standard parallelization. The trivial way to process n cells in parallel is to divide the cells in groups of n/p, one for each core. Let assume now that the shared cache behaves as well as p private caches (it should be worse in practice). This parallelization yields at best on each core the performance of the sequential algorithm on a cache of size M/p, i.e.

$$O\left(\frac{n/p}{B}+\frac{n/p}{(M/p)^{1/3}}\right).$$

Thus, the total number of cache misses is

$$O\left(\frac{n}{B}+p^{1/3}\cdot\frac{n}{M^{1/3}}\right).$$

This is a factor of $p^{1/3}$ worse than the shared cache aware parallelization, which induces the same number of cache misses as the sequential algorithm, i.e. $O(n/B+n/M^{1/3})$.

However, the shared cache aware parallelization has much more global synchronizations, where all cores wait for the last one to finish. There is one synchronization per chunk, n/m compared to only 1 for the standard parallelization. We show in the experiments that this additional cost of synchronization does not impair too much the performance of the shared cache aware parallelization.

4.4. Parallel Tree Accelerated MT

We now apply the shared cache aware parallelization to the MT accelerated with a min-max tree. We use as a min-max tree the BSP tree built with the layout. Let consider nodes of the BSP tree that correspond to a region of the mesh fitting in the shared cache. By theorem 1, we know that each of these nodes can be processed in any order without affecting the cache performance. Thus each node of the BSP tree can be processed in parallel by the p processors. A new node starts being processed only when the previous one is terminated. The cache complexity of the sequential algorithm of section 3.5 is still valid for this parallel algorithm. The overhead is the same as for the classic parallel version, i.e. $O(p^{1/3})$. However the shared cache aware parallelization also has more synchronizations, one per node of the BSP tree.

5. Implementation and Experiments

5.1. Architectures and Meshes

We took 4 different meshes (Blunt fin, buckyball, liquid oxygen post, plasma64 from the AIM@SHAPE Shape Repository (http://shapes.aim-at-shape.net/), processed to generate instances of 150M cells. We used tetgen (available at http://tetgen.berlios.de/) to refine the meshes by adding volume constraints to each tetrahedron (we used the command `tetgen -raq`). For each mesh, we generated two finer meshes. In the first one, all tetrahedra have approximately the same volume. In the second one, we used a volume constraint proportional to the inverse of the gradient of the scalar field to mimic adaptive mesh refinement. It leads to a set of 8 meshes with approximately 150M cells each.

For each mesh, 3 layouts are compared. The original layout as downloaded from the web, the geometric layout where points and cells are sorted according to x,y,z coordinates using lexicographic order, and the CO layout generated by FastCOL [TDR10].

The experiments were conducted on three different architectures. Two multicores (Intel Core2 E6750 @ 2.66Ghz, dual core, private cache L1 32KB, shared cache L2 4MB and Intel Xeon E5530 @ 2.4Ghz, quad core, private cache L1 32KB and L2 256K, shared cache L3 8MB) with a shared last level of cache and one multicore with only private caches (AMD Opteron 875 @ 2.2Ghz, dual core, private cache L1 8KB, private cache L2 1MB).

We measured the execution time and the number of L1, L2 and L3 cache misses using the PAPI software [BDG*00]. For each experiment (architecture, layout and algorithm fixed), the execution time and the numbers of cache misses are very stable. Results of the experiments are the median of 10 runs.

5.2. Min-Max Tree Implementation

We first describe the data structure used to store a min-max tree, whatever the cell layout is. Each node of the tree corresponds to a region of the mesh and contains the minimum and maximum value of the scalar field in this region. Those nodes are stored linearly in memory like a pointerless binary tree. A more efficient layout could be used, like the van Emde Boas layout of the cache-oblivious B-tree [ABF04]. We used a simple layout as the tree is small and the tree traversal is only a small part of the computation (less than 10%). For each leaf of the tree, we need the cells lying in the corresponding region of the mesh. A simple way is to store in each leaf node a list of cells, which results in a complex and inefficient storage. Instead we store a permutation

© The Eurographics Association 2010.

π of cells indices such that to each node of the tree corresponds an interval of cells. This way, each leaf contains only two cell indices i and j. To find the cells laying in a region, we apply the permutation to each cell indices in the interval: $\pi(i),\ldots,\pi(j)$. Thus the data structure contains the tree and the permutation.

In the case of a min-max tree based on the CO layout and its associated BSP tree (Section 3.5), the cell permutation is simply the identity. Storing the permutation is no longer needed, leading to a very compact data structure.

The VTK binary tree vtkSimpleTree [SML04] can be used to accelerate a MT algorithm. It is based on a recursive decomposition of the layout of the mesh. This tree does not need to store a cell permutation either but the regions it defines are not based on geometry and thus could map together very distant cells. It may not be as efficient when pruning active regions.

The BSP tree associated with the CO layout is both based on the layout and on the geometry. It is compact, can efficiently discard inactive regions and induces a mesh traversal with few cache misses.

We compare our min-max tree to the compact interval tree of [WJV07]. To our knowledge, it is the most compact implementation of the I/O efficient interval tree for isosurface extraction. To index a 25M cells mesh decomposed into 9x9x9 metacells, they need between 200MB and 250MB, a factor 2 improvement over the standard I/O efficient interval tree of [CS97]. In comparison, for a mesh of 150M cells, a min-max tree using our permutation based storage needs 579MB, only 3 times bigger for a 6 times bigger mesh. With the CO layout the permutation is not stored and the tree only requires 6MB, which is several orders of magnitude smaller than trees reported in the literature. The smallest tree reported in [SHwSS00], the branch-on-need octree, requires roughly 100MB for a 16M cell mesh. In the experiments we used 2x2x2 metacells (8 cells per leaf) leading to a space requirement of 958MB for the standard min-max tree and 385MB for the CO version, 36% and 14%, respectively, of the size of the considered 150M cells mesh (one scalar field).

5.3. Sequential Performance

Table 1 shows the sequential performance of the MT algorithm and the tree accelerated MT algorithm on various layouts and architectures.

For the MT algorithm, the geometric layout outperforms both in time and cache misses the original layout. The cache oblivious layout shows the best performance with an improvement of a factor between 1.5 and 2 over the original layout on all architectures.

For each layout, the tree accelerated MT algorithm is always faster than the regular MT algorithm. However, for the tree accelerated MT algorithm, the geometric layout does not improve performance over the original layout. The geometric and the original layout use the same kd-tree so the difference is not due to a smallest number of active cells but only to a better cache behavior. As both the layout and the kd-tree are based on the geometry, it is possible that the tree accelerated algorithm accesses the mesh less efficiently. The CO layout with the adapted min-max tree shows the best performance with very few cache misses. The reduction in cache misses over the original layout with the kd-tree is between 3 and 5. This is impressive as the tree accelerated algorithm already induces few cache misses. This results in time speedup of 2.

5.4. Parallel MT Implementation

We present here the implementation of the two parallel MT algorithms. The standard parallelization that acts as if the shared cache was split into p private caches is denoted *split cache*, while the shared cache aware parallelization is called *shared cache*. We used `pthread` to parallelize all algorithms as this allows a fine grain control on synchronizations and to reduce parallelism overhead over high level parallel libraries like OpenMP.

For the split cache parallel MT algorithm, the array of cells is statically divided into p groups. Each group is assigned to one thread and all threads synchronize at the end of the computation.

For the shared cache version, the array of cells is first divided into chunks as large as possible but that can fit in the shared cache. Then each chunk is statically divided into p groups and all threads synchronize at the end of each chunk before starting to compute the next one. Each synchronization is implemented with a `pthread_barrier`. Threads wait at the barrier and are released when all of them have reached the barrier. Compared to the split cache version where there is only one synchronization at the end of the computation, this version has n/m synchronizations, one per chunk. So we expect the threads to spend more time waiting for other threads to finish their work. Nevertheless, the shared cache version keeps all threads working on data close in mesh space and thus in memory, which should result in less cache misses. We show in the next section that the shared cache version outperforms the split cache version. The reduction of cache misses more than counterbalance the augmentation of waiting times due to an increased number of synchronizations.

5.5. Parallel MT Performance

Table 2 (top) compares the performance of the two parallel MT algorithms on a multicore with only private caches and two multicores with a shared last level of cache. Both algorithms execute the same number of instructions. The very good speedups on the Opteron suggest that the work load is well balanced. We also examined the number of instructions

© The Eurographics Association 2010.

Table 1: *Sequential performance of the MT algorithm and the tree accelerated MT (TA) on Opteron, Core2 and Nehalem. Only one core is used. For the original and geometric layout, the min-max tree is a kd-tree. For the CO layout the associated BSP tree is used. Times are in ms. Cache misses for all levels of caches (L1, L2 and L3) are in millions.*

		Opteron			Core2			Nehalem		
		Time	L1	L2	Time	L1	L2	Time	L2	L3
MT	Original	10800	155.2	48.2	2940	190.2	59.4	3600	190.2	54.0
	Geometric	6200	122.8	25.9	2120	141.6	49.8	2600	216.4	53.2
	CO	5250	45.5	14.7	1935	47.8	41.2	2280	91.5	7.6
TA	Original	1650	16.8	13.0	970	29.3	21.3	855	23.8	16.8
	Geometric	2890	25.9	22.0	1385	52.6	40.4	1300	44.4	23.8
	CO	690	5.0	4.0	565	5.4	4.4	415	8.2	3.5

executed by each thread and they are very close. Thus the performance difference observed between both algorithms is mainly based on the cache behavior.

Both algorithms scale very well on the Opteron. We obtain a speedup close to 2 for the original and geometric layouts. The CO layout shows a super linear speedup due to a very low number of cache misses for the parallel algorithms, 2 times less cache misses than the sequential algorithm. However we are not able to explain such a reduction of cache misses. The two parallel schemes perform equally well with a very close number of cache misses for both levels of cache L1 and L2. This was expected as there is no shared level of cache.

For the Core2 and Nehalem processors, the parallel algorithms do not scale as well as on the Opteron (2 threads run on the Core2 and 4 on the Nehalem). Indeed, the parallel algorithms do not have more cache than the sequential algorithm. Similarly to the Opteron, the number of private cache misses are very close for the two parallel schemes. As expected, the split cache version induces more cache misses than the sequential algorithm on the shared cache. The shared cache version keeps a number of cache misses very close to the one of the sequential version and thus is faster than the split cache version almost all the time for the original and geometric layout. Performances are similar for the CO layout due to the very low number of cache misses of the sequential algorithm (behavior analyzed in section 5.8) .

5.6. Parallel Min-Max Tree Implementation

In our implementation of the parallel min-max tree algorithm, only the generation of triangles is performed in parallel and not the tree traversal to select active cells. As the active cells selection phase represents only 7% of the sequential time, we expect the algorithms to still scale well. In the split cache version, the active leaves are statically divided into p groups, one per thread. Each thread processes its group of cells and all of them synchronize at the end. In the shared cache version, the min-max tree is first divided into nodes that correspond to regions of the mesh that fit in cache. Then actives leaves of each region are distributed to the threads and processed in parallel. Threads synchronize at the end of each region using a `pthread_barrier` before starting processing the next region. Like for the MT algorithm, the shared cache aware version uses more synchronizations to stay close to the sequential order but it leads to less cache misses compared to the split cache version.

5.7. Parallel Min-Max Tree Performance

Table 2 (bottom) compares the performance of the two parallel tree accelerated MTs. The behavior of the tree accelerated version is similar to the regular MT. Both schemes perform equally well on Opteron and the shared cache version performs better on the Core2 and the Nehalem. However in this case, the shared cache version still offers some improvement over the split cache version with the CO layout.

5.8. Measure of Locality

To better analyze the properties of the different layouts, we analytically relate the performance improvements to the better data locality in memory. We call "edge length" the memory gap between two vertices of the same edge in the vertex array loaded in memory. If a mesh has shorter edges, more of them will fit in cache and a better performance should be observed. Figure 3 shows that the CO layout favors smaller edge lengths than the two other layouts.

We now estimate the number of cache misses using an edge length based metric. Let N be the size of a mesh (in bytes), E the set of all edges of the mesh, B the cache line size and M the cache size, we estimate the number of cache misses by:

$$CM_{\text{seq}} \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M}$$

where λ_e is the length of the edge e. We count the number of cache misses for a linear full read of the data arrays and we add one cache miss per edge whose length is bigger than the cache size M.

© The Eurographics Association 2010.

Table 2: *Performance comparisons on three different processors, for three different layouts original (Ori.), geometric (Geo.) and cache-oblivious (CO) of the sequential MT algorithm, the split cache and shared cache parallel MT algorithms (top), or the tree accelerated sequential MT algorithm, the split cache and shared cache tree accelerated parallel MT algorithms (bottom). Hyperthreading is disabled on Intel processors. The parallel algorithms are executed with 2 threads on the Opteron and Core2 processors, and with 4 threads on the Nehalem. Cache misses are in millions. Speedups are relative to the sequential algorithm on the same layout (Seq. speedup) or relative to the sequential algorithm on the original layout (Orig. speedup).*

			Opteron				Core2				Nehalem			
			Speedup		Cache Misses		Speedup		Cache Misses		Speedup		Cache Misses	
			Seq.	Orig.	L1	L2	Seq.	Orig.	L1	L2	Seq.	Orig.	L2	L3
MT	Orig.	Sequential	1.00		155.2	48.2	1.00		190.2	59.4	1.00		190.2	54.0
		Split Cache	2.07		155.2	42.1	1.43		189.9	75.0	2.83		190.1	70.0
		Shared Cache	1.99		155.3	44.0	1.72		188.3	57.8	3.36		191.1	55.2
	Geo.	Sequential	1.00	1.74	122.8	25.9	1.00	1.39	141.6	49.8	1.00	1.38	216.4	53.2
		Split Cache	1.98	3.45	122.8	21.1	1.59	2.21	140.9	59.0	3.02	4.19	217.8	68.5
		Shared Cache	1.98	3.45	122.8	21.4	1.78	2.47	140.2	48.6	3.31	4.59	220.2	56.9
	CO	Sequential	1.00	2.06	45.5	14.7	1.00	1.52	47.8	41.2	1.00	1.58	91.5	7.6
		Split Cache	2.69	5.53	45.5	7.0	1.74	2.65	47.6	41.3	3.10	4.90	92.3	8.9
		Shared Cache	2.56	5.27	45.6	6.6	1.71	2.60	47.6	41.1	3.06	4.83	93.5	8.7
Tree Accelerated MT	Orig.	Sequential	1.00		16.8	13.1	1.00		29.3	21.3	1.00		23.8	16.8
		Split Cache	1.55		16.6	13.1	1.42		28.9	22.1	2.59		22.9	17.9
		Shared Cache	1.54		16.6	13.1	1.60		29.1	21.1	2.85		22.8	16.7
	Geo.	Sequential	1.00	0.57	25.9	22.0	1.00	0.70	52.6	40.4	1.00	0.66	44.4	23.8
		Split Cache	1.68	0.96	25.8	22.0	1.56	1.09	52.5	41.6	2.89	1.90	43.8	25.9
		Shared Cache	1.64	0.93	25.6	22.0	1.73	1.21	52.5	40.2	3.02	1.99	43.9	24.6
	CO	Sequential	1.00	2.39	5.0	4.4	1.00	1.72	5.4	4.4	1.00	2.06	8.2	3.5
		Split Cache	1.31	3.14	4.5	3.8	1.43	2.46	5.1	5.1	2.08	4.28	7.3	3.7
		Shared Cache	1.33	3.17	4.5	3.7	1.53	2.62	5.3	3.6	2.18	4.50	7.3	3.4

For the split cache version, we could think of each core having a private cache of size M/p instead of a shared cache of size M, which gives an expected number of cache misses of

$$CM_{\text{split}} \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M/p}$$

As the shared cache version has the same cache performance as the sequential algorithm, we have

$$\frac{CM_{\text{shared}} - \frac{N}{B}}{CM_{\text{split}} - \frac{N}{B}} \approx \frac{\sum_{e \in E} \mathbb{1}_{\lambda_e > M}}{\sum_{e \in E} \mathbb{1}_{\lambda_e > M/p}}$$

Let check this formula for Core2 (Table 2). In our experiments, we measured that a linear read of the mesh induces 37.6 millions of L2 cache misses so we have $N/B = 37.6$. On the original layout, the experiments give

$$\frac{CM_{\text{shared}} - \frac{N}{B}}{CM_{\text{split}} - \frac{N}{B}} = \frac{75.0 - 37.6}{57.8 - 37.6} \approx 1.85\,,$$

which is close to the value of 1.90 found with the edge length metric. A similar calculation using the experiments and the geometric layout gives 1.95, with an edge length metric of 1.95. On Nehalem we measured 2.45 for the original layout and 2.00 for the geometric layout, close to the edge length metric values of 2.30 and 2.35. For the CO layout, both the measured cache misses and the edge length values are close to 1. This is consistent with the split cache and shared cache algorithms inducing approximately the same number of cache misses.

This reasoning allows to extrapolate the gain in cache misses of the shared cache parallelization over the split cache parallelization for various cache sizes. Figure 4 presents expected gain for all three layouts on various shared cache sizes. One can remark 3 things. The more cache is available, the less speedup is to be expected. The more cores share the same cache, the more speedup is to be expected. If the sequential application has already very few cache misses, then it performs well under both parallelization schemes (CO case). However the shared cache approach has still an interest for well optimized layouts if the cache is small and shared by a lot of cores. The L1 cache of the Fermi architecture can hold 48KB of data and is shared amongst 32 processing units. Using the same edge length prediction, the shared cache scheme should reduce the number of cache

© The Eurographics Association 2010.

misses by 66% compared to the split cache scheme for the CO layout.

6. Related Work

There is another method for efficient isosurface extraction not studied in this article: the seed set and propagation algorithm [BPS96] that proposes to find cells intersected by the isosurface using a graph search on the topology of the mesh. We believe such an approach can benefit for our shared cache scheme. There is also an entirely different approach, not based on marching tetrahedra but on ray tracing [WFM*05]. We plan to further study the impact on the shared cache on ray tracing applications.

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [ZNZ04]. Those techniques could be coupled with our shared cache scheme to efficiently balance the load inside a chunk. However those techniques only take into account the number of instructions and not the cache misses. To overcome this problem, we plan to use a work stealing load balancing scheme.

The closest work is the out-of-core parallel interval tree of [WJV07]. Authors provide a provably efficient load balancing scheme. However the technique is cache aware. Moreover, they focus on distributed processors that do not share caches.

Other layouts exist for efficient mesh traversal. Space filling curves have been used for regular meshes in [PF01]. For unstructured meshes, OpenCCL [YLPM05] provides an heuristic for building efficient layouts but without any performance guarantee. Using the algorithm of OpenCCL, [YM06] proposes an efficient layout for both a mesh and a bounding volume hierarchy tree. This is similar to our adapted min-max tree as both trees are tailored to efficiently access the mesh.

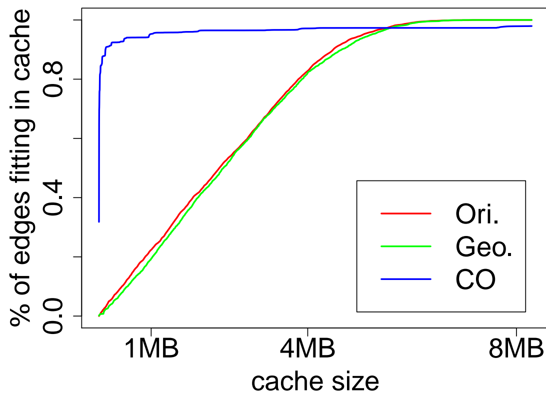


Figure 3: *Cumulative distribution function of edge lengths for various layouts applied to the 150M plasma mesh (the other meshes produce similar graphs).*

To our knowledge, the first work on shared cache is [BG04, CGK*07] that presents a scheduler that follows as much as possible the sequential execution order. However, threads working on too close data can impair the performance of private caches. By using a suitable chunk size, our shared cache algorithm benefits from the shared level of cache while still using private caches efficiently.

7. Conclusion

This paper focused on cache efficiency for isosurface extraction. We theoretically guarantee the performance of the sequential MT algorithm relying on the CO layout introduced in [TDR10], as well as a tree accelerated version taking advantage of the BSP tree built to compute the layout. Then, we consider a parallelization scheme that takes into account that caches may be shared on multicore processors. We prove that this parallelization leads to the same number of cache misses than the sequential algorithm, less than a traditional parallelization assuming caches are all privates. Experiments confirm the benefits of shared cache aware approaches and of CO based algorithms. We expect these techniques to be even more effective on architectures with small caches like the Fermi GPU. Future work will try to combine shared cache aware access patterns with advanced load balancing schemes.

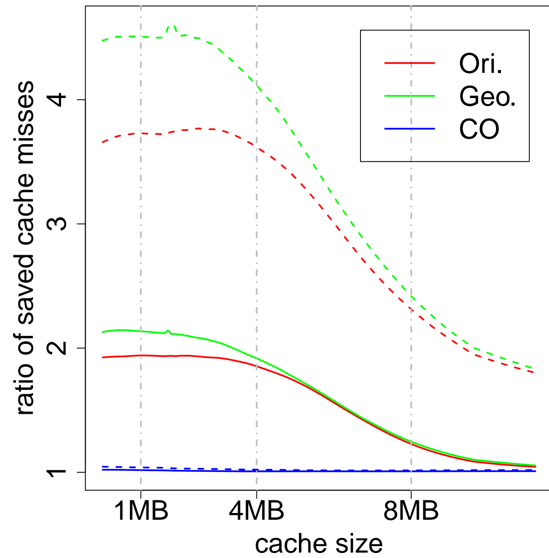


Figure 4: *Gain in cache misses of shared cache over split cache for the* 150*M plasma mesh. Solid lines assume a cache shared between 2 cores, dashed lines assume a cache shared between 4 cores.*

© The Eurographics Association 2010.

References

[ABF04] ARGE L., BRODAL G., FAGERBERG R.: Cache oblivious data structures. *Handbook on Data Structures and Applications* (2004). 5

[BDG*00] BROWNE S., DONGARRA J., GARNER N., HO G., MUCCI P.: A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications 14* (2000), 189–204. 5

[BG04] BLELLOCH G. E., GIBBONS P. B.: Effectively sharing a cache among threads. In *Proceedings of SPAA '04* (2004), pp. 235–244. 9

[BPS96] BAJAJ C. L., PASCUCCI V., SCHIKORE D. R.: Fast isocontouring for improved interactivity. In *Proceedings of VVS '96* (1996), p. 39. 9

[CGK*07] CHEN S., GIBBONS P. B., KOZUCH M., ILEIOS LIASKOVITIS V., AILAMAKI A., BLELLOCH G. E., FALSAFI B., FIX L., HARDAVELLAS N., MOWRY T. C., WILKERSON C.: Scheduling threads for constructive cache sharing on cmps. In *Proceedings of SPAA '07* (2007), pp. 105–115. 9

[CMPS96] CIGNONI P., MONTANI C., PUPPO E., SCOPIGNO R.: Optimal isosurface extraction from irregular volume data. In *Proceedings of VVS '96* (1996), pp. 31–38. 3

[CS97] CHIANG Y.-J., SILVA C.: I/O optimal isosurface extraction. In *Proceedings of Visualization '97* (1997), pp. 293–300. 3, 6

[CS99] CHIANG Y.-J., SILVA C. T.: External memory techniques for isosurface extraction in scientific visualization. In *External memory algorithms* (Boston, MA, USA, 1999), American Mathematical Society, pp. 247–277. 3, 4

[CSS98] CHIANG Y.-J., SILVA C., SCHROEDER W.: Interactive out-of-core isosurface extraction. In *Proceedings of Visualization '98* (1998), pp. 167–174. 3

[FLPR99] FRIGO M., LEISERSON C. E., PROKOP H., RAMACHANDRAN S.: Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), p. 285. 3

[Has10] HASSIDIM A.: Cache replacement policies for multicore processors. In *Proceedings of Innovations in Computer Science* (2010). 1, 4

[PF01] PASCUCCI V., FRANK R.: Global Static Indexing for Real-Time Exploration of Very Large Regular Grids. In *Proceedings of Supercomputing '01* (2001), pp. 45–45. 9

[SHwSS00] SUTTON P. M., HANSEN C. D., WEI SHEN H., SCHIKORE D.: A case study of isosurface extraction algorithm performance. In *Data Visualization 2000* (2000), Springer, pp. 259–268. 3, 6

[SML04] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed.* Kitware Inc., 2004. 2, 3, 6

[TDR10] TCHIBOUKDJIAN M., DANJEAN V., RAFFIN B.: Binary mesh partitioning for cache-efficient visualization. *IEEE Transactions on Visualization and Computer Graphics 99*, PrePrints (2010). http://moais.imag.fr/membres/marc.tchiboukdjian/pub/tvcg10.pdf. 2, 3, 4, 5, 9

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11* (2005), 562–572. 9

[WJV07] WANG Q., JAJA J., VARSHNEY A.: An efficient and scalable parallel algorithm for out-of-core isosurface extraction and rendering. *J. Parallel Distrib. Comput. 67*, 5 (2007), 592–603. 6, 9

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3 (1992), 201–227. 2

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. In *ACM SIGGRAPH* (2005), pp. 886–893. 9

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516. 9

[ZNZ04] ZHANG H., NEWMAN T. S., ZHANG X.: Case study of multithreaded in-core isosurface extraction algorithms. In *EGPGV* (2004), pp. 83–92. 9

© The Eurographics Association 2010.

Vol de travail efficace en cache pour les boucles parallèles 7

Sommaire

7.1 Résumé des contributions **114**
7.2 Discussion et perspectives **116**
7.3 A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores **119**
1 Introduction 119
2 Scheduling for Efficient Shared Cache Usage 120
3 Work-Stealing Window Algorithms with Kaapi 123
4 Marching Tetrahedra for Isosurface Extraction 125
5 Experiments 125
6 Related works 127
7 Conclusions 127
References 128

Dans le chapitre précédent, nous avons introduit deux parallélisations différentes, *Split-Cache* et *Shared-Cache*, pour les algorithmes d'extraction d'isosurfaces opérant sur le maillage FastCOL. Nous étendons cette approche à travers deux aspects.

– Ces algorithmes parallèles utilisent une découpe statique du travail sans équilibrage de charge. Or le temps de traitement d'une cellule n'est pas constant mais dépend notamment du nombre de triangles générés. Nous améliorons ces algorithmes en ajoutant un équilibrage de charge dynamique par vol de travail.
– De plus, nous généralisons les analyses de défauts de cache au cas d'une exécution séquentielle quelconque. Nous avons montré dans le chapitre précédent que la parallélisation *Split-Cache* pour un filtre opérant sur le maillage FastCOL a la même garantie sur les défauts de cache que l'exécution séquentielle. Nous cherchons ici à garantir le nombre de défauts de cache d'un schéma de parallélisation par rapport à l'exécution séquentielle pour n'importe quel algorithme.

Nous étudions dans ce chapitre la parallélisation d'une boucle dont chaque itération peut être traitée indépendamment. Ce schéma parallèle est très courant, particulièrement dans les filtres de visualisation scientifique qui appliquent souvent une même opération à chaque élément (point ou cellule) d'un maillage. Nous examinons cette parallélisation dans le contexte où l'exécution séquentielle de la boucle a une bonne localité (c'est le cas lorsqu'un filtre utilise le maillage FastCOL). Le but est de trouver une parallélisation par vol de travail qui n'augmente pas le nombre de défauts de cache comparé à l'exécution

séquentielle.

Nous nous basons sur le schéma de boucle parallèle présenté dans la section 3.5.3. Ce schéma est implémenté dans le moteur d'exécution de programmes adaptatifs X-Kaapi qui limite les surcoûts [BLTG09]. Nous utilisons le vol coopératif qui permet de descendre à un grain très fin.

7.1 Résumé des contributions

Nous résumons ici les contributions de l'article *A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores* [TDG+10].

Nous étudions les ordonnancements qui conservent la localité de l'exécution séquentielle en se restreignant au cas des tâches indépendantes. Nous proposons un ordonnancement qui est un compromis entre le vol de travail et PDF (*cf.* chapitre 4). Cet ordonnancement est à la fois efficace pour les caches privés et le cache partagé. De plus cet ordonnancement est décentralisé et basé sur du vol de travail donc efficace en pratique.

Ordonnancement à fenêtre optimisé pour les caches

Nous proposons un ordonnancement qui force les threads à rester à distance au plus m de l'exécution séquentielle. L'itération d'indice k de la boucle ne peut être traitée que lorsque toutes les itérations d'indices 1 à $k - m$ ont été traitées. On peut se représenter l'exécution parallèle comme une fenêtre qui se déplace sur l'ordre de traitement séquentiel. Le paramètre m, la taille de la fenêtre, exprime le compromis entre le vol de travail et PDF :

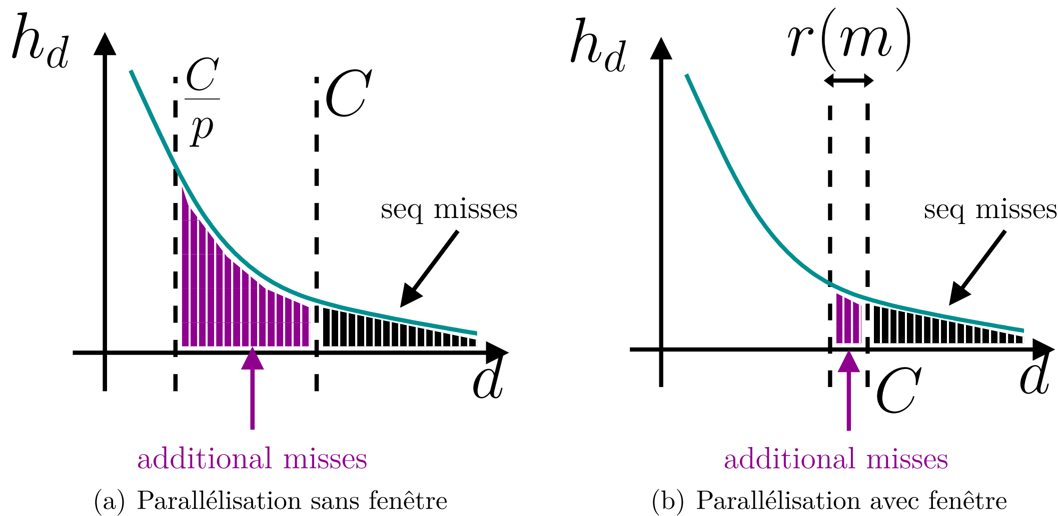
- choisir m grand donne de bonnes performances pour les caches privés et expose beaucoup de parallélisme,
- choisir m petit donne de bonnes performances pour le cache partagé mais expose peu de parallélisme.

Pour généraliser l'analyse des défauts de cache au cas d'un programme séquentiel quelconque, nous utilisons la notion de distances de réutilisation (*cf.* section 1.2.4). L'ensemble de ces distances capture la localité temporelle des accès mémoires du programme. On peut considérer que les distances de réutilisation sont une généralisation des longueurs d'arêtes utilisées dans les chapitres 5 et 6 pour mesurer la localité d'une organisation de maillage. On note $Q_X(C)$ le nombre de défauts de cache de l'exécution X sur un cache de taille C.

Les distances de réutilisation nous permettent d'analyser le nombre de défauts de cache partagé de deux ordonnancements différents : l'ordonnancement standard sans fenêtre et l'ordonnancement optimisé pour les caches avec fenêtre.

- Nous montrons que l'ordonnancement sans fenêtre, qui s'éloigne très fortement de l'exécution séquentielle, génère autant de défauts de cache sur p cœurs que l'exécution séquentielle sur un cache de taille C/p (*cf.* figure 7.1(a)).

$$Q_{no-win}(C) = Q_{seq}\left(\frac{C}{p}\right)$$



(a) Parallélisation sans fenêtre

(b) Parallélisation avec fenêtre

Figure 7.1 – Histogramme des distances de réutilisation dans le cas des ordonnancements avec et sans fenêtre. L'exécution parallèle sans fenêtre cause beaucoup plus de défauts de cache que l'exécution séquentielle alors que l'exécution parallèle avec fenêtre limite le nombre de défauts de cache supplémentaires.

– Par contre, l'ordonnancement avec fenêtre n'induit qu'une augmentation légère des défauts de cache par rapport à l'exécution séquentielle sur la même taille de cache. Ce surcoût $f(m)$ dépend de la proximité avec l'exécution séquentielle m (*cf.* figure 7.1(b)).

$$Q_{win}(C) = Q_{seq}(C) + f(m)$$

Cette analyse montre que la garantie donnée dans le chapitre précédent dans le cas du maillage FastCOL, c'est-à-dire la parallélisation par morceaux *Split-Cache* du maillage FastCOL cause autant de défauts de cache sur un cache partagé que l'exécution séquentielle, se généralise pour des calculs quelconques.

Implémentation efficace de l'ordonnancement à fenêtre

Nous proposons une implémentation par vol de travail de l'ordonnancement sans fenêtre appelée NoWindow et une avec fenêtre appelée StaticWindow. Ces deux implémentations sont les équivalents de *Split-Cache* et *Shared-Cache* mais avec un équilibrage de charge dynamique par vol de travail. Nous proposons également une implémentation optimisée de l'ordonnancement avec fenêtre appelée SlidingWindow. Dans cette version, la fenêtre autour de l'ordre d'exécution séquentiel avance dès que les itérations au début de la fenêtre ont été traitées. SlidingWindow exploite tout le parallélisme disponible contrairement à StaticWindow.

Nous expérimentons ces algorithmes parallèles sur une extraction d'isosurface. Nous comparons à la fois les ordonnancements avec et sans fenêtre, les parallélisations statiques et dynamiques, et les bibliothèques de programmation parallèle X-Kaapi, TBB et la STL parallèle GNU.

Sur une architecture à cache partagé, les versions avec fenêtre génèrent moins de défauts de cache et sont donc plus performantes. Les parallélisations dynamiques

Ordonnancement	Défauts de cache L_2	Défauts de cache L_3	Temps (s)
NoWindow	$4,496 \cdot 10^9$	$37,4 \cdot 10^6$	24.8
StaticWindow	$4,496 \cdot 10^9$	$21,2 \cdot 10^6$	25.5

Table 7.1 – Comparaison des ordonnancements avec et sans fenêtre sur un filtre de rendu volumique par lancer de rayons sur le processeur Xeon 5530. Chaque cœur a un cache L_2 privé de 256KO et les 4 cœurs partageant un cache L_3 de 8MO. L'ordonnancement avec fenêtre réduit le nombre de défauts de cache partagé.

obtiennent de meilleurs performances que les parallélisations statiques, X-Kaapi étant l'implémentation la plus rapide.

L'ordonnancement par fenêtre glissante qui exploite mieux le parallélisme permet de diminuer le nombre de vols sans augmenter le nombre de défauts de cache et obtient les meilleures performances.

7.2 Discussion et perspectives

Extensions directes

Nous avons également testé l'ordonnancement à fenêtre glissante sur deux autres algorithmes : un rendu volumique par lancer de rayons et une multiplication de matrice.

- L'algorithme de lancer de rayons parcourt les rayons en utilisant une *space filling curve* sur les pixels de l'image. Cet ordre de traitement permet d'améliorer la localité car deux rayons correspondant à des pixels proches dans l'image traversent le maillage dans des zones similaires ce qui permet de réutiliser les points et les cellules déjà chargés dans le cache. Utiliser un ordonnancement à fenêtre limite le nombre de défauts de cache partagé d'une exécution parallèle sur multicœur (*cf.* table 7.1). Ce gain en défaut de cache ne se traduit pas par un gain en temps pour l'instant. L'implémentation testée n'utilise pas la fenêtre glissante et le filtre n'est pas suffisamment optimisé. Ce travail a été réalisé lors du stage de Mathieu Westphal.
- L'algorithme de multiplication de matrices est basé sur une multiplication de matrices par blocs où chaque couple de blocs des matrices d'entrée est multiplié en utilisant un appel aux BLAS puis accumulé dans un bloc de la matrice finale. L'ordre de traitement des blocs est basé sur une *space filling curve*. Dans ce cas, le traitement d'une itération de la boucle, c'est-à-dire la multiplication de blocs, n'est pas indépendant. En effet, plusieurs itérations s'accumulent dans le même bloc de la matrice résultat. Nous proposons un ordre de traitement inspiré d'une *space filling curve* pour garantir la localité mais qui permet aussi d'éloigner les itérations qui s'accumulent dans le même bloc de la matrice résultat. L'ordonnancement à fenêtre permet à la fois une bonne localité sur la lecture des blocs des matrices d'entrée et empêche que les itérations qui s'accumulent dans le même bloc soient traitées en parallèle (ce qui garantie la correction de l'algorithme). C'est un travail en cours. Nous espérons que cet algorithme donnera de meilleures performances que l'algorithme de multiplication de matrices basé sur Controlled-PDF grâce à une meilleure exploitation du parallélisme.

Il serait aussi intéressant de tester les ordonnancements à fenêtre sur les nouveaux processeurs multicœurs comportant 8 cœurs partageant le même cache pour évaluer le passage à l'échelle. On pourrait aussi envisager le couplage de plusieurs ordonnancements à fenêtre, un par processeur multicœur, en utilisant du vol de travail classique pour répartir la charge entre les processeurs.

Nous pensons que cette méthode de traitement des boucles parallèles est efficace et générique. De plus, cette méthode est facile à intégrer dans l'interface d'une boucle parallèle existante en ajoutant le paramètre de taille de la fenêtre m.

Ordonnancement efficace en cache

La limitation principale de l'ordonnancement à fenêtre est le choix de ce paramètre m. En effet, il faut connaître la taille du cache partagé et des caches privés pour obtenir le paramètre optimal. Cet ordonnancement n'est donc pas *cache-oblivious*, même si l'algorithme séquentiel est lui *cache-oblivious*.

Cependant il parait très difficile de se passer de cette connaissance. Il serait intéressant d'obtenir une preuve d'impossibilité théorique d'ordonnancer de manière *cache-oblivious* un algorithme déjà *cache-oblivious* dès que l'architecture contient à la fois des caches privés et des caches partagés. En effet, le choix de m doit être tel que :

- la quantité de données que représentent m/p itérations consécutives de la boucle soit plus grande que la taille d'un cache privé afin d'éviter les défauts de cache de cohérence,
- la quantité de données que représentent m itérations consécutives soit plus petite que la taille du cache partagé par ces p cœurs afin de limiter le nombre de défauts de cache pour le cache partagé.

Enfin, il serait intéressant de généraliser cette analyse au cas avec dépendances en combinant le modèle du DAG avec les distances de réutilisation.

A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier*,
Fabien Lementec, and Bruno Raffin

MOAIS Project, INRIA- LIG
ZIRST 51, avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
(marc.tchiboukdjian, vincent.danjean, fabien.lementec,
bruno.raffin)@imag.fr thierry.gautier@inrialpes.fr

Abstract. Reordering instructions and data layout can bring significant performance improvement for memory bounded applications. Parallelizing such applications requires a careful design of the algorithm in order to keep the locality of the sequential execution. In this paper, we aim at finding a good parallelization of memory bounded applications on multicore that preserves the advantage of a shared cache. We focus on sequential applications with iteration through a sequence of memory references. Our solution relies on an adaptive parallel algorithm with a dynamic sliding window that constrains cores sharing the same cache to process data close in memory. This parallel algorithm induces the same number of cache misses as the sequential algorithm at the expense of an increased number of synchronizations. Experiments with a memory bounded application confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs. On quad cores Nehalem processor, our algorithms are 10% to 30% faster than algorithms not optimized for shared cache thanks to a reduced number of last level cache misses.

1 Introduction

Many applications in scientific computing are memory bounded. Favoring the locality of access patterns through data and computation reordering can bring significant performance benefits. When designing parallel algorithms, one must be extra careful not to lose the locality of the sequential application, which is the key for good performance.

In most last generation multicores, the last level of cache is shared among all cores of the chip. For instance the Intel Nehalem, the AMD Phenom and Opteron (only for the quadcores and hexacores) and the IBM Power7 all have a shared L_3 cache. Recent GPU architectures also adopt this cache design: the L_1 cache of a NVIDIA Fermi streaming multiprocessor is shared among 32 cores.

In this paper, we focus on one specific aspect of the parallelization of memory bounded applications: how to adapt the scheduling to take advantage of the shared

* Part of this work was done while the second author was visiting the ArTeCS group of the University Complutense, Madrid, Spain.

caches of multicore processors. The goal is to propose a scheduling algorithm that improves performance by reducing cache misses, compared to parallel algorithms that do not take into account the shared cache amongst several cores. We propose to have cores working on independent but close (regarding the memory layout) data sets that can all fit in the shared cache. If a core needs a data that is not in its data set, there is a good chance it will find it in the data set loaded in the cache by one of its neighbors, thus saving cache misses. The algorithm behaves as if each core would benefit from a full-size private cache, at the price of a few extra synchronizations required to ensure a proper collaboration between cores.

This paper focuses on algorithms that take an input sequence to produce an output sequence of results. Such algorithms encompass many of the C++ Standard Template Library (STL) functions like `for_each` or `transform`. Moreover, many parallel libraries such as Intel TBB or the GNU STL parallel mode provide parallel implementations of the STL. Thus providing shared cache aware parallelizations of these algorithms can improve performance of many applications running on multicores.

We provide a cache constraint that parallel algorithms should respect to induce no more cache misses than the sequential algorithms. We present two new algorithms respecting this cache constraint and two implementations, one based on PThread and the other one based on work-stealing allowing efficient dynamic load balancing. We also implement those new algorithms with the parallel library TBB and the GNU parallel STL and compare them with our implementations on the `for_each` function.

The paper is organized as follows. In section 2, we present the cache constraint and the associated algorithms. In section 3, we detail the implementation of these two algorithms using the work-stealing based framework Kaapi. Finally, we introduce the application we use to benchmark our algorithms in section 4 and the experimental data in section 5 before the conclusions.

2 Scheduling for Efficient Shared Cache Usage

2.1 Review of Work-Stealing and Parallel Depth First Schedules

Work Stealing (WS) is a scheduling algorithm that is very efficient both in theory and in practice. It has been implemented in many languages and parallel libraries including Cilk [1] and TBB [2]. In WS, each processor manages its own list of tasks. When a processor becomes idle, it becomes a thief, randomly chooses another processor, the victim, and try to steal some work. For an efficient load balancing, the thief should choose a task that represents a big amount of work far in memory from the work of the victim. This reduces the number of steal operations and thus synchronization costs. Unfortunately, stealing such tasks may not be optimal if one takes into account the shared cache of recent multicores.

Contrary to WS, the Parallel Depth First (PDF) schedule of [3] tries to optimize shared cache usage. This schedule is based on the sequential order of execution, which is supposed to be cache-efficient. When several tasks are available, a processor will preferably execute the earliest task in the sequential order. The authors showed that a PDF schedule induces no more cache misses

than the sequential execution when the parallel execution uses a slightly bigger cache. However, computing and maintaining such a schedule is costly in practice.

Informally, one could think of the PDF scheduler as a WS scheduler where the thieves would choose the closest task in the victim list inducing lots of steal operations. This is not as simple as all processors, not only a victim and its thief, should work on data close in memory. In addition to the steal close operation, another mechanism is needed to prevent processors to deviate from each other after the steal operation. The cache constraint we present in the next section serves exactly this purpose. The processing order we proposed is a trade-off between WS and PDF. Processors work on data just close enough in memory to fit in the shared cache. This way the parallel application should not make more cache misses than the sequential application. The number of synchronizations is better than PDF but not as good as WS. However, as the number of cache misses is reduced, the overall performance should be improved over WS.

2.2 Window Algorithms for Sequence Processing

We consider algorithms that take an input sequence $i_1, i_2, \ldots, i_n$ (different input elements can share some data) and a function *op* to be applied on all elements of the input producing an output sequence $o_1, o_2, \ldots, o_{n'}$. Notice that treating one element may produce a different number of elements in the output sequence. Most STL algorithms are variations over this model. The sequential algorithm processes the sequence in order from i_1 to i_n. We assume that the sequential algorithm already performs well with respect to temporal locality of data accesses. Data processed closely in the sequential execution are also close in memory. We focus on the case where all elements of the sequence can be processed in parallel.

We introduce two parallel algorithms to process such a sequence in parallel. These two algorithms are parameterized by m, the maximum distance between the threads. In the first one, denoted *static-window*, the sequence is first divided into n/m chunks of m contiguous elements. Then, each chunk is processed in parallel by the p processors sharing the same cache. Several strategies can be used to parallelize the processing of each chunk. The m elements could be statically partitioned into p groups of m/p elements, one per processor, or a work-stealing scheme can be used to dynamically balance the load. The second parallel algorithm, denoted *sliding-window*, is a relaxed version of the *static-window* algorithm. At the beginning of the algorithm, the first m elements of the sequence are ready and can be processed in any order. Each time the first element i_k not yet processed in the sequence is treated by a processor, it enables the element i_{k+m} at the end of a window of size m. These two algorithms will be compared with an algorithm denoted *no-window* that do not respect the cache constraint. All the elements of the sequence can be processed in any order. This algorithm induces more cache misses than the sequential algorithm and the window algorithms, but it requires fewer synchronizations.

2.3 Cache Performance of Window Algorithms

The re-use distance captures the temporal locality of a program [4]. Let consider a series of memory references $(x_k)_{k \geq 0}$. When a reference x_k access an element

for the first time, the re-use distance of x_k is infinite. If the element has been previously accessed, $x_{k'} = x_k$ with $k' > k$, the re-use distance of $x_{k'}$ is equal to the number of distinct elements accessed between these two references x_k and $x_{k'}$. Let h_d denote the number of memory references with a re-use distance d. The number of cache misses of a fully associative LRU cache of size C is equal to $M_{\text{seq}} = \sum_{d=C+1}^{\infty} h_d$. We can extend this definition to sequence processing algorithms: if processing i_k and $i_{k'}$ uses similar data, the re-use distance is $k' - k$.

We consider now p processors sharing the same cache that process the sequence in parallel in distant places like the *no-window* algorithm. As we assumed the sequence has good temporal locality, elements far-away in the sequence use distinct data. In this case, the re-use distance is multiplied by p as to each access of one processor corresponds $p - 1$ accesses of the others to distinct elements. Thus, the number of cache misses is $M_{\text{no-win}} = \sum_{d=C+1}^{\infty} h_{d/p} \approx \sum_{d=C/p+1}^{\infty} h_d$. The *no-window* algorithm induces as many cache misses as the sequential algorithm with a cache p times smaller. We now restrain the processors to work on elements at distance less than m like in the window algorithms. Let $r(m)$ be the maximum number of distinct memory references when processing $m - 1$ consecutive elements of the input sequence. In the worst case, when processing element i_k, all elements $i_{k+1}, \ldots, i_{k+m-1}$ have already been processed accessing at most $r(m)$ additional distinct elements compared to the sequential order. Thus the re-use distance is increased by at most $r(m)$. The number of cache misses is $M_{\text{window}} \leq \sum_{d=C+1}^{\infty} h_{d-r(m)} = M_{\text{seq}} + \sum_{d=C+1-r(m)}^{C} h_d$. As we assumed the sequence has good temporal locality, $r(m)$ is small compared to m and h_d is small for large d. Therefore $\sum_{d=C+1-r(m)}^{C} h_d$ is small and the window algorithms induce approximately the same number of cache misses as the sequential algorithm.

2.4 PThread Parallelization of Window Algorithms

We present here the implementation of the *no-window* and *static-window* algorithms using PThreads. The PThread implementation allows a fine grain control on synchronizations with very little overhead.

For the *no-window* algorithm, the sequence is statically divided into p groups. Each group is assigned to one thread bound to one processor and all threads synchronize at the end of the computation. For the *static-window* algorithm, the sequence is first divided into chunks of size m. Then each chunk is statically divided into p groups and all threads synchronize at the end of each chunk before starting to compute the next one. Each synchronization is implemented with a `pthread_barrier`. Threads wait at the barrier and are released when all of them have reached the barrier. Although we expect the threads in the *static-window* algorithm to spend more time waiting for other threads to finish their work, the reduction of cache misses should compensate this extra synchronization cost. The *sliding-window* algorithm has not been implemented in PThread because it would require a very complex code. We present in the next section a work-stealing framework allowing to easily implement all these algorithms.

```c
typedef struct {
  InputIterator       ibeg;
  InputIterator       iend;
  OutputIterator      obeg;
  size_t              osize;
} Work_t ;

void dowork(...) {
  complete_work:
    while (iend != ibeg) {
      kaapi_stealpoint(..., &splitter);
      for(i=0; i<grain; ++i, ++ibeg)
        op(ibeg, obeg, &osize);
      kaapi_preemptpoint(..., &reducer);
    }
    if ( kaapi_preempt_next_thief(...) )
      goto complete_work ;
} // no more work -> become a thief

void reducer(Work_t *victim, Work_t *thief) {
  memmove( victim->obeg, thief->obeg,
           thief->osize );
  victim->osize += thief->osize;
  victim->ibeg   = thief->ibeg;
  victim->iend   = thief->iend;
} // victim -> dowork / thief -> try to steal
```

```c
void splitter( Work_t *victim, int count,
               kaapi_request_t* request ) {
  int i = 0;
  size_t size = victim->iend - victim->ibeg;
  size_t bloc = size / (1+count);
  InputIterator local_end = victim->iend;
  Work_t *thief;

  if (size < gain)
    return;
  while (count >0) {
    if (kaapi_request_ok(&request[i])) {
      thief->iend  = local_end;
      thief->ibeg  = local_end - bloc;
      thief->obeg  = intermediate_buffer;
      thief->osize = 0;
      local_end   -= bloc;
      kaapi_request_reply_ok(thief,
                             &request[i]);
      --count;
    }
    ++i;
  }
  victim->iend  = local_end;
} // victim and thieves -> dowork
```

Fig. 1. C implementation of the adaptive *no-window* algorithm using the Kaapi API.

3 Work-Stealing Window Algorithms with Kaapi

In this section, we present the low level API of Kaapi [5] and detail the implementation of the windows algorithms.

3.1 Kaapi Overview

Kaapi is a programming framework for parallel computing using work-stealing. At the initialization of a Kaapi program, the middleware creates and binds one thread on each processor of the machine. All non-idle threads process work by executing a sequential algorithm (`dowork` in fig. 1). All idle threads, the thieves, send work requests to randomly selected victims. To allow other threads to steal part of its work, a non-idle thread must regularly check if it received work requests using the function `kaapi_stealpoint`. At the reception of `count` work requests, a `splitter` is called and divides the work into `count+1` well-balanced pieces, one for each of the thieves and one for the victim.

When a previously stolen thread runs out of work, it can decide to preempt its thieves with the `kaapi_preempt_next_thief` call. For each thief, the victim merges part of the work processed by the thief using the `reducer` function and takes back the remaining work. The preemption can reduce the overhead of storing elements of the output sequence in an intermediate buffer when the final place of an output element is not known in advance. To allow preemption, each thread regularly checks for preemption requests using the function `kaapi_preemptpoint`.

To amortize the calls to the Kaapi library, each thread should process several units of work between these calls. This number is called the *grain* of the algorithm. In particular, a victim thread do not answer positively to a work request when it has less than *grain* units of work.

Compared to classical WS implementations, tasks (`Work_t`) are only created when a steal occurs which reduces the overhead of the parallel algorithm compared

to the sequential one [6]. Moreover, the steal requests are treated by the victim and not by the thieves themselves. Although the victim has to stop working to process these requests, synchronization costs are reduced. Indeed, instead of using high-level synchronization functions (mutexes, etc.) or even costly atomic assembly instructions (compare and swap, etc.), the thieves and the victim can communicate by using standard memory writes followed by memory barriers, so no memory bus locking is required. Additionally, the `splitter` function knows the number `count` of thieves that are trying to steal work to the same victim. Therefore, it permits a better balance of the workload. This feature is unique to Kaapi when compared to other tools having a work-stealing scheduler.

3.2 Work-Stealing Algorithm for Standard (*no-window*) Processing

It is straightforward to implement the *no-window* algorithm using Kaapi. The work owned by a thread is described in a structure by four variables: `ibeg` and `iend` represents the range of elements to process in the input sequence, `obeg` is an iterator on the output sequence and `osize` is the number of elements written on the output. At the beginning of the computation, a unique thread possesses the whole work: `ibeg=0` and `iend=n`. Each thread processes its assigned elements in a loop. Code of Fig. 1 shows the main points of the actual implementation.

3.3 Work-Stealing Window Algorithms

The *static-window* algorithm is very similar to the *no-window* algorithm of the previous section. The first thread owning the total work has a specific status, it is the *master* of the window. Only the master thread has knowledge of the remaining work outside the m-size window. When all elements of a window have been processed, the master enables the processing of the new window by updating its input iterators `ibeg = iend` and `iend += m`. This way, when idle threads request work to the master thread, the stolen work is close in the input sequence. Moreover, all threads always work on elements at distance at most m.

The *sliding-window* algorithm is a little bit more complex. In addition to the previous iterators, the master also maintains `ilast` an iterator on the first element after the stolen work in the input sequence (see Fig. 2). When the master does not receive any work request, then `iend == ilast == ibeg+m`. When the master receives work requests, it can choose to give work on both sides of the stolen work. Distributing work in the interval `[ibeg,iend]` corresponds to the previous algorithm. The master thread can also choose to distribute work close to the end of the window, in the interval `[ilast,ibeg+m]`. We implemented several variants of the `splitter`. The `local_splitter` gives in priority work in the interval `[ibeg,iend]`. It favors processing elements at the beginning to fast-forward the window thus enabling new elements to be processed. The `distant_splitter` gives in priority work in the interval `[ilast,ibeg+m]`. By distributing work at the end of the window, it should reduce the number of preemptions. The last one, `balanced_splitter` try to give well-balanced amount of work to all thieves by dividing the union of both intervals into equal size pieces. No piece of work can contains elements on both sides of the window as the resulting work would not be an interval.

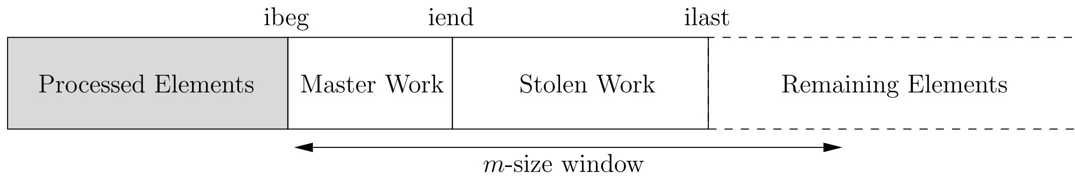


Fig. 2. Decomposition of the input sequence in the *sliding-window* algorithm.

4 Marching Tetrahedra for Isosurface Extraction

Isosurface extraction is one on the most classical filters of scientific visualization. It provides a way to understand the structure of a scalar field in a three dimensional mesh by visualizing surfaces of same scalar value. The marching tetrahedrons (MT) is an efficient algorithm for isosurface extraction [7]. For one cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a linear approximation of the isosurface going through this cell. Applied on all mesh cells sequentially, it leads to a cost linear in the number of cells.

We now look at cache misses induced by MT. The mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). Points are accessed by following a reference from the cell array, e.g. reading coordinates of a point. As cells close in the cell array often use common points or points with close indices, processing cells in the same order as the sequential algorithm induces fewer cache misses when accessing the point array due to an improved temporal locality.

When implementing the window algorithms, the window size m should be chosen such that a sub-part of m cells of the mesh fits in the shared cache. Each point is coded on four doubles and each tetrahedron with four references (64bit integers) to points. On average, meshes have six times more tetrahedrons than points. So, for an 8MB cache, we approximately have $m = 225,000$. The same reasoning could apply to other mesh processing applications.

5 Experiments

We present experiments using the MT algorithm for isosurface extraction. We first calibrate the grain for the work-stealing implementation and the window size m for the window algorithms. Then, we compare the Kaapi framework with other parallel libraries on a central part of the MT algorithm which can be written as a `for_each`. Finally we compare the *no-window*, *static-window* and *sliding-window* algorithms implementing the whole MT.

All the measures reported are averaged over 20 runs and are very stable. The numbers of cache misses are obtained with PAPI [8]. Only last level cache misses are reported as the lower level cache misses are the same for all algorithms. Two different multicores are used, a quadcore Intel Xeon Nehalem E5540 at 2.4Ghz with a shared 8MB L_3 cache and a dualcore AMD Opteron 875 at 2.2Ghz with two 1MB L_2 private caches. If the window algorithms reduce the number of cache misses on the Nehalem but not on the Opteron, one can conclude that this is due to the shared cache.

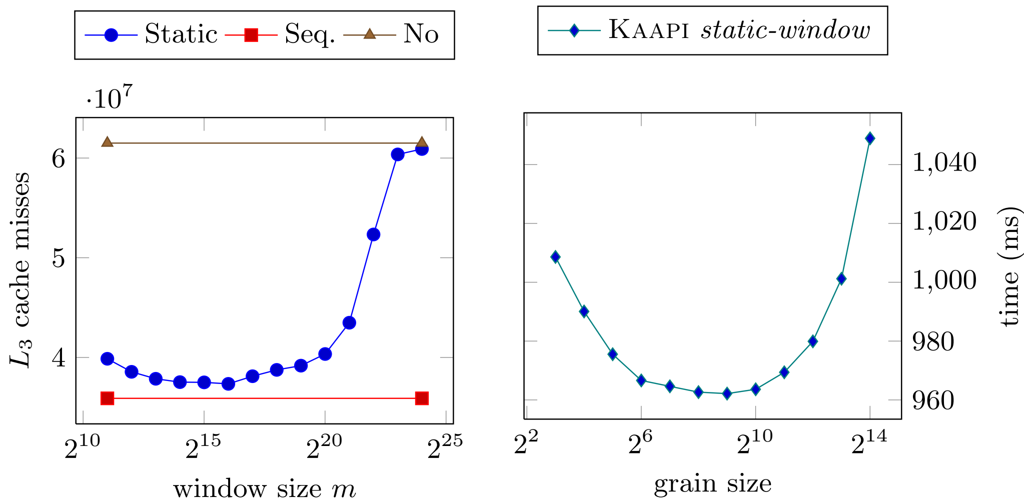


Fig. 3. (Left) Number of L_3 cache misses for the PThread implementation of the *static-window* algorithm for various window sizes compared to the sequential algorithm and the *no-window* algorithm. (Right) Parallel time for the KAAPI implementation of the *static-window* algorithm with various grain sizes. (Both) All parallel algorithms use the 4 cores of the Nehalem processor.

5.1 Calibrating the Window Algorithms

Fig. 3(left) shows the number of L_3 cache misses for the *static-window* algorithm compared to the sequential algorithm and the *no-window* algorithm. The *static-window* algorithm is very close to the sequential algorithm for window sizes less than 2^{20}. It does not exactly match the sequential performance due to additional `reduce` operations for managing the output sequence in parallel. With bigger windows, L_3 misses increase and tend to the *no-window* algorithm. For the remaining experiments, we set $m = 2^{19}$.

Fig. 3(right) shows the parallel time of the *static-window* algorithm with the KAAPI implementation for various grain sizes. Performance does not vary much, less than 10% on the tested grains. For small grains, the overhead of the KAAPI library becomes significant. For bigger grains, the load balancing is less efficient. For the remaining experiments, we choose a grain size of 128. We can notice that the KAAPI library allows very fine grain parallelism: processing 128 elements takes approximately 3μs on the Nehalem processor.

5.2 Comparison of Parallel Libraries on `for_each`

Table 1 compares KAAPI with the GNU parallel library (from gcc 4.3) (denoted GNU) and Intel TBB (v2.1) on a `for_each` used to implement a central sub-part of the MT algorithm. The GNU parallel library uses the best scheduler (parallel balanced). TBB uses the auto partitioner with a grain size of 128. TBB is faster than GNU on Nehalem and it is the other way around on Opteron. KAAPI shows the best performance on both processors. This can be explained by the cost of the synchronization primitives used: POSIX locks for GNU, compare and swap for TBB and atomic writes followed by memory barriers for KAAPI.

Time (ms)		Nehalem				Opteron			
Algorithms	#Cores	STL	GNU	TBB	KAAPI	STL	GNU	TBB	KAAPI
no-window	1	3,987	4,095	3,975	4,013	9,352	9,154	10,514	9,400
	4		1,158	1,106	1,069		2,514	2,680	2,431
static-window	1	3,990	4,098	3,981	4,016	9,353	9,208	10,271	9,411
	4		1,033	966	937		2,613	2,776	2,598

Table 1. Performance of the *no-window* and *static-window* algorithms on a `for_each` with various parallel libraries. GNU is the GNU parallel library. Time are in ms.

5.3 Performance of the Window Algorithms

We now compare the performance of the window algorithms. Table 1 shows that the *static-window* algorithm improves over the *no-window* algorithm for all libraries on the Nehalem processor. However, on the Opteron with only private caches, performances are in favor of the *no-window* algorithm. This was expected as the Opteron has only private caches and the *no-window* algorithm has less synchronizations. We can conclude that the difference observed on Nehalem is indeed due to the shared cache.

Fig. 4(left) presents speedup of all algorithms and ratio of cache misses compared to the sequential algorithm. The *no-window* versions induces 50% more cache misses whereas the window versions only 13% more. The window versions are all faster compared to the *no-window* versions. Work stealing implementations with KAAPI improves over the static partitioning of the PThread implementations. The *sliding-window* (with the best splitter: `balanced_splitter`) shows the best performance.

Fig. 4(right) focus on the comparison of the *sliding-window* and *static-window* algorithms. Due to additional parallelism, the number of steal operations are greatly reduced in the *sliding-window* algorithm (up to 2.5 time less for bigger windows) leading to an additional gain around 5%.

6 Related works

Previous experimental approaches have shown the interest of efficient cache sharing usage, on a recent benchmark in [9] and on data mining applications in [10]. In this paper, we go beyond those specific approaches by providing general algorithms for independent tasks parallelism which respect the sequential locality.

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [11]. However, none of these techniques take into account the number of cache misses and the shared cache of multicore processors. Optimization of sequential locality for mesh applications has been studied through mesh layout optimization in [12].

7 Conclusions

This paper focuses on exploiting the shared cache of last generation multicores. We presented new algorithms to parallelize STL-like sequence processing. Experiments on several parallel libraries confirm that these techniques increase performance from 10% to 30% thanks to a reduced number of last level cache misses.

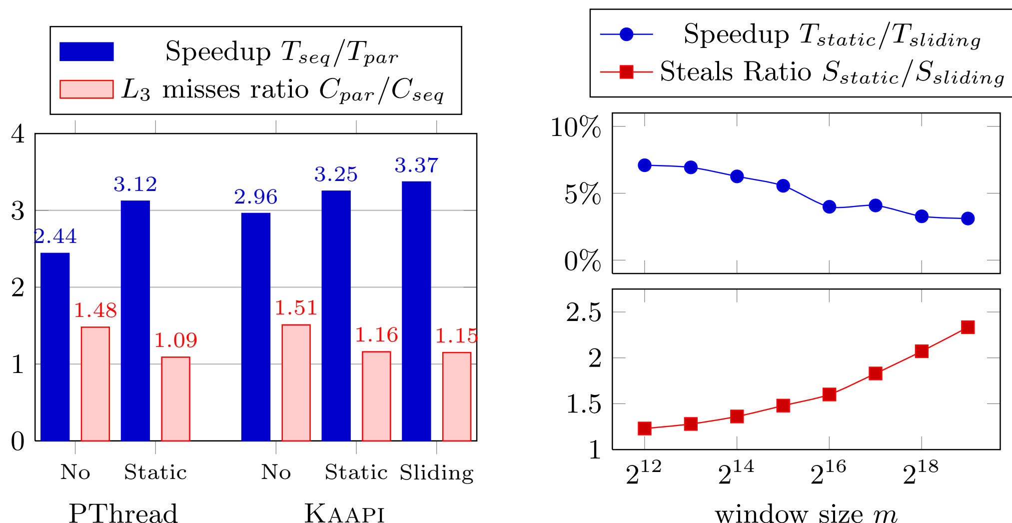


Fig. 4. (Left) Speedup and ratio of increased cache misses over the sequential algorithm for the *no-window*, *static-window* and *sliding-window* algorithms with PThread and KAAPI implementations. (Right) Speedup and ratio of saved steal operations for the *sliding-window* algorithm over the *static-window* algorithm with the KAAPI implementation. (Both) All algorithms run on the 4 cores of the Nehalem.

References

1. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37**(1) (1996) 55–69
2. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: IPDPS. (2008)
3. Blelloch, G.E., Gibbons, P.B.: Effectively sharing a cache among threads. In: SPAA. (2004)
4. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: Proc. of ICS. (2003)
5. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO. (2007)
6. Traoré, D., Roch, J.L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel stl algorithms. In: Euro-Par. (2008)
7. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed. Kitware Inc. (2004)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications **14** (2000)
9. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: PPoPP. (2010)
10. Jaleel, A., Mattina, M., Jacob, B.: Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In: HPCA. (2006)
11. Zhang, H., Newman, T.S., Zhang, X.: Case study of multithreaded in-core isosurface extraction algorithms. In: EGPGV. (2004)
12. Tchiboukdjian, M., Danjean, V., Raffin, B.: Binary mesh partitioning for cache-efficient visualization. Visualization and Computer Graphics, IEEE Transactions on **16**(5) (sept.-oct. 2010) 815 –828

Nouvelle analyse des ordonnancements par vol de travail 8

Sommaire

8.1 Résumé des contributions **130**
8.2 Discussion et perspectives **131**
8.3 Decentralized List Scheduling **133**
1 Introduction 134
2 Model and Notations 136
3 Principle of the analysis and main theorem 138
4 Unit independent tasks 141
5 Going further on the unit tasks model 144
6 Weighted independent tasks 147
7 Tasks with precedences 148
8 Experimental study 151
9 Concluding remarks 153
References 154

Nous avons vu dans le chapitre 3 que le temps d'exécution T_m d'un programme parallèle ordonnancé sur m cœurs par vol de travail vérifie l'équation suivante :

$$T_m = \left(1 + \frac{c_f}{g}\right) \cdot \frac{W}{m} + c_s \cdot \frac{S}{m}$$

avec S le nombre de vols. Une interprétation classique de cette équation (par exemple Frigo *et al.* [FLH98]) consiste à dire : le premier terme domine le second car $W \gg D$ et $S = O(m \cdot D)$.

L'ordonnancement efficace pour cache partagé du chapitre précédent nécessite de paralléliser de petits volumes de données. En effet, il faut que la quantité de données traitée en parallèle rentre dans le cache partagé. On ne peut donc pas négliger le second terme $c_s \cdot S/m$. Ce terme dû aux vols n'a pas été étudié dans les travaux actuels et devient de plus en plus important avec la multiplication du nombre de cœurs.

Pour améliorer l'efficacité des ordonnancements par vol de travail, on peut tenter de réduire les deux composantes de ce coût dû aux vols.

- Le coût d'un vol c_s peut être réduit en améliorant la structure de données concurrente qui contient le travail (par exemple la pile de tâches prête et son protocole THE). Par exemple, dans le cadre du moteur de vol de travail adaptatif X-Kaapi, Thierry Gautier et Fabien Le Mentec ont proposé un mécanisme qui

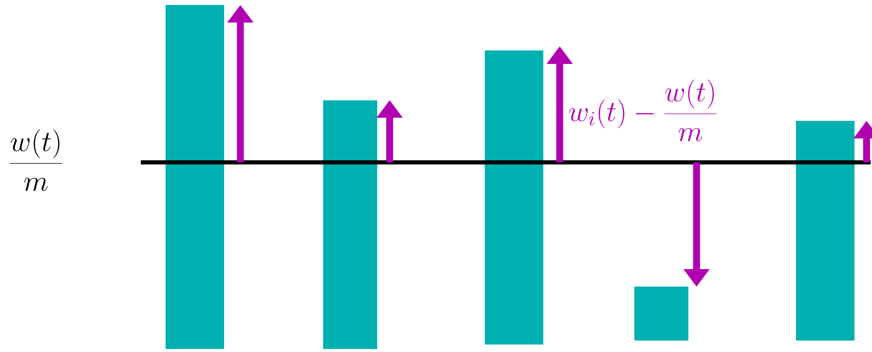


FIGURE 8.1 – Fonction potentielle mesurant le déséquilibre de travail entre les cœurs

permet de réduire le coût des vols en combinant plusieurs requêtes de vol lorsque la même victime est choisie par plusieurs voleurs. De plus, cette technique permet de mieux équilibrer le travail au moment du vol. En effet, lorsque k voleurs visent la même victime, le travail w est découpé en $k+1$ parts de $w/(k+1)$, une par voleur et la dernière pour la victime. En utilisant la pile de tâches standard, le travail se trouve divisé en puissances de 2 et non pas en parts égales.

– Le nombre de vols S peut être réduit en modifiant la politique de vol.

Dans ce chapitre, nous proposons une nouvelle analyse du vol de travail qui vise à mieux comprendre le nombre de vols S. En effet, l'analyse précédente de Arora *et al.* [ABP98] ne donne qu'une borne très large sur S avec un facteur constant important. Notre nouvelle analyse est plus fine et permet, par exemple, de quantifier l'impact de la combinaison des requêtes de vol sur le nombre de vols S.

8.1 Résumé des contributions

Nous résumons ici les contributions de l'article *Decentralized List Scheduling* actuellement en soumission dans un journal. La version courte de cet article a été acceptée pour présentation à la conférence ISAAC [TGT+10].

Notre analyse se base sur l'étude d'une fonction potentielle Φ qui représente le déséquilibre de travail entre les processeurs (*cf.* figure 8.1). Lors d'un vol, une partie du travail d'un processeur actif est transféré vers un processeur inactif ce qui fait diminuer le déséquilibre de travail et donc la fonction potentielle (*cf.* figure 8.2).

En supposant connu le nombre de voleurs r_t à l'instant t, on analyse la diminution moyenne de la fonction potentielle en une étape. On peut exprimer la diminution du potentiel due aux vols à l'instant t en fonction du potentiel à l'instant t et du nombre de voleurs

$$\mathbb{E}[\Phi_t - \Phi_{t+1}] = f(r_t, \Phi_t). \tag{8.1}$$

L'équation obtenue utilise comme paramètre le nombre de voleurs à l'instant t. Comme il est difficile de modéliser l'évolution stochastique du nombre de voleurs, on considère qu'un adversaire choisit le pire nombre de voleurs à chaque étape. Ce nombre de voleurs ne reflète plus une exécution valide mais nous permet de résoudre l'équation (8.1) et d'obtenir une borne supérieure sur le nombre de vols. Cette analyse nous donne

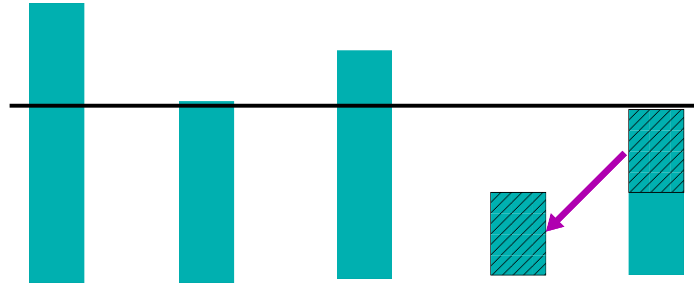


FIGURE 8.2 – Diminution du déséquilibre de travail lors d'un vol

l'espérance du nombre de vols mais aussi une borne sur la déviation par rapport à l'espérance.

Muni de cette technique d'analyse par diminution de la fonction potentielle et du théorème principal qui nous permet de résoudre de telles équations, nous examinons plusieurs modèles d'ordonnancement classiques :

– tâches indépendantes unitaires $P|p_j = 1|C_{\max}$,
– tâches indépendantes avec poids $P||C_{\max}$,
– DAG de tâches unitaires avec une seule source et degré sortant au plus 2, c'est-à-dire le modèle de la preuve de Arora *et al.* [ABP98], $P|cilkprec, p_j = 1|C_{\max}$,
– les tâches unitaires avec combinaison des requêtes de vol.

Dans chacun de ces modèles, nous montrons comment choisir la fonction potentielle et comment définir le travail présent sur un processeur.

Enfin, nous étudions, à l'aide d'un simulateur implémentant parfaitement le modèle, le comportement du nombre de vols S. L'analyse des facteurs constants montre que notre technique donne des bornes supérieures très proches des valeurs théoriques. L'analyse en pire cas sur le nombre de voleurs ne dégrade donc pas beaucoup les bornes. Nous montrons également par des tests statistiques que la distribution du temps d'exécution suit des lois connues.

Comparée à l'analyse de Arora *et al.* [ABP98], notre technique donne des bornes beaucoup plus précises et permet d'étudier des modifications de la politique de vol telles que la combinaison des requêtes de vol. De plus, notre analyse est basée sur l'équilibrage de charge entre les processeurs au moment du vol et non pas sur la diminution de la profondeur. Elle ne fait pas référence au DAG mais seulement au travail ce qui permet, par exemple, de l'appliquer directement aux programmes adaptatifs qui ne créent pas de tâches.

8.2 Discussion et perspectives

Nous avons également analysé d'autres variations sur les modèles précédents. Lorsque le travail est très déséquilibré, il peut être intéressant de biaiser les vols, normalement uniformes, vers les processeurs les plus chargés. Notre technique permet de montrer que la meilleure façon de biaiser les vols est de voler un processeur avec une probabilité proportionnelle à son travail. Nous avons aussi analysé le cas des DAG généraux (non limité à un degré sortant 2 et une seule source).

La limitation principale de notre approche est de considérer un pire cas lors du

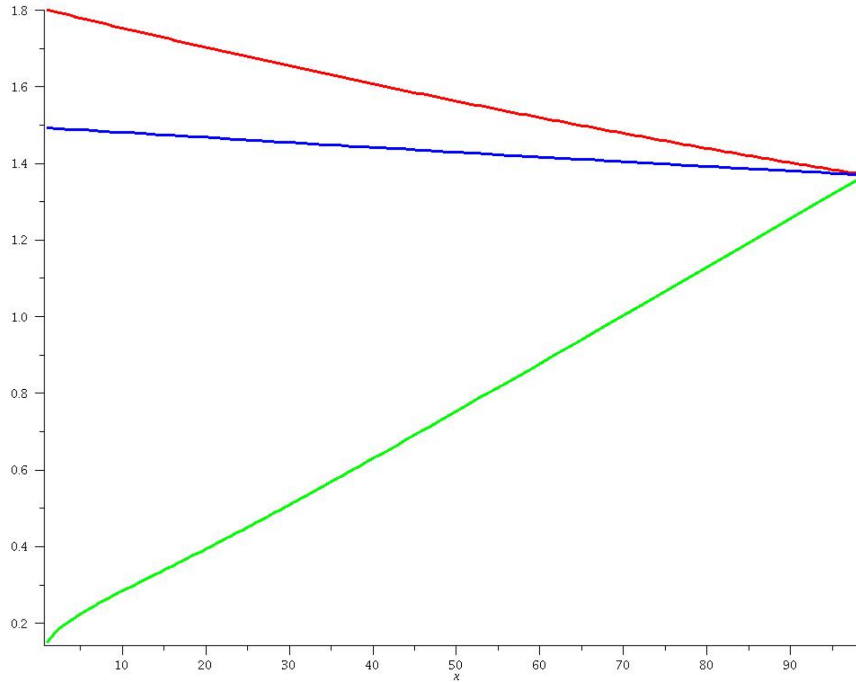


FIGURE 8.3 – Facteurs constants λ devant le nombre de vols $S = \lambda \cdot \log_2 \Phi_0$ lorsque le nombre de voleurs r_t est constant sur toute l'exécution pour le vol standard (en haut, en rouge), le vol combiné (au milieu, en bleu) et le vol biaisé (en bas, en vert). Le facteur constant varie en fonction du nombre de processeurs actifs $m - r_t$ entre 1 et 99 pour $m = 100$ processeurs.

calcul de l'espérance de la diminution de la fonction potentielle sur une étape. Ce pire cas est à la fois sur le nombre de voleurs à l'instant t mais aussi sur la répartition du travail entre les processeurs. On voit sur la figure 8.3 que le pire cas sur le nombre de requêtes de vols ne modifie pas beaucoup la borne pour le vol standard et le vol combiné (courbes du haut), c'est-à-dire le facteur constant ne change pas beaucoup en fonction du nombre de voleurs. Cependant, la variation est importante pour le vol biaisé (courbe du bas). Il serait intéressant de modifier notre technique pour arriver à prendre en compte l'évolution du nombre de voleurs.

Enfin, il serait intéressant de trouver des paramètres supplémentaires pour décrire le DAG, en complément du travail W et de la profondeur D, pour mieux caractériser le nombre de vols. En effet, des DAGs ayant le même travail et la même profondeur peuvent avoir un nombre de vols très différents. Une analyse uniquement basée sur le travail et la profondeur ne permet pas de les distinguer. Un exemple est la différence entre les ordonnancements StaticWindow et SlidingWindow du chapitre 7 qui sont identiques du point de vue du travail et de la profondeur alors que SlidingWindow nécessite beaucoup moins de vols.

Annals of Operations Research manuscript No.
(will be inserted by the editor)

Decentralized List Scheduling

Marc Tchiboukdjian · Nicolas Gast · Denis Trystram

the date of receipt and acceptance should be inserted later

Abstract Classical list scheduling is a very popular and efficient technique for scheduling jobs in parallel and distributed platforms. It is inherently centralized. However, with the increasing number of processors, the cost for managing a single centralized list becomes too prohibitive. A suitable approach to reduce the contention is to distribute the list among the computational units: each processor has only a local view of the work to execute. Thus, the scheduler is no longer greedy and standard performance guarantees are lost.

The objective of this work is to study the extra cost that must be paid when the list is distributed among the computational units. We first present a general methodology for computing the expected makespan based on the analysis of an adequate potential function which represents the load unbalance between the local lists. We obtain an equation on the evolution of the potential by computing its expected decrease in one step of the schedule. Our main theorem shows how to solve such equations to bound the makespan. Then, we apply this method to several scheduling problems, namely, for unit independent tasks, for weighted independent tasks and for tasks with precendence constraints. More precisely, we prove that the time for scheduling a global workload W composed of independent unit tasks on m processors is equal to W/m plus an additional term proportional to $\log_2 W$. We provide a lower bound which shows that this is optimal up to a constant. This result is extended to the case of weighted independent tasks. In the last setting, precedence task graphs, our analysis leads to an improvement on the bound of Arora et al (2001). We finally provide some experiments using a simulator. The distribution of the makespan is shown to fit existing probability laws. Moreover, the simulations give a better insight on the additive term whose value is shown to be around $3 \log_2 W$ confirming the tightness of our analysis.

Marc Tchiboukdjian
CNRS / CEA,DAM,DIF - Arpajon
E-mail: marc.tchiboukdjian@imag.fr

Nicolas Gast
Grenoble University
EPFL, IC-LCA2, BC 203 Bâtiment BC, Station 14, 1015 Lausanne-EPFL, Switzerland
E-mail: nicolas.gast@epfl.ch

Denis Trystram
Grenoble University, 51 av Jean-Kuntzman, 38330 Montbonnot, France
E-mail: denis.trystram@imag.fr

Keywords Scheduling · List algorithms · Work stealing

1 Introduction

1.1 Context and motivations

Scheduling is a crucial issue while designing efficient parallel algorithms on new multi-core platforms. The problem corresponds to distribute the tasks of an application (that we will called load) among available computational units and determine at what time they will be executed. The most common objective is to minimize the completion time of the latest task to be executed (called the *makespan* and denoted by $C_{\max}$). It is a hard challenging problem which received a lot of attention during the last decade (Leung, 2004). Two new books have been published recently on the topic (Drozdowski, 2009; Robert and Vivien, 2009), which confirm how active is the area.

List scheduling is one of the most popular technique for scheduling the tasks of a parallel program. This algorithm has been introduced by Graham (1969) and was used with profit in many further works (for instance the earliest task first heuristic which extends the analysis for communication delays in Hwang et al (1989), for uniform machines in Chekuri and Bender (2001), or for parallel rigid jobs in Schwiegelshohn et al (2008)). Its principle is to build a list of ready tasks and schedule them as soon as there exist available resources. List scheduling algorithms are low-cost (greedy) whose performances are not too far from optimal solutions. Most proposed list algorithms differ in the way of considering the priority of the tasks for building the list, but they always consider a centralized management of the list. However, today the parallel and distributed platforms involve more and more processors. Thus, the time needed for managing such a centralized data structure can not be ignored anymore. Practically, implementing such schedulers induces synchronization overheads when several processors access the list concurrently. Such overheads involve low-level synchronization mechanisms.

1.2 Related works

Most related works dealing with scheduling consider centralized list algorithms. However, at execution time, the cost for managing the list is neglected. To our knowledge, the only approach that takes into account this extra management cost is *work stealing* (Blumofe and Leiserson, 1999) (denoted by WS in short).

Contrary to classical centralized scheduling techniques, WS is by nature a distributed algorithm. Each processor manages its own list of tasks. When a processor becomes idle, it randomly chooses another processor and *steals* some work. To model contention overheads, processors that request work on the same remote list are in competition and only one can succeed. WS has been implemented in many languages and parallel libraries including Cilk (Frigo et al, 1998), TBB (Robison et al, 2008) and KAAPI (Gautier et al, 2007). It has been analyzed in a seminal paper of Blumofe and Leiserson (1999) where they show that the expected makespan of series-parallel precedence graph with W unit tasks on m processors is bounded by $\mathbb{E}[C_{\max}] \leq W/m + \mathrm{O}(D)$ where D is the critical path of the graph (its depth). This analysis has been improved in Arora et al (2001) using a proof based on a potential function. The case of varying processor speeds has been analyzed in Bender and Rabin (2002). However, in all these previous analyses, the precedence graph is constrained

to have only one source and out-degree at most 2 which does not easily model the basic case of independent tasks. Simulating independent tasks with a binary tree of precedences gives a bound of $W/m + \mathrm{O}(\log W)$ as a complete binary tree of W nodes has a depth of $D \leq \log_2 W$. However, with this approach, the structure of the binary tree dictates which tasks are stolen. Our approach achieves a bound of the same order with a better constant and processors are free to choose which tasks to steal. Notice that there exist other ways to analyze work stealing where the work generation is probabilist and that targets steady state results (Berenbrink et al, 2003; Mitzenmacher, 1998; Gast and Gaujal, 2010).

Another related approach which deals with distributed load balancing is *balls into bins* games (Azar et al, 1999; Berenbrink et al, 2008). The principle is to study the maximum load when n balls are randomly thrown into m bins. This is a simple distributed algorithm which is different from the scheduling problems we are interested in. First, it seems hard to extend this kind of analysis for tasks with precendence constraints. Second, as the load balancing is done in one phase at the beginning, the cost of computing the schedule is not considered. Adler et al (1995) study parallel allocations but still do not take into account contention on the bins. Our approach, like in WS, considers contention on the lists.

Some works have been proposed for the analysis of algorithms in data structures and combinatorial optimization (including variants of scheduling) using potential functions. Our analysis is also based on a potential function representing the load unbalance between the local queues. This technique has been successfully used for analyzing convergence to Nash equilibria in game theory (Berenbrink et al, 2007), load diffusion on graphs (Berenbrink et al, 2009) and WS (Arora et al, 2001).

1.3 Contributions

List scheduling is centralized in nature. The purpose of this work is to study the effects of decentralization on list scheduling. The main result is a new framework for analyzing distributed list scheduling algorithms (DLS). Based on the analysis of the load balancing between two processors during a work request, it is possible to deduce the total expected number of work requests and then, to derive a bound on the expected makespan.

This methodology is generic and it is applied in this paper on several relevant variants of the scheduling problem.

- We first show that the expected makespan of DLS applied on W unit independent tasks is equal to the absolute lower bound W/m plus an additive term in $3.65 \log_2 W$. We propose a lower bound which shows that the analysis is tight up to a constant factor. This analysis is refined and applied to several variants of the problem. In particular, a slight change on the potential function improves the multiplicative factor from 3.65 to 3.24. Then, we study the possibility of processors to cooperate while requesting some tasks in the same list. Finally, we study the initial repartition of the tasks and show that a balanced initial allocation induces less work requests.
- Second, the previous analysis is extended to the weighted case of any unknown processing times. The analysis achieves the same bound as before with an extra term involving $p_{\max}$ (the maximal value of the processing times).
- Third, we provide a new analysis for the WS algorithm of Arora et al (2001) for scheduling DAGs that improves the bound on the number of work requests from $32mD$ to $5.5mD$.
- Fourth, we developed a complete experimental campaign that gives statistical evidence that the makespan of DLS follows known probability distributions depending on the

considered variant. Moreover, the experiments show that the theoretical analysis for independent tasks is almost tight: the overhead to W/m is less than 37% away of the exact value.

1.4 Content

We start by introducing the model and we recall the analysis for classical list scheduling in Section 2. Then, we present the principle of the analysis in Section 3 and we apply this analysis on unit independent tasks in Section 4. Section 5 discusses variations on the unit tasks model: improvements on the potential function and cooperation among thieves. We extend the analysis for weighted independent tasks in Section 6 and for tasks with dependencies in Section 7. Finally, we present and analyze simulation experiments in Section 8.

2 Model and notations

2.1 Platform and workload characteristics

We consider a parallel platform composed of m identical processors and a workload of n tasks with processing times p_j. The total work of the computation is denoted by $W = \sum_{j=1}^{n} p_j$. The tasks can be independent or constrained by a directed acyclic graph (DAG) of precedences. In this case, we denote by D the critical path of the DAG (its depth). We consider an online model where the processing times and precedences are discovered during the computation. More precisely, we learn the processing time of a task when its execution is terminated and we discover new tasks in the DAG only when all their precedences have been satisfied. The problem is to study the maximum completion time (*makespan* denoted by $C_{\max}$) taking into account the scheduling cost.

2.2 Centralized list scheduling

Let us recall briefly the principle of list scheduling as it was introduced by Graham (1969). The analysis states that the makespan of any list algorithm is not greater than twice the optimal makespan. One way of proving this bound is to use a geometric argument on the Gantt chart: $m \cdot C_{\max} = W + S_{\text{idle}}$ where the last term is the surface of idle periods (represented in grey in figure 1).

Depending on the scheduling problem (with or without precedence constraints, unit tasks or not), there are several ways to compute S_{idle}. With precedence constraints, $S_{\text{idle}} \leq (m-1) \cdot D$. For independent tasks, the results can be written as $S_{\text{idle}} \leq (m-1) \cdot p_{\max}$ where $p_{\max}$ is the maximum of the processing times. For unit independent tasks, it is straightforward to obtain an optimal algorithm where the load is evenly balanced. Thus $S_{\text{idle}} \leq m-1$, *i.e.* at most one slot of the schedule contains idle times.

2.3 Decentralized list scheduling

When the list of ready tasks is distributed among the processors, the analysis is more complex even in the elementary case of unit independent tasks. In this case, the extra S_{idle} term

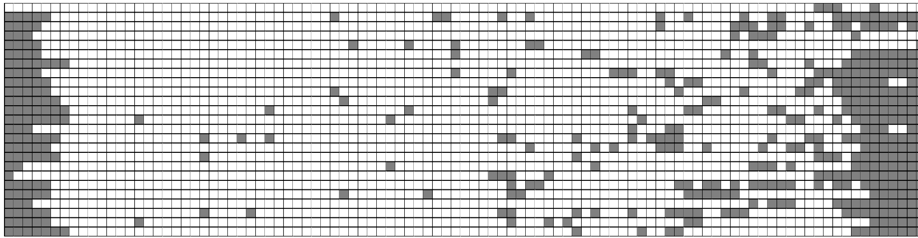


Fig. 1 A typical execution of $W = 2000$ unit independent tasks on $m = 25$ processors using distributed list scheduling. Grey area represents idle times due to steal requests.

is induced by the distributed nature of the problem. Processors can be idle even when ready tasks are available. Fig. 1 is an example of a schedule obtained using distributed list scheduling which shows the complicated repartition of the idle times S_{idle}.

2.4 Model of the distributed list

We now describe precisely the behavior of the distributed list. Each processor i maintains its own local queue Q_i of tasks ready to execute. At the beginning of the execution, ready tasks can be arbitrarily spread among the queues. While Q_i is not empty, processor i picks a task and executes it. When this task has been executed, it is removed from the queue and another one starts being processed. When Q_i is empty, processor i sends a *steal request* to another processor k chosen uniformly at random. If Q_k is empty or contains only one task (currently executed by processor k), then the request fails and processor i will send a new request at the next time step. If Q_k contains more than one task, then i is given half of the tasks and it will restart a normal execution at the next step. To model the contention on the queues, no more than one steal request per processor can succeed in the same time slot. If several requests target the same processor, a random one succeeds and all the others fail. This assumption will be relaxed in Section 5.2. A steal request is said *successful* if the target queue contains more than one task and the request is not aborted due to contention. In all the other cases, the steal request is said *unsuccessful*.

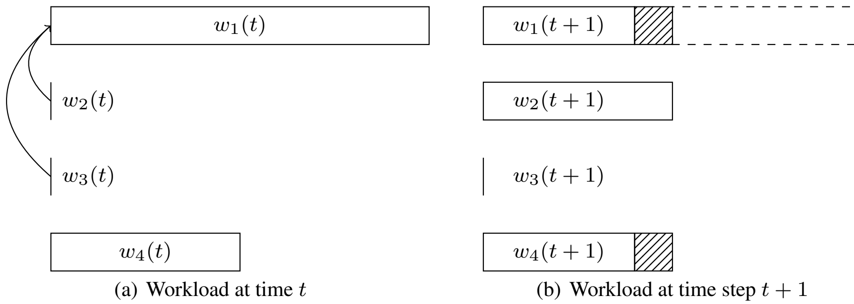
This is a high level model of a distributed list but it accurately models the case of independent tasks and the WS algorithm of Arora et al (2001). We justify here some choices of this model. There is no explicit communication cost since WS algorithms most often target shared memory platforms. In addition, a steal request is done in constant time independently of the amount of tasks transfered. This assumption is not restrictive as the description of a large number of tasks can be very short. In the case of independent tasks, a whole subpart of an array of tasks can be represented in a compact way by the range of the corresponding indices, each cell containing the effective description of a task (a STL transform in Traoré et al (2008)). For more general cases with precedence constraints, it is usually enough to transfer a task which represents a part of the DAG. More details on the DAG model are provided in Section 7. Finally, there is no contention between a processor executing a task from its own queue and a processor stealing in the same queue. Indeed, one can use queue data structures allowing these two operations to happen concurrently (Frigo et al, 1998).

2.5 Properties of the work

At time t, let $w_i(t)$ represent the amount of work in queue Q_i (*cf.* Fig. 2). $w_i(t)$ may be defined as the sum of processing times of all tasks in Q_i as in Section 4 but can differ as in Sections 6 and 7. In all cases, the definition of $w_i(t)$ satisfies the following properties.

1. When $w_i(t) > 0$, processor i is active and executes some work: $w_i(t+1) \leq w_i(t)$.
2. When $w_i(t) = 0$, processor i is idle and send a steal request to a random processor k. If the steal request is successful, a certain amount of work is transfered from processor k to processor i and we have $\max\{w_i(t+1), w_k(t+1)\} < w_k(t)$.
3. The execution terminates when there is no more work in the system, *i.e.* $\forall i, w_i(t) = 0$.

We also denote the total amount of work on all processors by $w(t) = \sum_{i=1}^{m} w_i(t)$ and the number of processors sending steal requests by $r_t \in [0, m-1]$. Notice that when $r_t = m$, all queues are empty and thus the execution is complete.



(a) Workload at time t (b) Workload at time step $t+1$

Fig. 2 Evolution of the workload of the different processors during a time step. At time t, processors 2 and 3 are idle and they both choose processor 1 to steal from. At time $t+1$, only processor 2 succeed in stealing some of the work of processor 1. The work is split between the two processors. Processors 1 and 4 both execute some work during this time step (represented by a shaded zone).

3 Principle of the analysis and main theorem

This section presents the principle of the analysis. The main result is Theorem 1 that gives bounds on the expectation of the steal requests done by the schedule as well as the probability that the number of work requests exceeds this bound. As a processor is either executing or requesting work, the number of work requests plus the total amount of tasks to be executed is equal to $m \cdot C_{\max}$, where $C_{\max}$ is the total completion time. The makespan can be derived from the total number of work requests:

$$C_{\max} = \frac{W}{m} + \frac{R}{m}. \quad (1)$$

The main idea of our analysis is to study the decrease of a potential Φ_t.The potential Φ_t depends on the load on all processors at time t, $\mathbf{w}(t)$. The precise definition of Φ_t varies depending on the scenario (see Sections 4 to 7). For example, the potential function used in Section 4 is $\Phi_t = \sum_{i=1}^{m} (w_i(t) - w(t)/m)^2$. For each scenario, we will prove that the diminution of the potential during one time step depends on the number of steal requests,

r_t. More precisely, we will show that there exists a function $h : \{0 \dots m\} \to [0;1]$ such that the average value of the potential at time $t+1$ is less than $\Phi_t/h(r_t)$.

Using the expected diminution of the potential, we derive a bound on the number of steal requests until Φ_t becomes less than one, $R = \sum_{s=0}^{\tau-1} r_s$, where τ denotes the first time that Φ_t is less than 1. If all r_t were equal to r and the potential decrease was deterministic, the number of time steps before $\Phi_t \leq 1$ would be $\left\lceil \log \Phi_0 / \log h(r) \right\rceil$ and the number of steal requests would be $r/\log h(r) \log \Phi_0$. As r can vary between 1 and m, the worst case for this bound is $m\lambda \cdot \log \Phi_0$, where $m\lambda = \max_{1 \leq r \leq m} r/\log(h(r))$.

The next theorem shows that number of steal requests is indeed bounded by $m\lambda \log \Phi_0$ plus an additive term due to the stochastic nature of Φ_t. The fact that λ corresponds to the worst choice of r_t at each time step makes the bound looser than the real constant. However, we show in Section 8 that the gap between the obtained bound and the values obtained by simulation is small. Moreover, the computation of the constant λ is simple and makes this analysis applicable in several scenarios, such as the ones presented in Sections 4 to 7.

In the following theorem and its proof, we use the following notations. $\mathscr{F}_t$ denotes the knowledge of the system up to time t (namely, the filtration associated to the process $\mathbf{w}(t)$). For a random variable X, the conditional expectation of A knowing $\mathscr{F}_t$ is denoted $\mathbb{E}[X \mid \mathscr{F}_t]$. Finally, the notation $\mathbf{1}_A$ denotes the random variable equal to 1 if the event A is true and 0 otherwise. In particular, this means that the probability of an event A is $\mathbb{P}\{A\} = \mathbb{E}[\mathbf{1}_A]$.

Theorem 1 *Assume that there exists a function $h : \{0 \dots m\} \to [0,1]$ such that the potential satisfies:*

$$\mathbb{E}[\Phi_{t+1} \mid \mathscr{F}_t] \leq h(r_t) \cdot \Phi_t.$$

Let Φ_0 denotes the potential at time 0 and λ be defined as:

$$\lambda \stackrel{\text{def}}{=} \max_{1 \leq r \leq m} \frac{r}{-m \log_2(h(r))}$$

Let τ be the first time that Φ_t is less than 1, $\tau \stackrel{\text{def}}{=} \min\{t : \Phi_t < 1\}$. The number of steal requests until τ, $R = \sum_{s=0}^{\tau-1} r_s$, satisfies:

(i) $\mathbb{P}\{R \geq m \cdot \lambda \cdot \log_2 \Phi(0) + m + u\} \leq 2^{-u/(m\cdot\lambda)}$

(ii) $\mathbb{E}[R] \leq m \cdot \lambda \cdot \log_2 \Phi(0) + m(1 + \frac{\lambda}{\ln 2})$.

Proof For two time steps $t \leq T$, we call R_t^T the number of steal requests between t and T:

$$R_t^T \stackrel{\text{def}}{=} \sum_{s=t}^{\min\{\tau,T\}-1} r_s.$$

The number of steal requests until $\Phi_t < 1$ is $R = \sum_{s=0}^{\tau-1} r_s = \lim_{T\to\infty} R_0^T$.

We show by a backward induction on t that for all $t \leq T$:

$$\text{if } \Phi_t \geq 1, \text{then } \forall u \in \mathbb{R} : \mathbb{E}\left[\mathbf{1}_{R_t^T \geq m\cdot\lambda\cdot\log_2 \Phi_t + m + u} \mid \mathscr{F}_t\right] \leq 2^{-u/(m\cdot\lambda)}. \quad (2)$$

For $t{=}T$, $R_T^T = 0$ and $\mathbb{E}\left[\mathbf{1}_{R_t^T \geq m\cdot\lambda\cdot\log_2 \Phi_t + m + u} \mid \mathscr{F}_t\right] = 0$. Thus, (2) is true for $t{=}T$.

Assume that (2) holds for some $t+1 \leq T$ and suppose that $\Phi_t \geq 1$. Let $u > 0$ (if $u \leq 0$ ...). Since $R_t^T = r_t + R_{t+1}^T$, the probability $\mathbb{P}\left\{R_t^T \geq m \cdot \lambda \cdot \log_2 \Phi_t + m + u \mid \mathscr{F}_t\right\}$ is equal to

$$\mathbb{E}\left[\mathbf{1}_{R_t^T \geq m\lambda \log_2 \Phi_t + m + u} \mid \mathscr{F}_t\right] = \mathbb{E}\left[\mathbf{1}_{r_t + R_{t+1}^T \geq m\lambda \log_2 \Phi_t + m + u} \mid \mathscr{F}_t\right] \tag{3}$$

$$= \mathbb{E}\left[\mathbf{1}_{r_t + R_{t+1}^T \geq m\lambda \log_2 \Phi_t + m + u} \mathbf{1}_{\Phi_{t+1} \geq 1} \mid \mathscr{F}_t\right] \tag{4}$$

$$+ \mathbb{E}\left[\mathbf{1}_{r_t + R_{t+1}^T \geq m\lambda \log_2 \Phi_t + m + u} \mathbf{1}_{\Phi_{t+1} < 1} \mid \mathscr{F}_t\right] \tag{5}$$

If $\Phi_{t+1} < 1$, then $R_{t+1}^T = 0$. Since $m \geq r_t$ and $\Phi_t \geq 1$, $m\lambda \log_2 \Phi_t + m + u - r_t \geq 0$. This shows that the term of Equation (5) is equal to zero. (4) is the probability that R_{t+1}^T is greater than

$$m\lambda \log_2 \Phi_t + m + u - r_t = m\lambda \log_2 \Phi_{t+1} + m + (u - r_t - m\lambda \log(\Phi_{t+1}/\Phi_t)$$

Therefore, using the induction hypothesis, (4) is equal to

$$\begin{aligned}
\mathbb{E}\left[\mathbf{1}_{R_{t+1}^T \geq m\lambda \log_2 \Phi_t + m + u - r_t} \mathbf{1}_{\Phi_{t+1} > 1} \mid \mathscr{F}_t\right] &= \mathbb{E}\left[2^{-\frac{u - r_t - m\lambda \log(\Phi_{t+1}/\Phi_t)}{m\lambda}} \mathbf{1}_{\Phi_{t+1} > 1} \mid \mathscr{F}_t\right] \\
&= 2^{-\frac{u - r_t}{m\lambda}} \mathbb{E}\left[\frac{\Phi_{t+1}}{\Phi_t} \mathbf{1}_{\Phi_{t+1} > 1} \mid \mathscr{F}_t\right] \\
&= 2^{-\frac{u - r_t}{m\lambda}} h(r_t) \\
&= 2^{-\frac{u}{m\lambda}} 2^{r_t/\lambda + \log_2(h(r_t))},
\end{aligned}$$

where at the first line we used both the fact that for a random variable X, $\mathbb{E}[X \mid \mathscr{F}_t] = \mathbb{E}[\mathbb{E}[X \mid \mathscr{F}_{t+1}] \mid \mathscr{F}_t]$ and the induction hypothesis.

If $r_t = 0$, $2^{r_t/\lambda + \log_2(h(r_t))} = h(r_t) \leq 1$. Otherwise, by definition of $\lambda = \max_{1 r \leq m} r / -\log(h(r))$, $r_t/\lambda + \log_2(h(r_t)) \leq 0$ and $2^{r_t/\lambda + \log_2(h(r_t))} \leq 1$. This shows that (2) holds for t. Therefore, by induction on t, this shows that (2) holds for $t = 0$: for all $u \geq 0$:

$$\mathbb{E}\left[\mathbf{1}_{R_0^T \geq m \cdot \lambda \cdot \log_2 \Phi_t + m + u} \mid \mathscr{F}_0\right] \leq 2^{-u/(m \cdot \lambda)}$$

As $r_t \geq 0$, the sequence $(R_0^T)_T$ is increasing and converges to R. Therefore, the sequence $\mathbf{1}_{R_0^T \geq m \cdot \lambda \cdot \log_2 \Phi_0 + m + u}$ is increasing in T and converges to $\mathbf{1}_{R \geq m \cdot \lambda \cdot \log_2 \Phi_0 + m + u}$. Thus, by Lebesgue's monotone convergence theorem, this shows that

$$\mathbb{P}\{R \geq m \cdot \lambda \cdot \log_2 \Phi_0 + m + u\} = \lim_{T \to \infty} \mathbb{E}\left[\mathbf{1}_{R_0^T \geq m \cdot \lambda \cdot \log_2 \Phi_0 + m + u}\right] \leq 2^{-\frac{u}{m\lambda}}.$$

The second part of the theorem *(ii)* is a direct consequence of *(i)*. Indeed,

$$\begin{aligned}
\mathbb{E}[R] &= \int_0^\infty \mathbb{P}\{R \geq u\} \, du \\
&\leq m \cdot \lambda \cdot \log_2 \Phi_0 + m + \int_0^\infty \mathbb{P}\{R \geq m \cdot \lambda \cdot \log_2 \Phi_0 + m + u\} \, du \\
&\leq m \cdot \lambda \cdot \log_2 \Phi_0 + m + \int_0^\infty 2^{-\frac{u}{m\lambda}} \, du \\
&\leq m \cdot \lambda \cdot \log_2 \Phi_0 + m(1 + \frac{\lambda}{\ln 2}).
\end{aligned}$$

□

4 Unit independent tasks

We apply the analysis presented in the previous section for the case of independent unit tasks. In this case, each processor i maintains a local queue Q_i of tasks to execute. At every time slot, if the local queue Q_i is not empty, processor i picks a task and executes it. When Q_i is empty, processor i sends a steal request to a random processor j. If Q_j is empty or contains only one task (currently executed by processor j), then the request fails and processor i will have to send a new request at the next slot. If Q_j contains more than one task, then i is given half of the tasks (after that the task executed at time t by processor j has been removed from Q_j). The amount of work on processor i at time t, $w_i(t)$, is the number of tasks in $Q_i(t)$. At the beginning of the execution, $w(0) = W$ and tasks can be arbitrarily spread among the queues.

4.1 Potential function and expected decrease

Applying the method presented in Section 3, the first step of the analysis is to define the potential function and compute the potential decrease when a steal occurs. For this example, $\Phi(t)$ is defined by:

$$\Phi(t) = \sum_{i=1}^{m} \left(w_i(t) - \frac{w(t)}{m} \right)^2 = \sum_{i=1}^{m} w_i(t)^2 - \frac{w^2(t)}{m}.$$

This potential represents the load unbalance in the system. If all queues have the same load $w_i(t) = w(t)/m$, then $\Phi(t) = 0$. $\Phi(t) \leq 1$ implies that there is at most one processor with at most one more task than the others. In that case, there will be no steal until there is just one processor with 1 task and all others idle. Moreover, the potential function is maximal when all the work is concentrated on a single queue. That is $\Phi(t) \leq w(t)^2 - w(t)^2/m \leq (1 - 1/m)w^2(t)$.

Three events contribute to a variation of potential: successful steals, tasks execution and decrease of $w^2(t)/m$.

1. If the queue i has $w_i(t) \geq 1$ tasks and it receives one or more steal requests, it chooses a processor j among the thieves. At time $t + 1$, i has executed one task and the rest of the work is split between i and j. Therefore,

   $$w_i(t+1) = \left\lceil (w_i(t) - 1)/2 \right\rceil \quad \text{and} \quad w_j(t+1) = \left\lfloor (w_i(t) - 1)/2 \right\rfloor.$$

   Thus, we have:

   $$w_i(t+1)^2 + w_j(t+1)^2 = \left\lceil (w_i(t) - 1)/2 \right\rceil^2 + \left\lfloor (w_i(t) - 1)/2 \right\rfloor^2 \leq w_i(t)^2/2 - w_i(t) + 1.$$

   Therefore, this generates a difference of potential of

   $$\delta_i(t) \geq w_i(t)^2/2 + w_i(t) - 1. \tag{6}$$

2. If i has $w_i(t) \geq 1$ tasks and receives zero steal requests, it potential goes from $w_i(t)^2$ to $(w_i(t) - 1)^2$, generating a potential decrease of $2w_i(t) - 1$.
3. As there are $m - r_t$ active processors, $(\sum_{i=1}^{m} w_i(t))^2/m$ goes from $w(t)^2/m$ to $w(t+1)^2 = (w(t) - m + r)^2/m$, generating a potential increase of $2(m - r_t)w(t)/m - (m - r_t)^2/m$.

Recall that at time t, there are r_t processors that send steal requests. A processor i receives zero steal requests if the r_t thieves choose another processor. Each of these events is independent and happens with probability $(m-2)/(m-1)$. Therefore, the probability for the processor to receive one or more steal requests is $q(r_t)$ where

$$q(r_t) = 1 - \left(1 - \frac{1}{m-1}\right)^{r_t}.$$

If $\Phi_t{=}\Phi$ and $r_t{=}r$, by summing the expected decrease on each active processor δ_i, the expected potential decrease is greater than:

$$\sum_{i/w_i(t)>0} \left[q(r) \underbrace{\left(\frac{w_i(t)^2}{2} + w_i(t){-}1\right)}_{\geq \delta_i} + (1-q(r))(2w_i(t){-}1) \right] - 2w(t)\frac{m-r}{m} + \frac{(m-r)^2}{m}$$
$$= \left[\sum_{i/w_i(t)>0} \frac{q(r)}{2} w_i(t)^2 \right] - q(r)w(t) + 2w(t) - (m-r) - 2w(t)\frac{m-r}{m} + \frac{(m-r)^2}{m}.$$

Using that $2w(t) - 2w(t)\frac{m-r}{m} = 2w(t)\frac{r}{m}$, that $-(m-r) + \frac{(m-r)^2}{m} = -(m-r)\frac{r}{m}$ and that $\sum w_i(t)^2 = \Phi + w(t)^2$, this equals:

$$\begin{aligned}
&\frac{q(r)}{2}\Phi + \frac{q(r)}{2}\frac{w(t)^2}{m} - q(r)w(t) + 2w(t)\frac{r}{m} - (m-r)\frac{r}{m} \\
&\quad = \frac{q(r)}{2}\Phi + \frac{q(r)}{2}\frac{w(t)^2}{m} - q(r)w(t) + \frac{r}{m}(2w(t) - m + r) \\
&\quad = \frac{q(r)}{2}\Phi + \frac{q(r)w(t)}{2}\left(\frac{w(t)}{m} - 2 + \frac{2r}{mq(r)}\right) + \frac{r}{m}\left(w(t) - m + r\right).
\end{aligned}$$

By concavity of $x \mapsto (1-(1-x)^r)$, $(1-(1-x)^r) \leq r \cdot x$. This shows that $q(r) = 1 - (1 - \frac{1}{m-1})^r \leq r/(m-1)$. Thus, $r/q(r) \geq m-1$. Moreover, as $m-r$ is the number of active processors, $w \geq m-r$ (each processor has at least one task). This shows that the expected decrease of potential is greater than:

$$\frac{q(r)}{2}\Phi + \frac{q(r)w(t)}{2}\left(\frac{w(t)}{m} - 2 + 2\frac{m-1}{m}\right) = \frac{q(r)}{2}\Phi + \frac{q(r)w(t)}{2m}(w(t)-2).$$

If $w(t) \geq 2$, then the expected decrease of potential is greater than $q(r_t)\Phi_t/2$. If $w(t) < 2$, this means that $w(t) = 1$ and $w(t+1) = 0$ and therefore $\Phi_{t+1} = 0$. Thus, for all t:

$$\mathbb{E}\left[\Phi_{t+1} \mid \mathscr{F}_t\right] \leq \left(1 - \frac{q(r_t)}{2}\right) \cdot \Phi_t. \tag{7}$$

4.2 Bound on the makespan

Using Theorem 1 of the previous section, we can solve equation (7) and conclude the analysis.

Theorem 2 *Let $C_{\max}$ be the makespan of $W = n$ unit independent tasks scheduled by DLS and $\Phi_0 \stackrel{\text{def}}{=} \sum_i (w_i - \frac{W}{m})^2$ the potential when the schedule starts. Then:*

(i) $\mathbb{E}[C_{\max}] \leq \frac{W}{m} + \frac{1}{1-\log_2(1+\frac{1}{e})} \cdot \Big(\log_2 \Phi_0 + \frac{1}{\ln 2}\Big) + 1$

(ii) $\mathbb{P}\Big\{C_{\max} \geq \frac{W}{m} + \frac{1}{1-\log_2(1+\frac{1}{e})} \cdot \Big(\log_2 \Phi_0 + \log_2 \frac{1}{\epsilon}\Big) + 1\Big\} \leq \epsilon$

In particular:

(iii) $\mathbb{E}[C_{\max}] \leq \frac{W}{m} + \frac{2}{1-\log_2(1+\frac{1}{e})} \cdot \Big(\log_2 W + \frac{1}{2\ln 2}\Big) + 1$

These bounds are optimal up to a constant factor in $\log_2 W$.

Proof Equation (7) shows that $\mathbb{E}[\Phi_{t+1}|\mathscr{F}_t] \leq g(r_t)\Phi_t$ with $g(r) = 1 - q(r)/2$. Defining $\Phi'_t = \Phi_t/(1-1/(m-1))$, the potential function Φ'_t also satisfies (7). Therefore, Φ'_t satisfies the conditions of Theorem 1. This shows that the number of work requests R until $\Phi'_t < 1$ satisfies

$$\mathbb{E}[R] \leq m \cdot \lambda \log_2(\Phi_0) + m\Big(1 + \frac{\lambda}{\ln 2}\Big),$$

with $\lambda = \max_{1 \leq r \leq m-1} r/(-m \log_2 h(r))$. One can show that $r/(-m \log_2 h(r))$ is decreasing in r. Thus its minimum is attained for $r = 1$. This shows that $\lambda \leq 1/(1 - \log_2(1 + \frac{1}{e}))$.

The minimal non zero-value for Φ_t is when one processor has one task and the others zero. In that case, $\Phi_t = 1 - 1/(m-1)$. Therefore, when $\Phi'_t < 1$, this means that $\Phi_t = 0$ and the schedule is finished.

As pointed out in Equation (1), at each time step of the schedule, a processor is either computing one task or stealing work. Thus, the number of steal requests plus the number of tasks to be executed is equal to $m \cdot C_{\max}$, *i.e.* $m \cdot C_{\max} = W + R$. This shows that

$$\mathbb{E}[C_{\max}] \leq \frac{W}{m} + \frac{1}{1-\log_2(1+\frac{1}{e})} \cdot \Big(\log_2 \Phi_0 + \frac{1}{\ln 2}\Big) + 1.$$

This concludes the proof of *(i)*. The proof of the *(i)* applies *mutatis mutandis* to prove the bound in probability *(ii)* using Theorem 1 *(ii)*.

We now give a lower bound for this problem. Consider $W = 2^{k+1}$ tasks and $m = 2^k$ processors, all the tasks being on the same processor at the beginning. In the best case, all steal requests target processors with highest loads. In this case the makespan is $C_{\max} = k + 2$: the first $k = \log_2 m$ steps for each processor to get some work; one step where all processors are active; and one last step where only one processor is active. In that case, $C_{\max} \geq \frac{W}{m} + \log_2 W - 1$. □

This theorem shows that the factor before $\log_2 W$ is bounded by 1 and $2/(1 - \log_2(1 + 1/e)) < 3.65$. Simulations reported in Section 8 seem to indicate that the factor of $\log_2 W$ is slightly less than 3.65. This shows that the constants obtained by our analysis are sharp.

4.3 Influence of the initial repartition of tasks

In the worst case, all tasks are in the same queue at the beginning of the execution and $\Phi_0 = (W - W/m)^2 \leq W^2$. This leads to a bound on the number of work requests in $3.65m \log_2 W$ (see the item *(iii)* of Theorem 2). However, using bounds in terms of Φ_0, our analysis is able to capture the difference for the number of work requests if the initial repartition is more balanced. One can show that a more balanced initial repartition ($\Phi_0 \ll W^2$) leads to fewer steal requests on average.

Suppose for example that the initial repartition is a balls-and-bins assignment: each tasks is assigned to a processor at random. In this case, the initial number of tasks in queue i, $w_i(0)$, follows a binomial distribution $\mathcal{B}(W, 1/m)$. The expected value of Φ_0 is:

$$\mathbb{E}\left[\Phi_0\right] = \sum_i \mathbb{E}\left[w_i^2\right] - \frac{W^2}{m} = \sum_i \Big(\mathrm{Var}\left[w_i\right] + \mathbb{E}\left[w_i\right]^2\Big) - \frac{W^2}{m} = \Big(1 - \frac{1}{m}\Big) \cdot W$$

Since the number of work requests is proportional to $\log_2 \Phi_0$, this initial repartition of tasks reduces the number of steal requests by a factor of 2 on average. This leads to a better bound on the makespan in $W/m + 1.83 \log_2 W + 3.63$.

5 Going further on the unit tasks model

In this section, we provide two different analysis of the model of unit tasks of the previous section. We first show how the use of a different potential function $\Phi_t = \sum_i w_i(t)^\nu$ (for some $\nu > 1$) leads to a better bound on the number of work requests. Then we show how cooperation among thieves leads to a reduction of the bound on the number of work requests by 12%. The later is corroborated by our simulation that shows a decrease on the number of work requests between 10% and 15%.

5.1 Improving the analysis by changing the potential function

We consider the same model of unitary tasks as in Section 4. The potential function of our system is defined as

$$\Phi_t = \sum_{i=1}^{m} w_i(t)^\nu,$$

where $\nu > 1$ is a constant factor.

When an idle processor steals a processor with $w_i(t)$ tasks, the potential decreases by

$$\begin{aligned}\delta_i = w_i(t)^\nu - \left\lceil \frac{w_i(t)-1}{2} \right\rceil^\nu + \left\lfloor \frac{w_i(t)-1}{2} \right\rfloor^\nu &\geq w_i(t)^\nu - \left\lfloor \frac{w_i(t)}{2} \right\rfloor^\nu + \left\lfloor \frac{w_i(t)}{2} \right\rfloor^\nu \\ &\geq \Big(1 - 2^{1-\nu}\Big)\, w_i(t)^\nu.\end{aligned}$$

This shows that the expected value of the potential at time $t+1$ is

$$\mathbb{E}\left[\Phi_{t+1}\right] \leq (1 - q(r)(1 - 2^{1-\nu})) \cdot \Phi_t.$$

where $q(r)$ is the probability for a processor to receive at least one work request when r processors are stealing, $q(r) = 1 - \left(1 - \frac{1}{m-1}\right)^r$.

Following the analysis of the previous part, and as $\Phi_0 \leq W^\nu$ the expected makespan is bounded by:

$$\frac{W}{m} + \lambda(\nu) \cdot \Big(\log \Phi_0 + 1 + \frac{1}{\ln 2}\Big) \leq \frac{W}{m} + \nu\lambda(\nu) \cdot \Big(\log W + 1 + \frac{1}{\ln 2}\Big),$$

where $\lambda(\nu)$ is a constant depending on ν equal to:

$$\lambda(\nu) \stackrel{\text{def}}{=} \max_r \Big\{ \frac{r}{-\log_2(1 - q(r)(1 - 2^{1-\nu}))} \Big\} \qquad (8)$$

As for $\nu = 2$ of Section 4, it can be shown the maximum of Equation 8 is attained for $r = m - 1$.

The constant factor in front of $\log W$ is $\nu\lambda(\nu)$. Numerically, the minimum of $\nu\lambda(\nu)$ is for $\nu \approx 2.94$ and is less than 3.24.

Theorem 3 *Let $C_{\max}$ be the makespan of $W = n$ unit independent tasks scheduled DLS. Then:*

$$\mathbb{E}\left[C_{\max}\right] \leq \frac{W}{m} + 3.24 \cdot \left(\log_2 W + \frac{1}{2\ln 2}\right) + 1$$

In Section 4, we have shown that the makespan was bounded by

$$\frac{W}{m} + 2\lambda(2) \cdot \left(\log_2 \Phi_0 + \frac{1}{\ln 2}\right) + 1 \leq \frac{W}{m} + 3.65 \cdot \left(\log_2 W + \frac{1}{2\ln 2}\right) + 1.$$

Theorem 3 improves the constant factor in front of $\log_2 W$. However, we loose the information of the initial repartition of tasks Φ_0.

5.2 Cooperation among thieves

In this section, we modify the protocol for managing the distributed list. Previously, when $k > 1$ steal requests were sent on the same processor, only one of them could be served due to contention on the list. We now allow the k requests to be served in unit time. This model has been implemented in the middleware Kaapi (Gautier et al, 2007). When k steal requests target the same processor, the work is divided into $k+1$ pieces. In practice, allowing concurrent thieves increase the cost of a steal request but we neglect this additional cost here. We assume that the k concurrent steal requests can be served in unit time. We study the influence of this new protocol on the number of steal requests in the case of unit independent tasks.

We define the potential of the system at time t to be:

$$\Phi(t) = \sum_{i=1}^{m} \Big(w_i(t)^\nu - w_i(t)\Big).$$

Let us first compute the decrease of the potential when processor i receives $k \geq 1$ steal requests. If $w_i(t) > 0$, it can be written $w_i(t) = (k+1)q + b$ with $0 \leq b < k+1$. We neglect the decrease of potential due to the execution tasks ($\nu > 1$ implies that execution of tasks decreases the potential).

After one time step and k steal requests, the work will be divided into r parts with $q+1$ tasks and $k+1-r$ parts with q tasks. $\sum_i w_i(t)$ does not vary during the stealing phase. Therefore, the difference of potential due to these k work requests is

$$\delta_i^k = ((k+1)q + b)^\nu - b(q+1)^\nu - (k+1-b)q^\nu.$$

Let us denote $\alpha \stackrel{\text{def}}{=} b/(k+1) \in [0;1)$ and let $f(x) = (x+\alpha)^\nu + (1-2^{1-\nu})(x+\alpha) - (1-\alpha)x^\nu - \alpha(x+1)^\nu$. The first derivative of f is $f'(x) = \nu(x+\alpha)^{\nu-1} + (1-2^{1-\nu}) - \nu(1-\alpha)x^{\nu-1} - \alpha(x+1)^{\nu-1}$ and the derivative of f' is $f''(x) = \nu(1-\nu)((x+\alpha)^{\nu-2} - (1-\alpha)x^{\nu-2} - \alpha(x+1)^{\nu-2}$. As $\nu < 3$, the function $x \mapsto x^{\nu-2}$ is concave which implies than $f''(x) \geq 0$. Therefore, f' is increasing. Moreover, $f'(0) = \nu(\alpha^{\nu-1} - \alpha) + (1 - 2^{1-\nu}) \geq 0$. This shows that for all x, $f'(x) \geq 0$ and that f is increasing. The value of f in 0 is $f(0) = \alpha^\nu - (1-2^{1-\nu})\alpha - \alpha = \alpha^\nu(1-(2\alpha)^{1-\nu}) \geq 0$ which implies that for all x, $f(x) \geq 0$.

Recall that $w_i(t) = (k+1)q + b$ and $\alpha = b/(k+1)$. Using the notation f and the fact that $(k+1)^{1-\nu} \leq 2^{1-\nu}$, the decrease of potential δ_i^k can be written

$$\begin{aligned}\delta_i^k &= (1-(k+1)^{1-\nu}) \cdot (w_i(t)^\nu - w_i(t)) + (k+1) \cdot f(q) \\ &\geq (1-(k+1)^{1-\nu}) \cdot (w_i(t)^\nu - w_i(t)). \end{aligned} \tag{9}$$

Let $q_k(r)$ be the probability for a processor to receive k work requests when r processors are stealing. $q_k(r)$ is equal to:

$$q_k(r) = \binom{r}{k} \frac{1}{(m-1)^k} \left(\frac{m-2}{m-1}\right)^{r-k}$$

The expected decrease of the potential caused by the steals on processor i is equal to $\sum_{k=0}^{r} \delta_i^k q_k(r)$. Using equation (9), we can bound the expected potential at time $t+1$ by

$$\mathbb{E}\left[\Phi_t - \Phi_{t+1} \mid \mathscr{F}_t\right] = \sum_{i=0}^{m} \sum_{k=0}^{r} \delta_i^k \cdot q_k(r)$$

$$\mathbb{E}\left[\Phi_{t+1} \mid \mathscr{F}_t\right] \leq \left(1 - \sum_{k=0}^{r} (1-(k+1)^{1-\nu}) \cdot q_k(r)\right) \cdot \Phi_t$$

Theorem 4 *The makespan $C_{\max}^{\text{coop}}$ of $W = n$ unit independent tasks scheduled with cooperative work stealing satisfies:*

(i) $\mathbb{E}\left[C_{\max}^{\text{coop}}\right] \leq \frac{W}{m} + 2.88 \cdot \log_2 W + 3.4$

(ii) $\mathbb{P}\left\{C_{\max}^{\text{coop}} \geq \frac{W}{m} + 2.88 \cdot \log_2 W + 2 + \log_2\left(\frac{1}{\epsilon}\right)\right\} \leq \epsilon.$

Proof The proof is very similar to the one of Theorem 2. Let

$$h(r) \stackrel{\text{def}}{=} 1 - \sum_{k=0}^{r} (1-(k+1)^{1-\nu}) \cdot q_k(r)$$

and

$$\lambda^{\text{coop}}(\nu) \stackrel{\text{def}}{=} \max_{1 \leq r \leq m} \frac{r}{-m \cdot \log_2 h(r)}.$$

Using Theorem 1, we have:

$$\mathbb{E}\left[C_{\max}^{\text{coop}}\right] \leq \frac{W}{m} + \nu\lambda^{\text{coop}}(\nu) \cdot \log_2 W + \frac{\lambda(\nu)}{\ln 2} + 1.$$

In the general case the exact computation of $h(r)$ is intractable. However, by a numerical computation, one can show that $3\lambda^{\text{coop}}(3) < 2.88$.

When $\Phi_t < 1$, we have $\sum_i w_i(t)^\nu - w_i(t) < 1$. This implies that for all processor i, $w_i(t)$ equals 0 or 1. This adds (at most) one step of computation at the end of the schedule. As $\lambda(3)/\ln(2) + 1 + 1 = 3.4$, we obtain the calimed bound. □

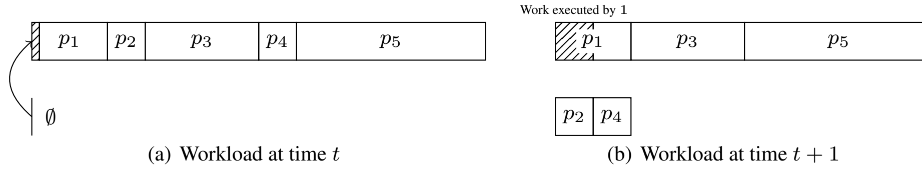
Compared to the situation with no cooperation among thieves, the number of steal requests is reduced by a factor $3.24/2.88 \approx 12\%$. We will see in Section 8 that this is close to the value obtained by simulation.

Remark 1 The exact computation can be accomplished for $\nu = 2$ (Tchiboukdjian et al, 2010) and leads to a constant factor of $2\lambda^{\text{coop}}(2) \leq -2/\log_2(1-\frac{1}{e}) < 3.02$.

6 Weighted independent tasks

In this section, we analyze the number of work requests for weighted independent tasks. Each task j has a processing time p_j which is unknown. When an idle processor attempts to steal a processor, half of the tasks of the victim are transfered from the active processor to the idle one. A task that is currently executed by a processor cannot be stolen. If the victim has $2k(+1)$ tasks (plus one for the task that is currently executed), the work is split in $k(+1)$, k. If the victim has $2k+1(+1)$ tasks, the work is split in $k(+1)$, $k+1$.

In all this analysis, we consider that the scheduler does not know the weight of the different tasks p_j. Therefore, when the work is split in two parts, we do not assume that the work is split fairly (see for example Figure 3) but only that the number of tasks is split in two equal parts.



(a) Workload at time t (b) Workload at time $t+1$

Fig. 3 Evolution of the repartition of tasks during one time step. At time t, one processor has all the tasks. p_1 can not be stolen since the processor 1 has already started executing it. After one work request done by the second processor, one processor has 3 tasks and one has 2 tasks but the workload may be very different, depending on the processing times p_j.

6.1 Definition of the potential function and expected decrease

As the processing times are unknown, the work cannot be shared evenly between both processors and can be as bad as one processor getting all the smallest tasks and one all the biggest tasks (see Figure 3). Let us call $w_i(t)$ the *number of tasks* possessed by the processor i. The potential of the system at time t is defined as:

$$\Phi_t \stackrel{\text{def}}{=} \sum_i \left(w_i(t)^\nu - w_i(t)\right). \tag{10}$$

During a work request, half of the tasks are transfered from an active processor to the idle processor. If the processor j is stealing tasks from processor i, the number of tasks possessed by i and j at time $t+1$ are $w_j(t+1) = \lceil w_i(t)/2 \rceil$ and $w_i(t+1) = \lfloor w_i(t)/2 \rfloor$. Therefore, the decrease of potential is equal to the one of the cooperative steal of Equation 9 for $k=1$:

$$\delta_i \geq (1 - 2^{1-\nu}) \cdot \left(w_i(t)^\nu - w_i(t)\right).$$

Following the analysis of Section 5.2, this shows that in average:

$$\mathbb{E}\left[\Phi_{t+1}\right] \leq (1 - (1 - 2^{1-\nu})q(r)) \cdot \Phi_t. \tag{11}$$

6.2 Bound on the makespan

Equation 11 allows us to apply Theorem 1 to derive a bound on the makespan of weighted tasks by the distributed list scheduling algorithm. This bound differs from the one for unit tasks only by an additive term of $p_{\max}$.

Theorem 5 *Let* $p_{\max} \stackrel{\text{def}}{=} \max p_j$ *be the maximum processing times. The expected makespan to schedule* n *weighted tasks of total processing time* $W = \sum p_j$ *by DLS is bounded by*

$$\mathbb{E}\left[C_{\max}\right] \leq \frac{W}{m} + \frac{m-1}{m} p_{\max} + 3.24 \cdot \left(\log_2 n + \frac{1}{2 \ln 2}\right) + 1$$

Proof Let Φ_t be the potential defined by Equation 10. At time $t = 0$, the potential of the system is bounded by $W^\nu - W$. Therefore, by Theorem 1, the number of work requests before $\Phi_t < 1$ is bounded by

$$m \cdot \lambda \cdot \left(\log_2 \Phi_0 + 1 + \frac{1}{\ln 2}\right) \leq m \cdot \nu\lambda(\nu) \cdot \left(2 \log_2 W + 1 + \frac{1}{\ln 2}\right),$$

where $\nu\lambda(\nu) < 3.24$ is the same constant as the bound for the unit tasks with the potential function $\sum_i w_i^\nu$ of Theorem 3.

As $\Phi_t \in \mathbb{N}$, $\Phi_t < 1$ implies that $\Phi_t = 0$. Moreover, by definition of Φ_t, this implies that for all i: $w_i(t)^\nu - w_i(t) = 0$, which implies that for all i: $w_i(t) \leq 1$. Therefore, once Φ_t is equal to 0, there is at most one task per processor. This phase can last for at most $p_{\max}$ unit of time, generating at most $(m-1)p_{\max}$ work requests. □

Remark 2 The same analysis applies for the cooperative stealing scheme of Section 5.2 leading to the same improved bound in $2.88 \log_2 n$ instead of $3.24 \log_2 n$.

7 Tasks with precedences

In this section, we show how the well known non-blocking work stealing of Arora et al (2001) (denoted ABP in the sequel) can be analyzed with our method which provides tighter bounds for the makespan. We first recall the WS scheduler of ABP, then we show how to define the amount of work on a processor $w_i(t)$, finally we apply the analysis of Section 3 to bound the makespan.

7.1 ABP work-stealing scheduler

Following Arora et al (2001), a multithreaded computation is modeled as a directed acyclic graph G with W unit tasks task and edges define precedence constraints. There is a single source task and the out-degree is at most 2. The critical path of G is denoted by D. ABP schedules the DAG G as follows. Each processor i maintains a double-ended queue (called a deque) Q_i of ready tasks. At each slot, an active processor i with a non-empty deque executes the task at the bottom of its deque Q_i; once its execution is completed, this task is popped from the bottom of the deque, enabling – *i.e.* making ready – 0, 1 or 2 child tasks that are pushed at the bottom of Q_i. At each top, an idle processor j with an empty deque Q_j becomes a thief: it performs a steal request on another randomly chosen victim deque; if the victim deque contains ready tasks, then its top-most task is popped and pushed into

the deque of one of its concurrent thieves. If j becomes active just after its steal request, the steal request is said successful. Otherwise, Q_j remains empty and the steal request fails which may occur in the three following situations: either the victim deque Q_i is empty; or, Q_i contains only one task currently in execution on i; or, due to contention, another thief performs a successful steal request on i simultaneously.

7.2 Definition of $w_i(t)$

Let us first recall the definition of the *enabling tree* of Arora et al (2001). If the execution of task u enables task v, then the edge (u, v) of G is an enabling edge. The sub-graph of G consisting of only enabling edges forms a rooted tree called the enabling tree. We denote by $h(u)$ the height of a task u in the enabling tree. The root of the DAG has height D. Moreover, it has been shown in Arora et al (2001) that tasks in the deque have strictly decreasing height from top to bottom except for the two bottom most tasks which can have equal heights.

We now define $w_i(t)$, the amount of work on processor i at time t. Let h_t be the maximum height of all tasks in the deque. If the deque contains at least two tasks including the one currently executing we define $w_i(t) = (2\sqrt{2})^{h_t}$. If the deque contains only one task currently executing we define $w_i(t) = \frac{1}{2} \cdot (2\sqrt{2})^{h_t}$. The following lemma states that this definition of $w_i(t)$ behaves in a similar way than the one used for the independent unit tasks analysis of Section 4.

Lemma 1 *For any active processor i, we have* $w_i(t+1) \leq w_i(t)$. *Moreover, after any successful steal request from a processor j on i,* $w_i(t+1) \leq w_i(t)/2$ *and* $w_j(t+1) \leq w_i(t)/2$ *and if all steal requests are unsuccessful we have* $w_i(t+1) \leq w_i(t)/\sqrt{2}$.

Proof We first analyze the execution of one task u at the bottom of the deque. Executing task u enables at most two tasks and these tasks are the children of u in the enabling tree. If the deque contains more than one task, the top most task has height h_t and this task is still in the deque at time $t+1$. Thus the maximum height does not change and $w_i(t) = w_i(t+1)$. If the deque contains only one task, we have $w_i(t) = \frac{1}{2} \cdot (2\sqrt{2})^{h_t}$ and $w_i(t+1) \leq (2\sqrt{2})^{h_t - 1}$. Thus $w_i(t+1) \leq w_i(t)$.

We now analyze a successful steal from processor j. In this case, the deque of processor i contains at least two tasks and $w_i(t) = (2\sqrt{2})^{h_t}$. The stolen task is one with the maximum height and is the only task in the deque of processor j thus $w_j(t+1) = \frac{1}{2} \cdot (2\sqrt{2})^{h_t} \leq w_i(t)/2$. For the processor i, either its deque contains only one task after the steal with height at most h_t and $w_i(t+1) \leq \frac{1}{2} \cdot (2\sqrt{2})^{h_t} \leq w_i(t)/2$, either there are still more than 2 tasks and $w_i(t+1) \leq (2\sqrt{2})^{h_t - 1} < w_i(t)/2$.

Finally, if all steal requests are unsuccessful, the deque of processor i contains at most one task. If the deque is empty $w_i(t+1) = w_i(t) = 0$ and thus $w_i(t+1) \leq w_i(t)/\sqrt{2}$. If the deque contains exactly one task, $w_i(t) = \frac{1}{2} \cdot (2\sqrt{2})^{h_t}$ and $w_i(t+1) \leq (2\sqrt{2})^{h_t - 1}$ thus $w_i(t+1) \leq w_i(t)/\sqrt{2}$. □

7.3 Bound on the makespan

To study the number of steals, we follow the analysis presented in Section 3 with the potential function $\Phi(t) = \sum_i w_i(t)^2$. Using results from lemma 1, we compute the decrease of

the potential $\delta_i(t)$ due to steal requests on processor i by distinguishing two cases. If there is a successful steal from processor j,

$$\delta_i(t) = w_i(t)^2 - w_i(t+1)^2 - w_j(t+1)^2 \geq w_i(t)^2 - 2 \cdot \left(\frac{w_i(t)}{2}\right)^2 \geq \frac{1}{2} \cdot w_i(t)^2.$$

If all steals are unsuccessful, the decrease of the potential is

$$\delta_i(t) = w_i(t)^2 - w_i(t+1)^2 \geq w_i(t)^2 - \left(\frac{w_i(t)}{\sqrt{2}}\right)^2 \geq \frac{1}{2} \cdot w_i(t)^2.$$

In all cases, $\delta_i(t) \geq w_i(t)^2/2$. We obtain the expected potential at time $t+1$ by summing the expected decrease on each active processor:

$$\mathbb{E}\left[\Phi_t - \Phi_{t+1} \mid \mathscr{F}_t\right] \geq \sum_{i=0}^{m} \frac{w_i(t)^2}{2} \cdot q(r_t)$$
$$\mathbb{E}\left[\Phi_{t+1} \mid \mathscr{F}_t\right] \leq \left(1 - \frac{q(r_t)}{2}\right) \cdot \Phi(t)$$

Finally, we can state the following theorem.

Theorem 6 *On a DAG composed of W unit tasks, with critical path D, one source and out-degree at most 2, the makespan of ABP work stealing verifies:*

(i) $$\mathbb{E}\left[C_{\max}\right] \leq \frac{W}{m} + \frac{3}{1 - \log_2(1 + \frac{1}{e})} \cdot D + 1 < \frac{W}{m} + 5.5 \cdot D + 1.$$

(ii) $$\mathbb{P}\left\{C_{\max} \geq \frac{W}{m} + \frac{3}{1 - \log_2(1 + \frac{1}{e})} \cdot \left(D + \log_2 \frac{1}{\epsilon}\right) + 1\right\} \leq \epsilon$$

Proof The proof is a direct application of Theorem 1. As in the initial step there is only one non empty deque containing the root task with height D, the initial potential is

$$\Phi(0) = \left(\frac{1}{2} \cdot \left(2\sqrt{2}\right)^D\right)^2.$$

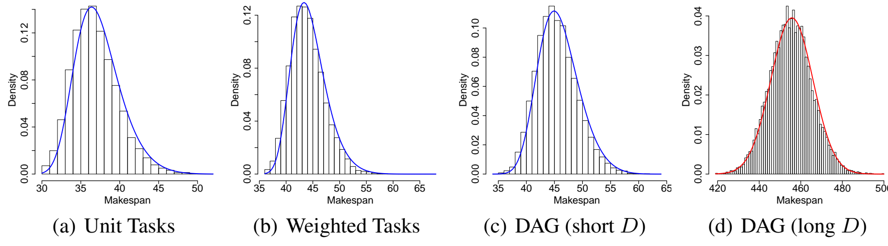
Thus the expected number of steal requests before $\Phi(t) < 1$ is bounded by

$$\begin{aligned}
\mathbb{E}\left[R\right] &\leq \lambda \cdot m \cdot \log_2\left[\left(\frac{1}{2} \cdot \left(2\sqrt{2}\right)^D\right)^2\right] + m \cdot \left(1 + \frac{\lambda}{\ln(2)}\right) \\
&\leq 2\lambda \cdot m \cdot D \cdot \log_2(2\sqrt{2}) + m \cdot \left(1 + \frac{\lambda}{\ln(2)} - 2\lambda\right) \\
&\leq 3\lambda \cdot m \cdot D \qquad\qquad (\text{as } 1 + \lambda/\ln(2) - 2\lambda < 0)
\end{aligned}$$

where $\lambda = (1 - \log_2(1 + 1/e))^{-1}$ is the same constant as the bound for the unit tasks of Section 4.

Moreover, when $\Phi(t) < 1$, we have $\forall i, w_i(t) < 1$. There is at most one task of height 0 in each deque, *i.e.* a leaf of the enabling tree which cannot enable any other task. This last step generates at most $m-1$ additional steal requests. In total, the expected number of steal requests is bounded by $\mathbb{E}\left[R\right] \leq 3\lambda \cdot m \cdot D + m - 1$. The bound on the makespan is obtained using the relation $m \cdot C_{\max} = W + R$.

The proof of (i) applies *mutatis mutandis* to prove the bound in probability (ii). □



(a) Unit Tasks (b) Weighted Tasks (c) DAG (short D) (d) DAG (long D)

Fig. 4 Distribution of the makespan for unit independent tasks 4(a), weighted independent tasks 4(b) and tasks with dependencies 4(c) and 4(d). The first three models follow a gev distribution (blue curves), the last one is gaussian (red curve).

Remark. In Arora et al (2001), the authors established the upper bounds :

$$\mathbb{E}\left[C_{\max}\right] \leq \frac{W}{m} + 32 \cdot D \quad \text{and} \quad \mathbb{P}\left\{C_{\max} \geq \frac{W}{m} + 64 \cdot D + 16 \cdot \log_2 \frac{1}{\epsilon}\right\} \leq \epsilon$$

in Section 4.3, proof of Theorem 9. Our bounds greatly improve the constant factors of this previous result.

8 Experimental study

The theoretical analysis gives an upper bounds on the expected value of the makespan and deviation from the mean for the various models we considered. In this section, we study experimentally the distribution of the makespan. Statistical tests give evidence that the makespan for independent tasks follows a generalized extreme value (gev) distribution (Kotz and Nadarajah, 2001). This was expected since such a distribution arises when dealing with maximum of random variables. For tasks with dependencies, it depends on the structure of the graph: DAGs with short critical path still follow a gev distribution but when the critical path grows, it tends to a gaussian distribution. We also study in more details the overhead to W/m and show that it is approximately $2.37 \log_2 W$ for unit independent tasks which is close to the theoretical result of $3.24 \log_2 W$ (*cf.* Section 5).

We developed a simulator that strictly follows our model. At the beginning, all the tasks are given to processor 0 in order to be in the worst case, *i.e.* when the initial potential Φ_0 is maximum. Each pair (m,W) is simulated 10000 to get accurate results, with a coefficient of variation about 2%.

8.1 Distribution of the makespan

We consider here a fixed workload $W = 2^{17}$ on $m = 2^{10}$ processors for independent tasks and $m = 2^7$ processors for tasks with dependencies. For the weighted model, processing times were generated randomly and uniformly between 1 and 10. For the DAG model, graphs have been generated using a layer by layer method. We generated two types of DAGs, one with a short critical path (close to the minimum possible $\log_2 W$) and the other one with a long critical path (around $W/4m$ in order to keep enough tasks per processor per layer). Fig. 4 presents histograms for $C_{\max} - \lceil W/m \rceil$.

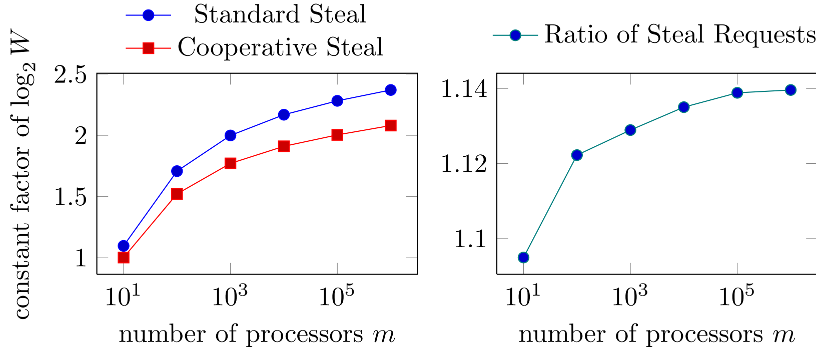


Fig. 5 (Left) Constant factor of $\log_2 W$ against the number of processors for the standard steal and the cooperative steal. (Right) Ratio of steal requests (standard/cooperative).

The distributions of the first three models (a,b,c in Fig. 4) are clearly not gaussian: they are asymmetrical with an heavier right tail. To fit these three models, we use the generalized extreme value (gev) distribution (Kotz and Nadarajah, 2001). In the same way as the normal distribution arises when studying the sum of independent and identically distributed (iid) random variables, the gev distribution arises when studying the maximum of iid random variables. The extreme value theorem, an equivalent of the central limit theorem for maxima, states that the maximum of iid random variables converges in distribution to a gev distribution. In our setting, the random variables measuring the load of each processor are not independent, thus the extreme value theorem cannot apply directly. However, it is possible to fit the distribution of the makespan to a gev distribution. In Fig. 4, the fitted distributions (blue curve) closely follow the histograms. To confirm this graphical approach, we performed a goodness of fit test. The χ^2 test is well-suited to our data because the distribution of the makespan is discrete. We compared the results of the best fitted gev to the best fitted gaussian. The χ^2 test strongly rejects the gaussian hypothesis but does not reject the gev hypothesis with a p-value of more than 0.5. This confirms that the makespan follows a gev distribution. We fitted the last model, DAG with long critical path, with a gaussian (red curve in Fig. 4(d)). In this last case, the completion time of each layer of the DAG should correspond to a gev distribution but the total makespan, the sums of all layers, should tend to a gaussian by the central limit theorem. Indeed the χ^2 test does not reject the gaussian hypothesis with a p-value around 0.3.

8.2 Study of the $\log_2 W$ term

We focus now on unit independent tasks as the other models rely on too many parameters (the choice of the processing times for weighted tasks and the structure of the DAG for tasks with dependencies). We want to show that the number of work requests is proportional to $\log_2 W$ and study the proportionality constant. We first launch simulations with a fixed number of processors m and a wide range of work in successive powers of 10. A linear regression confirms the linear dependency in $\log_2 W$ with a coefficient of determination ("r squared") greater than 0.9999[1].

[1] the closer to 1, the better

Then, we obtain the slope of the regression for various number of processors. The value of the slope tends to a limit around 2.37 (*cf.* Fig. 5(left)). This shows that the theoretical analysis of Theorem 2 is almost accurate with a constant of approximately 3.24. We also study the constant factor of $\log_2 W$ for the cooperative steal of Section 5. The theoretical value of 2.88 is again close to the value obtained by simulation 2.08 (*cf.* Figure 5(left)). The difference between the theoretical and the practical values can be explained by the worst case analysis on the number of steal requests per time step in Theorem 1.

Moreover, simulations in Fig. 5(right) show that the ratio of steal requests between standard and cooperative steals goes asymptotically to 14%. The ratio between the two corresponding theoretical bounds is about 12%. This indicates that the biais introduced by our analysis is systematic and thus, our analysis may be used as a good prediction while using cooperation among thieves.

9 Concluding Remarks

In this paper, we presented a complete analysis of the cost of distribution in list scheduling. We proposed a new framework, based on potential functions, for analyzing the complexity of distributed list scheduling algorithms. In all variants of the problem, we succeeded to characterize precisely the overhead due to the decentralization of the list. These results are summarized in the following table comparing makespans for standard (centralized) and decentralized list scheduling.

	Centralized	Decentralized (WS)
Unit Tasks $(W = n)$	$\left\lceil \frac{W}{m} \right\rceil$	$\frac{W}{m} + 3.24 \log_2 W + 3.33$
Initial repartition	–	$\frac{W}{m} + 1.83 \log_2 \sum_{i=0}^{m} \left(w_i - \frac{W}{m} \right)^2 + 3.63$
Cooperative	–	$\frac{W}{m} + 2.88 \log_2 W + 3.4$
Weighted Tasks	$\frac{W}{m} + \frac{m-1}{m} \cdot p_{\max}$	$\frac{W}{m} + \frac{m-1}{m} \cdot p_{\max} + 3.24 \log_2 n + 3.33$
Tasks w. predecences	$\frac{W}{m} + \frac{m-1}{m} \cdot D$	$\frac{W}{m} + 5.5D + 1$

In particular, in the case of independent tasks, the overhead due to the distribution is small and only depends on the number of tasks and not on their weights. In addition, this analysis improves the bounds for the classical work stealing algorithm of Arora et al (2001) from $32D$ to $5.5D$. We believe that this work should help to clarify the links between classical list scheduling and work stealing.

Furthermore, the framework to analyze DLS algorithms described in this paper is more general than the method of Arora et al (2001). Indeed, we do not assume a specific rule (*e.g.* depth first execution of tasks) to manage the local lists. Moreover, we do not refer to the structure of the DAG (*e.g.* the depth of a task in the enabling tree) but on the work contained in each list. Thus, we plan to extend this analysis to the case of general precedence graphs.

Acknowledgements The authors would like to thank Julien Bernard and Jean-Louis Roch for fruitful discussions on the preliminary version of this work.

References

Adler M, Chakrabarti S, Mitzenmacher M, Rasmussen L (1995) Parallel randomized load balancing. In: Proceedings of STOC, pp 238–247

Arora NS, Blumofe RD, Plaxton CG (2001) Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems 34(2):115–144

Azar Y, Broder AZ, Karlin AR, Upfal E (1999) Balanced allocations. SIAM Journal on Computing 29(1):180–200, DOI 10.1137/S0097539795288490

Bender MA, Rabin MO (2002) Online scheduling of parallel programs on heterogeneous systems with applications to cilk. Theory of Computing Systems 35:2002

Berenbrink P, Friedetzky T, Goldberg LA (2003) The natural work-stealing algorithm is stable. SIAM Journal of Computing 32(5):1260–1279

Berenbrink P, Friedetzky T, Goldberg LA, Goldberg PW, Hu Z, Martin R (2007) Distributed selfish load balancing. SIAM Journal on Computing 37(4), DOI 10.1137/060660345

Berenbrink P, Friedetzky T, Hu Z, Martin R (2008) On weighted balls-into-bins games. Theoretical Computer Science 409(3):511 – 520

Berenbrink P, Friedetzky T, Hu Z (2009) A new analytical method for parallel, diffusion-type load balancing. Journal of Parallel and Distributed Computing 69(1):54 – 61

Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5):720–748

Chekuri C, Bender M (2001) An efficient approximation algorithm for minimizing makespan on uniformly related machines. Journal of Algorithms 41(2):212 – 224

Drozdowski M (2009) Scheduling for Parallel Processing. Springer

Frigo M, Leiserson CE, Randall KH (1998) The implementation of the Cilk-5 multithreaded language. In: Proceedings of PLDI

Gast N, Gaujal B (2010) A Mean Field Model of Work Stealing in Large-Scale Systems. In: Proceedings of SIGMETRICS

Gautier T, Besseron X, Pigeon L (2007) KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Proceedings of PASCO, pp 15–23

Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics 17:416–429

Hwang JJ, Chow YC, Anger FD, Lee CY (1989) Scheduling precedence graphs in systems with interprocessor communication times. SIAM Journal on Computing 18(2):244–257

Kotz S, Nadarajah S (2001) Extreme Value Distributions: Theory and Applications. World Scientific Publishing Company

Leung J (2004) Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press

Mitzenmacher M (1998) Analyses of load stealing models based on differential equations. In: Proceedings of SPAA, pp 212–221

Robert Y, Vivien F (2009) Introduction to Scheduling. Chapman & Hall/CRC Press

Robison A, Voss M, Kukanov A (2008) Optimization via reflection on work stealing in TBB. In: Proceedings of IPDPS, pp 1–8

Schwiegelshohn U, Tchernykh A, Yahyapour R (2008) Online scheduling in grids. In: Proceedings of IPDPS

Tchiboukdjian M, Gast N, Trystram D, Roch JL, Bernard J (2010) A tighter analysis of work stealing. In: The 21st International Symposium on Algorithms and Computation (ISAAC)

Traoré D, Roch JL, Maillard N, Gautier T, Bernard J (2008) Deque-free work-optimal parallel STL algorithms. In: Proceedings of Euro-Par, pp 887–897

Conclusion

Pour exploiter efficacement la puissance des nouvelles architectures de calcul, nous proposons dans cette thèse de combiner les approches *processor-oblivious* et *cache-oblivious*. Ces techniques permettent de relever les deux défis du parallélisme et de la hiérarchie mémoire dans le cas des machines à mémoire partagée. De plus les applications utilisant ces techniques ont un code plus simple, sont portables sur différentes architectures et peuvent s'adapter à la variation des paramètres de l'architecture.

Les techniques pour concevoir des algorithmes *processor-oblivious* et *cache-oblivious* sont connues et ont été présentées dans les chapitres 2 et 3. Elles sont en général compatibles et peuvent donc être combinées. Certaines techniques sont même communes comme par exemple l'utilisation d'algorithmes diviser pour régner. La principale difficulté pour obtenir un algorithme à la fois *processor-oblivious* et *cache-oblivious* n'est donc pas sa conception mais celle du moteur exécutif chargé de l'ordonnancement de l'algorithme sur l'architecture cible. En effet, les algorithmes *processor-oblivious* sont ordonnancés par vol de travail, une technique décentralisée qui permet de limiter le surcoût du calcul et de l'exécution de l'ordonnancement. Afin d'utiliser efficacement les caches, il faut modifier la politique d'ordonnancement pour prendre en compte la localité des accès mémoire lors de l'exécution parallèle. Cependant, cette politique doit rester décentralisée pour conserver un faible surcoût à l'exécution.

Nous avons vu dans le chapitre 4 que la nature des caches, privés ou partagés, a une influence fondamentale sur la politique d'ordonnancement à adopter pour minimiser le nombre de défauts de cache. Dans le cas d'une architecture à caches privés, un ordonnancement efficace en cache répartit sur les différents cœurs de grands sous-ensembles de données distants en mémoire. Cette propriété est vérifiée par la politique de vol standard (c'est-à-dire vol de la tâche en haut de la pile), le moteur de vol peut donc être utilisé sans modification. Il faut cependant prêter attention aux défauts de cache de cohérence lors de la conception de l'algorithme parallèle.

Le cas des architectures à caches partagés est plus difficile. Il existe une politique d'ordonnancement qui minimise le nombre de défauts de cache (la politique PDF [BG04]) mais cette politique est basée sur une liste centralisée ce qui limite le passage à l'échelle à cause de la contention lorsqu'elle est accédée en concurrence. Il semble difficile d'obtenir un ordonnancement décentralisé efficace pour les caches partagés dans le cas général. En effet, un ordonnancement efficace sur un cache partagé répartit sur les différents cœurs de petits sous-ensembles de données proches en mémoire. C'est la tendance inverse d'une politique qui minimise le nombre de vols pour limiter le surcoût de l'ordonnancement.

Dans cette thèse, nous proposons un ordonnancement décentralisé par vol de travail efficace sur les architectures à caches partagés dans le cas particulier des boucles parallèles. Cet ordonnancement "à fenêtre glissante" (*cf.* chapitre 7) est un compromis entre l'ordonnancement par vol de travail standard qui minimise le nombre de vols et l'ordonnancement centralisé de la politique PDF qui minimise le nombre de défauts de cache. Dans cet ordonnancement, les cœurs sont contraints à travailler à l'intérieur

d'une fenêtre glissant sur les données. La taille de cette fenêtre est choisie de telle sorte que le sous-ensemble de données correspondant rentre dans le cache partagé. Grâce à cette fenêtre, les cœurs coopèrent pour l'utilisation du cache et profitent des défauts de cache des autres cœurs.

Cependant le moteur exécutif doit connaître la taille du cache pour calculer la taille de la fenêtre. Il n'est donc pas *cache-oblivious* même si l'algorithme l'est. Il parait difficile de se passer de la connaissance de la taille du cache même dans ce cas simple du parallélisme de tâches indépendantes. Il ne nous semble pas possible de trouver un ordonnanceur *cache-oblivious* dans le cas où l'architecture contient à la fois des caches privés et des caches partagés. Néanmoins, cette limitation n'affecte pas la portabilité ni la simplicité du code de l'application, seul le moteur d'exécution est dépendant de l'architecture.

Pour étendre le moteur d'exécution au cas général des tâches avec dépendances, il faut disposer d'un modèle simple pour exprimer les contraintes de précédence et les accès mémoire et trouver une politique de vol garantissant un bon compromis entre nombre de vols et défauts de cache. Nous pensons que trois points développés dans cette thèse peuvent aider à atteindre ce but ambitieux.

1. Une bonne politique de vol doit sélectionner une tâche dont les données sont proches des données utilisées par la tâche courante à la manière de l'ordonnancement à fenêtre glissante.
2. De plus, la nouvelle preuve du vol de travail du chapitre 8 permet de borner le nombre de vols de politiques différentes de la politique standard. Elle permet également d'analyser des politiques plus flexibles qui ne font pas référence à la profondeur dans le DAG mais seulement au travail et à sa répartition au moment du vol.
3. Enfin, nous avons montré dans le chapitre 7 que les distances de réutilisation sont un modèle utile pour analyser le nombre de défauts de cache d'une exécution parallèle. Une voie pour concevoir un modèle mêlant parallélisme et accès mémoire serait d'enrichir le DAG avec des informations permettant de calculer les distances de réutilisation de l'ordonnancement généré. On pourrait utiliser par exemple un graphe composé d'un nœud par donnée et relier par une arête chaque tâche avec les données qu'elle utilise lors de son exécution.

Le deuxième point abordé dans cette thèse est l'application des techniques *processor-oblivious* et *cache-oblivious* au domaine de la visualisation scientifique. Les filtres de visualisation scientifique étant gourmands en accès mémoire, les approches réduisant le nombre de défauts de cache ont un fort impact sur leur temps d'exécution [SCEL02]. Nos travaux ont permis de confirmer l'intérêt des techniques *cache-oblivious* pour la visualisation. L'organisation de maillage non structuré FastCOL que nous avons proposé dans le chapitre 5 permet d'accélérer de nombreux filtres de la bibliothèque VTK sur CPU. Elle offre également des gains de performances significatifs sur GPU. De plus, l'organisation FastCOL est plus rapide à calculer que l'organisation OpenCCL [YLPM05] et dispose d'une garantie de performance théorique. Par ailleurs, nous avons montré que les structures accélératrices basées sur les arbres telles que l'arbre min-max peuvent être adaptées à notre organisation mémoire, ce qui apporte un gain supplémentaire sur la localité.

Nous avons ensuite étudié dans le chapitre 6 l'impact de l'ordonnancement sur le nombre de défauts de cache dans le cas des algorithmes d'extraction d'isosurface. Une parallélisation standard génère beaucoup plus de défauts de cache que l'exécution séquentielle sur une architecture à caches partagés ce qui limite l'accélération obtenue. Nous avons montré que les ordonnancements "à fenêtre" (statique ou glissante) permettent de garantir un nombre de défauts de cache équivalent à l'exécution séquentielle et conduisent à un gain de performance significatif par rapport à l'ordonnancement standard sans fenêtre.

Le chapitre 6 se limite à l'étude des algorithmes d'extraction d'isosurface mais les boucles parallèles sont très fréquentes dans les filtres de visualisation scientifique et l'ordonnancement à fenêtre glissante pourrait être utilisé avec profit pour d'autres algorithmes. Néanmoins, certains filtres ne sont pas adaptés à un parallélisme de boucle indépendante. Un projet en collaboration entre l'équipe MOAIS et l'équipe de visualisation scientifique d'EDF vise à identifier d'autres schémas parallèles dans la bibliothèque VTK et à en proposer une implémentation.

Enfin, une autre source de parallélisme présente dans les applications de visualisation scientifique est le parallélisme de type pipeline lors de l'enchainement de plusieurs filtres. Silva *et al.* proposent une implémentation parallèle du pipeline de VTK dans laquelle chaque filtre est exécuté en séquentiel [VOS+10]. Pour profiter de tout le parallélisme disponible, il serait intéressant de combiner ces deux approches : parallélisme au niveau du filtre et au niveau du pipeline. Afin de réaliser ce couplage, il faut choisir le nombre de processeurs à allouer à chaque filtre en prenant en compte le surcoût du parallélisme, la hiérarchie mémoire, la taille des tampons reliant les filtres et la réactivité lors d'une action de l'utilisateur.

Bibliographie

[ABB02] Umut A. Acar, Guy E. Blelloch et Robert D. Blumofe : The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002. 64, 65, 66, 73

[ABF05] Lars Arge, Gerth S. Brodal et Rolf Fagerberg : Cache-oblivious data structures. *In Handbook of Data Structures and Applications*. Chapman& Hall/CRC, 2005. 27, 36

[ABP98] Nimar S. Arora, Robert D. Blumofe et C. Greg Plaxton : Thread scheduling for multiprogrammed multiprocessors. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998. 4, 10, 47, 49, 52, 66, 130, 131

[AGNS08] Lars Arge, Michael T. Goodrich, Michael J. Nelson et Nodari Sitchinava : Fundamental parallel algorithms for private-cache chip multiprocessors. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 197–206, 2008. 68

[AGS10] Lars Arge, Michael T. Goodrich et Nodari Sitchinava : Parallel external memory graph algorithms. *In International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–11, 2010. 68

[ALS10] K. Agrawal, Charles E. Leiserson et J. Sukha : Executing task graphs using work-stealing. *In International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010. 54

[ASZ10] Deepak Ajwani, Nodari Sitchinava et Norbert Zeh : Geometric algorithms for private-cache chip multiprocessors - (extended abstract). *In European Symposium on Algorithms (ESA)*, pages 75–86, 2010. 68

[AV88] A. Aggarwal et J. S. Vitter : The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 5, 26, 36

[BBF+03] Michael A. Bender, Gerth S. Brodal, Rolf Fagerberg, Ge, He, Haodong Hu, Iacono et Lopez-Ortiz : The cost of cache-oblivious searching. *In Symposium on Foundations of Computer Science (FOCS)*, 2003. 36

[BCG+08] Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen et Michael Kozuch : Provably good multicore cache performance for divide-and-conquer algorithms. *In Symposium on Discrete Algorithms (SODA)*, pages 501–510, 2008. 71, 72

[BD01] Kristof Beyls et Erik H. D'holl : Reuse distance as a metric for cache behavior. *In International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 617–662, 2001. 20

[BDFC02] Michael A. Bender, Erik Demaine et Martin Farach-Colton : Efficient tree layout in a multilevel memory hierarchy. *In European Symposium on Algorithms (ESA)*, 2002. 36

[BDFC05] Michael A. Bender, Erik Demaine et Martin Farach-Colton : Cache-oblivious B-trees. *SIAM Journal on Computing*, 35, 2005. 32

[BDG+00] S. Browne, Jack Dongarra, N. Garner, G. Ho et P. Mucci : A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000. 18

[BDW02] Michael A. Bender, Ziyang Duan et Jing Wu : A locality-preserving cache-oblivious dynamic dictionary. *In Symposium on Discrete Algorithms (SODA)*, pages 29–38, 2002. 36

[Bes10] Xavier Besseron : *Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle.* Thèse de doctorat, Université de Grenoble, 2010. 58

[BF03] Gerth S. Brodal et Rolf Fagerberg : On the limits of cache-obliviousness. *In Symposium on Theory of Computing (STOC)*, 2003. 37

[BFH07] Michael Bader, Robert Franz et Stephan Güntherand Alexander Heinecke : Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves. *In International Conference on Parallel Processing and Applied Mathematics (PPAM)*, volume 4967, pages 628–638, 2007. 36

[BFJ+96a] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson et Keith H. Randall : An analysis of DAG-consistent distributed-shared memory algorithms. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, 1996. 64, 65

[BFJ+96b] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson et Keith H. Randall : DAG-consistent distributed shared memory. *In International Parallel Processing Symposium (IPPS)*, pages 132–141, 1996. 64, 65

[BFV04] Gerth S. Brodal, Rolf Fagerberg et Vinther : Engineering a cache-oblivious sorting algorithm. *In International Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2004. 37

[BG04] Guy E. Blelloch et Phillip B. Gibbons : Effectively sharing a cache among threads. *In* Phillip B. Gibbons et Micah Adler, éditeurs : *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 235–244, 2004. 63, 69, 70, 155

[BGM99] Guy E. Blelloch, Phillip B. Gibbons et Yossi Matias : Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999. 69, 70

[BGS09] Guy E. Blelloch, Phillip B. Gibbons et Harsha Vardhan Simhadri : Brief announcement : low depth cache-oblivious sorting. *In Symposium*

on Parallel Algorithms and Architectures (SPAA), pages 121–123, 2009. 71

[BH07] Michael A. Bender et Haodong Hu : An adaptive packed-memory array. *Transactions on Database Systems*, 32(4), 2007. 37

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, B. C. Kuszmaul, Charles E. Leiserson, Keith H. Randall et Y. Zhou : Cilk : an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37:55–69, 1996. 4

[BL99] Robert D. Blumofe et Charles E. Leiserson : Scheduling multithreaded computations by workstealing. *Journal of the ACM*, 46, 1999. 3, 4, 47

[Ble96] Guy E. Blelloch : Programming parallel algorithms. *Communications of the ACM*, 39, 1996. 44

[BLTG09] Xavier Besseron, Christophe Laferriere, Daouda Traoré et Thierry Gautier : X-Kaapi : Une nouvelle implémentation eXtrême du vol de travail. *In Rencontres Francophones du Parallélisme (RenPar'19)*, 2009. 58, 114

[BM72] Rudolf Bayer et Edward M. McCreight : Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. 31

[BM96] Guy E. Blelloch et Maggs : Parallel algorithms. *Computing Surveys*, 28, 1996. 40

[BRT08] Julien Bernard, Jean-Louis Roch et Daouda Traoré : Processor-oblivious parallel stream computations. *In International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, pages 72–76, 2008. 56

[BZ05] Michael Bader et Christoph Zenger : A cache oblivious algorithm for matrix multiplication based on peano's space filling curve. *In International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 1042–1049, 2005. 34, 36

[CGK+07] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry et Chris Wilkerson : Scheduling threads for constructive cache sharing on CMPs. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 105–115, 2007. 69, 70

[CGKS05] Dhruba Chandra, Fei Guo, Seongbeom Kim et Yan Solihin : Predicting inter-thread cache contention on a chip multi-processor architecture. *In High-Performance Computer Architecture (HPCA)*, pages 340–351, 2005. 62

[CJR+99] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R.Lebeck, Shyam Mundhra et Mithuna Thottethodi : Nonlinear array layouts for hierarchical memory systems. *In International Conference on Supercomputing (ISC)*, pages 444–453, 1999. 36

[CLPT02] Siddhartha Chatterjee, Lebeck, Patnala et Mithuna Thottethodi : Recursive array layouts and fast matrix multiplication. *Transactions on Parallel and Distributed Systems*, 13, 2002. 35, 36

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Introduction to Algorithms, Third Edition*. The MIT Press, 3 édition, September 2009. 31, 44, 46

[Col88] R. Cole : Parallel merge sort. *SIAM Journal on Computing*, 1(4):770–785, 1988. 46

[CQ09] Michael Jason Cade et Apan Qasem : Balancing locality and parallelism on shared-cache multi-core systems. *In International Conference on High Performance Computing and Communications (HPCC)*, pages 188–195, 2009. 70

[CS06] Jichuan Chang et Gurindar S. Sohi : Cooperative caching for chip multiprocessors. *In International Symposium on Computer Architecture (ISCA)*, pages 264–276, 2006. 62

[CSBR10] Rezaul Alam Chowdhury, Francesco Silvestri, B. Blakeley et Vijaya Ramachandran : Oblivious algorithms for multicores and network of processors. *In International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010. 71, 72

[CUD10] *NVIDIA CUDA C Programming Guide*, 2010. 42

[DGG+07] Vincent Danjean, Roland Gillard, Serge Guelton, Jean-Louis Roch et Thomas Roche : Adaptive loops with kaapi on multicore and grid : applications in symmetric cryptography. *In Parallel Symbolic Computation (PASCO)*, pages 33–42, 2007. 53

[DGK+05] El Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire et Jean-Louis Roch : Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques*, 24(5):505–524, 2005. 56

[DLP03] Jack Dongarra, Piotr Luszczek et Antoine Petitet : The LINPACK benchmark : past, present and future. *Concurrency and Computation : Practice and Experience*, 15(9):803–820, 2003. 1

[DP09] Devdatt Dubhashi et Alessandro Panconesi : *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. 50

[Dre07] Ulrich Drepper : What every programmer should know about memory, 2007. 7, 15

[FGBS10] Jeremy T. Fineman, Phillip B. Gibbons, Guy E. Blelloch et Harsha Vardhan Simhadri : Efficient scheduling for parallel memory hierarchies, 2010. 71, 72

[Fic83] Faith E. Fich : New bounds for parallel prefix circuits. *In Symposium on Theory of Computing (STOC)*, pages 100–109, 1983. 55

[FLH98] Matteo Frigo, Charles E. Leiserson et Keith H.Randall : The implementation of the cilk-5 multithreaded language. *In Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998. 43, 52, 53, 129

[FLPR99] Matteo Frigo, Charles E. Leiserson, H. Prokop et S. Ramachandran : Cache-Oblivious Algorithms. *In Symposium on Foundations of Computer Science (FOCS)*, page 285, 1999. 6, 26, 36

[FPR09] Leonor Frias, Jordi Petit et Salvador Roura : Lists revisited : Cache-conscious STL lists. *Journal of Experimental Algorithmics*, 14, 2009. 36

[FS09] Matteo Frigo et V. Strumpen : The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009. 64, 67

[Gau10] Thierry Gautier : On the cost of managing data flow dependencies. *In Workshop INRIA/UIUC/NCSA*, 2010. 54

[GBH+06] Jia Guo, Ganesh Bikshandi, Daniel Hoeflinger, Gheorghe Almási, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua et Christoph von Praun : Hierarchically tiled arrays for parallelism and locality. *In International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006. 2

[GBP07] Thierry Gautier, Xavier Besseron et Laurent Pigeon : KAAPI : A thread scheduling runtime system for dataflow computations on cluster of multi-processors. *In International Workshop on Parallel Symbolic Computation (PASCO)*, pages 15–23, 2007. 43, 53

[Gib10] Phillip B. Gibbons : Trumping the memory hierarchy with hi-spade, 2010. 71

[GMM97] Devidas Gupta, Brian Malloy et Alice McRae : The complexity of scheduling for data cache optimization. *Information Sciences*, 100(1-4):27–48, 1997. 63

[Gra69] Ronald L. Graham : Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969. 3, 45, 63

[GRCD98] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro et Mathias Doreille : Athapascan-1 : On-line building data flow graph in a parallel language. *In Parallel Architectures and Compilation Techniques (PACT)*, page 88, 1998. 2

[GRW07] Thierry Gautier, Jean-Louis Roch et Frédéric Wagner : Fine grain distributed implementation of a dataflow language with provable performances. *In International Conference on Computational Science (ICCS)*, 2007. 53

[GST70] J. Gecsei, D. R. Slutz et I. L. Traiger : Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970. 19

[HB94] C. T. Howie et E. H. Blake : The mesh propagation algorithm for isosurface construction. *Computer Graphics Forum*, 13:65–74, 1994. 101

[HJ04] Charles D. Hansen et Chris Johnson : *The Visualization Handbook*. Academic Press, 2004. 8, 95, 96

[HK81] Hong et Kung : I/O complexity : The red-blue pebble game. *In Symposium on Theory of Computing (STOC)*, 1981. 29

[HKR04] Ralf HOFFMANN, Matthias KORCH et Thomas RAUBER : Performance evaluation of task pools based on hardware synchronization. *In Supercomputing (SC)*, page 44, 2004. 46, 72

[HP06] John L. HENNESSY et David A. PATTERSON : *Computer architecture : a quantitative approach, 4th Edition*. Morgan Kaufmann Publishers Inc., 2006. 15, 62

[HRF09] Everton HERMANN, Bruno RAFFIN et François FAURE : Interactive physical simulation on multicore architectures. *In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2009. 73

[HRF+10] Everton HERMANN, Bruno RAFFIN, François FAURE, Thierry GAUTIER et Jérémie ALLARD : Multi-GPU and multi-CPU parallelization for interactive physics simulations. *In Euro-Par*, pages 235–246, 2010. 73

[HS89] M. D. HILL et A. J. SMITH : Evaluating associativity in cpu caches. *IEEE Transactions on Computing*, 38(12):1612–1630, 1989. 17

[HS02a] Danny HENDLER et Nir SHAVIT : Non-blocking steal-half work queues. *In Symposium on Principles of Distributed Computing (PODC)*, pages 280–289, 2002. 52

[HS02b] Danny HENDLER et Nir SHAVIT : Work dealing. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 164–172, 2002. 73

[HS08] Maurice HERLIHY et Nir SHAVIT : *The Art of Multiprocessor Programming*. Morgan Kaufman, 2008. 40

[HYY09] Tasuku HIRAISHI, Masahiro YASUGI et Seiji Umataniand Taiichi YUASA : Backtracking-based load balancing. *In Principles and Practice of Parallel Programming (PPoPP)*, pages 55–64, 2009. 53, 55, 57

[IK95] T. ITOH et K. KOYAMADA : Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1:319–327, 1995. 101

[JL95] Tao JIANG et Ming LI : On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995. 63

[JMJ06] Aamer JALEEL, Matthew MATTINA et Bruce L. JACOB : Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. *In International Symposium on High-Performance Computer Architecture (HPCA)*, pages 88–98, 2006. 62

[JMR09] Mathias JACQUELIN, Loris MARCHAL et Yves ROBERT : Complexity analysis and performance evaluation of matrix product on multicore architectures. *In International Conference on Parallel Processing (ICPP)*, pages 196–203, 2009. 72

[KMN+09] Mahmut T. KANDEMIR, Sai Prashanth MURALIDHARA, Sri Hari Krishna NARAYANAN, Yuanrui ZHANG et Ozcan OZTURK : Optimizing shared cache behavior of chip multiprocessors. *In International Symposium on Microarchitecture (MICRO)*, pages 505–516, 2009. 70

[KR04] Matthias Korch et Thomas Rauber : A comparison of task pools for dynamic load balancingof irregular algorithms. *Concurrency and Computation : Practice and Experience*, 16(1):1–47, 2004. 46, 72

[KV07] Alexey Kukanov et Michael Voss : The foundations for scalable multicore software inintel threading building blocks. *Intel Technology Journal*, 2007. 4, 42, 43

[LFBH00] Richard E. Ladner, Ray Fortna et Nguyen Bao-Hoang : A comparison of cache aware and cache oblivious static search trees using program instrumentation. *In Experimental Algorithmics*, pages 78–92, 2000. 35, 37

[LKA04] Joseph Leung, Laurie Kelly et James H. Anderson : *Handbook of Scheduling : Algorithms, Models, and Performance Analysis.* CRC Press, Inc., 2004. 44, 45

[LSB09] Daan Leijen, Wolfram Schulte et Sebastian Burckhardt : The design of a task parallel library. *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, 2009. 4, 43

[MHSM09] Daniel Molka, Daniel Hackenberg, Robert Schöne et Matthias S. Müller : Memory performance and cache coherency effects on an intel nehalem multiprocessor system. *In Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009. 16

[MPS02] William Margo, Paul Petersen et Sanjiv Shah : Hyper-Threading Technology : Impact on Compute-Intensive Workloads. *Intel Technology Journal*, 6(1), 2002. 17

[MS05] Moir et Nir Shavit : Concurrent data structures. *In Handbook of Data Structures and Applications.* Chapman& Hall/CRC, 2005. 40

[MTTV98] G. L. Miller, S.-H. Teng, W. Thurston et S. A. Vavasis : Geometric separators for finite-element meshes. *SIAM Journal on Scientific Computing*, 19(2):364–386, 1998. 9, 78

[MVS09] Maged M. Michael, Martin T. Vechev et Vijay A. Saraswat : Idempotent work stealing. *In Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54, 2009. 52

[Ope] Openmp tutorial of the lawrence livermore national laboratory. 4, 42, 43

[OS02] Jesper Holm Olsen et Søren Skov : Cache-oblivious algorithms in practice. Mémoire de D.E.A., Department of Computer Science, University of Copenhagen, 2002. 37

[PCDL07] Shen Pan, Cary Cherng, Kevin Dick et Richard E. Ladner : Algorithms to take advantage of hardware prefetching. *In Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007. 27

[PF01] Valerio Pascucci et R.J. Frank : Global Static Indexing for Real-Time Exploration of Very Large Regular Grids. *In Supercomputing (SC)*, page 45, 2001. 9, 34, 78

[pth] Posix threads programming tutorial of the lawrence livermore national laboratory. 42

[RTB06] Jean-Louis Roch, Daouda Traoré et Julien Bernard : On-line adaptive parallel prefix computation. *In Euro-Par*, pages 841–850, 2006. 56

[RVK08] Arch Robison, Michael Voss et Alexey Kukanov : Optimization via reflection on work stealing in TBB. *In International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2008. 53, 55, 73

[SB09] Sébastien Barbier : *Visualisation distante temps-réel de grands volumes de données*. Thèse de doctorat, Université Joseph-Fourier - Grenoble I, 10 2009. 102

[SCEL02] Claudio T. Silva, Yi-jen Chiang, Jihad El-Sana et Peter Lindstrom : Out-of-core algorithms for scientific visualization and computer graphics. *In Visualization'02 Course Notes*, 2002. 156

[Sil06] Francesco Silvestri : On the limits of cache-oblivious matrix transposition. *In International Conference on Trustworthy Global Computing (TGC)*, pages 233–243, 2006. 37

[SML04] W. Schroeder, K. Martin et B. Lorensen : *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed.* Kitware Inc., 2004. 78, 96

[Sni09] Marc Snir : Software at exascale. *In Supercomputing Panel The Road to Exascale : Hardware and Software Challenges*, 2009. 1

[SRD04] G. Edward Suh, Larry Rudolph et Srinivas Devadas : Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004. 62

[ST90] Peter Shirley et Allan Tuchman : A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, 1990. 101

[Str98] V. Strumpen : Indolent closure creation. Technical Memo MIT/LCS/TM-580, Massachusetts Institute of Technology, Laboratory for Computer Science, 1998. 53

[TASS08] David Tam, Reza Azimi, Livio Soares et Michael Stumm : Managing shared L2 caches on multicore systems in software, 2008. 62

[TCBV10] Alexandros Tzannes, George C. Caragea, Rajeev Barua et Uzi Vishkin : Lazy binary-splitting : a run-time adaptive work-stealing scheduler. *In Principles and Practice of Parallel Programming (PPoPP)*, pages 179–190, 2010. 53

[TDG+10] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec et Bruno Raffin : A work stealing scheduler for parallel loops on shared cache multicores. *In Workshop on Highly Parallel Processing on a Chip (HPPC)*, 2010. 11, 114

[TDR10a] Marc Tchiboukdjian, Vincent Danjean et Bruno Raffin : Binary mesh partitioning for cache-efficient visualization. *Transactions on Visualization and Computer Graphics*, 16(5):815–828, 2010. 11, 78, 100

[TDR10b] Marc Tchiboukdjian, Vincent Danjean et Bruno Raffin : Cache-efficient parallel isosurface extraction for shared cache multicores. *In Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2010. 11, 95

[TGT+10] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch et Julien Bernard : A tighter analysis of work stealing. *In International Symposium on Algorithms and Computation (ISAAC)*, 2010. 11, 130

[Tra09] Daouda Traoré : *Algorithmes parallèles auto-adaptatifs et applications*. Thèse de doctorat, Institut National Polytechnique de Grenoble, 2009. 56, 57, 59

[TRM+08] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier et Julien Bernard : Deque-free work-optimal parallel STL algorithms. *In Euro-Par*, pages 887–897, 2008. 53, 56

[Vit01] J. S. Vitter : External memory algorithms and data structures : dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001. 26

[vKvOB+97] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci et Dan Schikore : Contour trees and small seed sets for isosurface traversal. *In Symposium on Computational Geometry (SCG)*, pages 212–220, 1997. 101

[VOS+10] Huy T. Vo, Daniel K. Osmari, Brian Summa, Joao L. D. Comba, Valerio Pascucci et Claudio T. Silva : Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29:1073–1082, 2010. 157

[VS07] Akshat Verma et Sandeep Sen : Algorithmic ramifications of prefetching in memory hierarchy. *In High-Performance Computing (HiPC)*, pages 9–21, 2007. 27

[Wil92] Peter L. Williams : Visibility-ordering meshed polyhedra. *ACM Transaction on Graphics*, 11(2):103–126, 1992. 101

[WP05] R. C. Whaley et Antoine Petitet : Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software : Practice and Experience*, 35(2):101, 2005. 5, 6

[YLPM05] S.-E. Yoon, Peter Lindstrom, Valerio Pascucci et D. Manocha : Cache-oblivious mesh layouts. *In SIGGRAPH*, page 886, 2005. 9, 78, 156

[YRP+07] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels et Fred Gustavson : An experimental comparison of cache-oblivious and cache-conscious programs. *In Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 93–104, 2007. 35, 37

[ZJS10] Eddy Z. Zhang, Yunlian Jiang et Xipeng Shen : Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs ? *In Principles and Practice of Parallel Programming (PPoPP)*, pages 203–212, 2010. 62

Table des figures

1 Tâches et contraintes de précédence . . . 2
2 Ordonnancement par liste . . . 3
3 Ordonnancement par vol de travail . . . 3
4 Hiérarchie mémoire . . . 5
5 Trois schémas d'accès . . . 5
6 Défaut de cache de cohérence . . . 7
7 Influence de l'ordonnancement sur l'utilisation des caches . . . 7
8 Simulation de la densité d'essence au sein d'une chambre de combustion 8

1.1 Comparaison entre les données AoS et SoA . . . 21
1.2 Comparaison de deux algorithmes de multiplication de matrices . . . 23

2.1 Modèle *cache-aware* . . . 26
2.2 Modèle *cache-oblivious* . . . 27
2.3 Parcours d'un tableau . . . 28
2.4 Multiplication de matrices avec l'algorithme classique . . . 28
2.5 Multiplication de matrices par blocs . . . 29
2.6 Multiplication de matrices diviser pour régner . . . 30
2.7 Recherche dans un arbre binaire . . . 31
2.8 Recherche d'éléments dans un B-arbre . . . 31
2.9 Organisation mémoire CO d'un arbre binaire . . . 32

3.1 Processeur multicœur . . . 40
3.2 Machine NUMA . . . 41
3.3 Architecture du GPU Fermi de NVIDIA . . . 42
3.4 Algorithme de fusion parallèle . . . 46
3.5 Arbre d'activation . . . 48
3.6 Pile de tâches . . . 49

4.1 Modèle d'une architecture à caches privés . . . 64
4.2 DAG série parallèle . . . 65
4.3 Exécution dans le désordre au moment du vol . . . 66
4.4 Modèle d'une architecture à caches partagés . . . 69
4.5 Modèle *Tree-of-Caches* combinant caches privés et caches partagés . . . 71

5.1 Quatre types de maillages . . . 77

6.1 Arbre min-max . . . 96
6.2 Trois possibilités de choix de l'arbre min-max . . . 97
6.3 Deux schémas de parallélisations différents . . . 99
6.4 Prédiction du gain de *shared-cache* sur Fermi . . . 100

7.1 Histogrammes des distances de réutilisation 115

8.1 Fonction potentielle mesurant le déséquilibre de travail entre les cœurs 130
8.2 Diminution du déséquilibre de travail lors d'un vol 131
8.3 Facteurs constants 132

Table des matières

Sommaire iii

Introduction 1

I État de l'art 13

1 Applications limitées par les accès mémoire 15

1.1 Les caches 15
1.1.1 Définition 15
1.1.2 La localité des accès mémoire 16
1.1.3 Fonctionnement d'un cache 16
1.1.4 Classification des défauts de cache 17
1.2 Caractérisation des applications 17
1.2.1 Applications limitées par les accès mémoire 17
1.2.2 Prédictibilité des accès mémoire et *prefetching* 18
1.2.3 Applications limitées par la bande passante ou la latence 19
1.2.4 Réutilisation et localité temporelle 19
1.3 Les principes pour concevoir une application efficace en cache 20
1.3.1 Les 3 principes 20
1.3.2 Exemple : parcourir un ensemble de points en 3 dimensions . . . 20
1.3.3 Exemple : la multiplication de matrices 22

2 Algorithmes utilisant efficacement les caches 25

2.1 Analyser les défauts de cache d'un algorithme 25
2.1.1 Le modèle *cache-aware* 26
2.1.2 Le modèle *cache-oblivious* 26
2.1.3 Limites du modèle *cache-oblivious* 27
2.2 Analyse de quelques algorithmes classiques 27
2.2.1 Parcours d'un tableau 28
2.2.2 Multiplication de matrices 28
2.2.3 Recherche d'éléments dans un arbre 30
2.3 Conception d'algorithmes efficaces en cache 33
2.3.1 Techniques algorithmiques CA et CO 33
2.3.2 Performances en pratique des techniques CA et CO 35
2.4 Algorithmes CA et CO : revue des résultats 35
2.4.1 Revue des algorithmes CA et CO 36
2.4.2 Comparaison des algorithmes CA et CO 37

3 Programmation parallèle par vol de travail **39**
3.1 Programmation parallèle 40
3.1.1 Machine parallèle à mémoire partagée 40
3.1.2 Interfaces de programmation parallèle 42
3.1.3 Programmation parallèle à base de tâches 43
3.1.4 Exemple : algorithme parallèle de tri fusion 43
3.2 Ordonnancement d'un programme parallèle à base de tâches 44
3.2.1 Graphe de précédences, travail, profondeur 44
3.2.2 Ordonnancement avec une liste centralisée 44
3.2.3 Temps d'exécution du tri fusion 45
3.2.4 Surcoût d'une gestion centralisée des tâches 46
3.3 Ordonnancement par vol de tâches 46
3.3.1 Une liste de tâches décentralisée 47
3.3.2 Garantie sur le nombre de vols 47
3.4 Programmation par tâches efficace 50
3.4.1 Surcoûts par rapport au programme séquentiel 50
3.4.2 La gestion des listes de tâches 52
3.4.3 La création des tâches 52
3.4.4 La gestion des dépendances entre les tâches 53
3.4.5 Le surcoût algorithmique 54
3.5 Programmation parallèle adaptative 55
3.5.1 Algorithmes parallèles adaptatifs 55
3.5.2 Algorithme adaptatif de tri fusion 56
3.5.3 Moteur adaptatif à vol concurrent 57
3.5.4 Moteur adaptatif à vol coopératif 58
3.5.5 La préemption 59

4 Algorithmes parallèles efficaces en cache **61**
4.1 Impact du parallélisme sur les caches 61
4.1.1 Caches privés et caches partagés 62
4.1.2 La cohérence de cache 62
4.1.3 Modéliser les accès mémoire d'un algorithme parallèle 63
4.2 Ordonnancement pour caches privés 64
4.2.1 Modéliser les caches privés 64
4.2.2 Ordonnancement par vol de travail pour caches privés 65
4.2.3 Raffiner l'analyse pour les algorithmes CO 67
4.2.4 Algorithmes parallèles pour caches privés 68
4.3 Ordonnancement pour un cache partagé 68
4.3.1 Modéliser un cache partagé 68
4.3.2 Ordonnancement avec liste centralisée pour cache partagé 69
4.3.3 Algorithmes parallèles pour cache partagé 70
4.4 Vers un ordonnanceur pour le cas général 71
4.4.1 Modèle combinant caches privés et caches partagés 71
4.4.2 Approches pour traiter le cas général 71
4.4.3 Algorithmes parallèles pour multicœurs 72
4.4.4 Ordonnancements basés sur l'affinité 73

II Contributions 75

5 Maillage cache-oblivious pour la visualisation scientifique 77

5.1 Résumé des contributions 78
5.2 Discussion et perspectives 80
5.3 Binary Mesh Partitioning for Cache-Efficient Visualization 81
1 Introduction 81
2 Related Work 82
2.1 Cache-Efficient Algorithms 82
2.2 Mesh Layouts 83
2.3 Isosurface Extraction 83
3 Framework 83
3.1 Common Mesh Access Patterns 83
3.2 Layout influence on Cache Performance 84
3.3 The Access Graph Model 84
3.4 Access Graph for Layout Order Traversals 85
3.5 Restriction to Overlap Graphs 85
4 Overlap Graphs Partitioning 86
4.1 Overlap Graphs 86
4.2 Geometric Separator Algorithm 86
5 Recursive Mesh Layout 86
5.1 Mesh Layout Algorithm 87
5.2 Layout Quality 87
5.3 Layout Computation 88
5.4 Choosing the Access Graph 88
5.5 Cells Layout 88
5.6 Consistent BSP Tree 89
6 Experiments 89
6.1 Architectures, Filters and Meshes 89
6.2 Layout Algorithm Performance 90
6.3 Mesh Layout Performance 90
7 Conclusion 93
References 94

6 Extraction d'isosurfaces parallèle et efficace en cache 95

6.1 Résumé des contributions 95
6.2 Discussion et perspectives 101
6.3 Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores 103
1 Introduction 103
2 Marching Tetrahedra Review 104
3 Cache-Efficient Isosurface Extraction 105
3.1 Source of Cache Misses in MT 105
3.2 Cache-Oblivious Model (CO) 105
3.3 CO Mesh Layout 105
3.4 MT Cache Performance 106
3.5 Tree Accelerated MT Cache Performance 106
4 Parallel Cache-Efficiency 106

4.1 *Shared Cache Multicore* 106
4.2 *Shared Cache Aware Parallelization* 106
4.3 *Parallel MT* 107
4.4 *Parallel Tree Accelerated MT* 107
5 *Implementation and Experiments* 107
5.1 *Architectures and Meshes* 107
5.2 *Min-Max Tree Implementation* 107
5.3 *Sequential Performance* 108
5.4 *Parallel MT Implementation* 108
5.5 *Parallel MT Performance* 108
5.6 *Parallel Min-Max Tree Implementation* 109
5.7 *Parallel Min-Max Tree Performance* 109
5.8 *Measure of Locality* 109
6 *Related Work* 111
7 *Conclusion* 111
References 112

7 Vol de travail efficace en cache pour les boucles parallèles **113**
7.1 Résumé des contributions 114
7.2 Discussion et perspectives 116
7.3 A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores 119
1 *Introduction* 119
2 *Scheduling for Efficient Shared Cache Usage* 120
2.1 *Review of Work-Stealing and Parallel Depth First Schedules* 120
2.2 *Window Algorithms for Sequence Processing* 121
2.3 *Cache Performance of Window Algorithms* 121
2.4 *PThread Parallelization of Window Algorithms* 122
3 *Work-Stealing Window Algorithms with Kaapi* 123
3.1 *Kaapi Overview* 123
3.2 *Work-Stealing Algorithm for Standard (*no-window*) Processing* 124
3.3 *Work-Stealing Window Algorithms* 124
4 *Marching Tetrahedra for Isosurface Extraction* 125
5 *Experiments* 125
5.1 *Calibrating the Window Algorithms* 126
5.2 *Comparison of Parallel Libraries on* `for_each` 126
5.3 *Performance of the Window Algorithms* 127
6 *Related works* 127
7 *Conclusions* 127
References 128

8 Nouvelle analyse des ordonnancements par vol de travail **129**
8.1 Résumé des contributions 130
8.2 Discussion et perspectives 131
8.3 Decentralized List Scheduling 133
1 *Introduction* 134

1.1 Context and motivations . . . 134
1.2 Related Works . . . 134
1.3 Contributions . . . 135
1.4 Content . . . 136
2 Model and Notations . . . 136
2.1 Platform and workload characteristics . . . 136
2.2 Centralized list scheduling . . . 136
2.3 Decentralized list scheduling . . . 136
2.4 Model of the distributed list . . . 137
2.5 Properties of the work . . . 138
3 Principle of the analysis and main theorem . . . 138
4 Unit independent tasks . . . 141
4.1 Potential function and expected decrease . . . 141
4.2 Bound on the makespan . . . 142
4.3 Importance of the initial repartition of tasks . . . 143
5 Going further on the unit tasks model . . . 144
5.1 Improving the analysis by changing the potential function 144
5.2 Cooperation among thieves . . . 145
6 Weighted independent tasks . . . 147
6.1 Definition of the potential function and expected decrease 147
6.2 Bound on the makespan . . . 148
7 Tasks with precedences . . . 148
7.1 ABP work-stealing scheduler . . . 148
7.2 Definition of $w_i(t)$ . . . 149
7.3 Bound on the makespan . . . 149
8 Experimental study . . . 151
8.1 Distribution of the makespan . . . 151
8.2 Study of the $\log_2 W$ *term* . . . 152
9 Concluding remarks . . . 153
References . . . 154

Conclusion **155**

Bibliographie **159**

Table des figures **169**

Table des matières **171**

Résumés **177**

Résumé. Pour exploiter pleinement les performances des processeurs actuels, il est nécessaire de prendre en compte le coût important des accès mémoire et l'augmentation du nombre de cœurs. Les algorithmes *cache-oblivious* maximisent l'utilisation des caches réduisant ainsi le coût des accès mémoire. Les ordonnanceurs par vol de travail répartissent équitablement et dynamiquement les calculs d'une application sur différents cœurs. De plus, les algorithmes basés sur ces techniques sont capables de s'adapter automatiquement à l'environnement d'exécution indépendamment des paramètres de l'architecture tels que la taille des caches ou le nombre de cœurs. Dans cette thèse, nous étudions la combinaison de ces deux techniques pour développer des algorithmes parallèles efficaces en cache. Nous visons en particulier les applications de visualisation scientifique qui sont consommatrices à la fois en accès mémoire et en puissance de calcul.

Nous proposons une nouvelle organisation en mémoire *cache-oblivious* pour les maillages non structurés qui améliore la localité des accès mémoire des filtres de visualisation. Cette organisation est obtenue par une découpe récursive du maillage utilisant le séparateur de Miller *et al.*. Nous donnons une garantie théorique sur le nombre de défauts de cache générés dans le cas de schémas d'accès classiques. Des expérimentations sur de nombreux filtres de visualisation usuels confirment des gains de performance significatifs sur CPU et sur GPU. Nous montrons par ailleurs que la découpe récursive du maillage détermine des intervalles qui peuvent être utilisés comme un arbre min-max pour accélérer l'extraction d'isosurfaces. La consommation mémoire est réduite d'un facteur 3 et la localité est encore améliorée.

Dans les processeurs récents, les cœurs partagent souvent le dernier niveau de cache. Pour en favoriser l'usage coopératif, nous proposons un ordonnanceur par vol de travail qui incite les cœurs à accéder en priorité à des données proches en mémoire de celles déjà chargées dans ce cache. Appliqué à l'extraction d'isosurface, cet ordonnanceur améliore les performances de 20% comparé à un ordonnanceur classique. Enfin ce travail nous a conduit à développer une nouvelle analyse théorique du vol de travail, plus précise que l'analyse classique de Blumofe *et al.*. Cette analyse nous permet d'évaluer la performance de nouveaux mécanismes de vols sur des algorithmes à grain fin tel que l'implémentation de l'ordonnanceur pour caches partagés.

Mots-clés : algorithmes efficaces en cache, algorithmes parallèles, ordonnancement par vol de travail, visualisation scientifique, extraction d'isosurfaces.

Abstract. To leverage the power of modern processors, algorithm design must take into account the cost of memory access and the increasing number of cores. Cache-oblivious algorithms fully exploit several levels of cache to reduce the cost of memory access. Work stealing schedulers are an efficient and scalable way to dynamically balance the workload on multiple cores. Moreover, algorithms based on these techniques can automatically adapt to the execution platform without tuning parameters such as the cache size or the number of cores. In this thesis, we propose to combine both techniques to design parallel cache-efficient algorithms. We target scientific visualization applications that need to process huge datasets interactively and thus greatly benefit from those improvements.

We develop a cache oblivious mesh layout, *i.e.* an algorithm to reorder points and cells of an unstructured mesh, which enhances locality of visualization filters. This layout is obtained by recursively dividing the mesh using the separator of Miller *et al.*. We give a theoretical performance guarantee on the number of cache misses for common access patterns. Experimental validation shows significant speed up both on CPU and GPU for many standard visualization filters. Moreover, we show that the same recursive decomposition of the mesh can be used to build a layout consistent min-max tree to accelerate isosurface extraction. Memory consumption is reduced by a threefold factor and locality is further enhanced.

The cores of modern processors often share the last level of cache. To favor its cooperative use, we design a new work stealing scheduler which ensures that cores access to data close to data currently stored in the shared cache. Applied to isosurface extraction, this scheduler brings a 20% performance improvement. Our last contribution is a refined theoretical analysis of work stealing schedulers. This analysis is more precise than the classic analysis of Blumofe *et al.* and thus allows to evaluate the performance of new work stealing mechanisms on fine grain workloads such as the scheduler for shared caches.

Keywords: cache-aware and cache-oblivious algorithms, parallel algorithms, work-stealing schedulers, scientific visualization, isosurface extraction.