HAL
open science

Cartographier la mémoire virtuelle d'une application de calcul scientifique

David Beniamine

To cite this version:

David Beniamine. Cartographier la mémoire virtuelle d'une application de calcul scientifique. ComPAS'13 / RenPar'21, Jan 2013, Grenoble, France. hal-01120000

HAL Id: hal-01120000

https://inria.hal.science/hal-01120000

Submitted on 24 Feb 2015

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

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

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Cartographier la mémoire virtuelle d'une application de calcul scientifique

David Beniamine *

Université Joseph Fourrier,
Laboratoire LIG - Bâtiment ENSIMAG de Montbonnot - 51 avenue Jean Kuntzmann
38330 Montbonnot Saint Martin - France
David.Beniamine@imag.fr

Résumé
L'étude de la mémoire principale est un point clef lors du développement d'applications parallèles ou d'applications de calcul haute performance. En premier lieu cette ressource est beaucoup plus lente que le processeur. Son utilisation trop intensive pourrait ainsi provoquer des contentions sur le bus mémoire et donc des baisses de performances. En second lieu, parce que les mémoires *cache* qui servent à palier ce problème sont très petites. Les développeurs sont ainsi amenés à prendre en compte leur spécificité au travers de méthodes telles que le *data padding* (c'est à dire l'ajout des octets inutilisés entre deux structures pour les aligner en mémoire). Les différents threads d'applications parallèles partagent la mémoire principale. Ce partage est un point clef dans l'optimisation d'applications parallèles : dès que des threads travaillent en parallèle sur la mémoire ils partagent le cache ce qui demande encore plus d'attention de la part du développeur.
Dans cette étude nous proposons une nouvelle méthode d'analyse de la mémoire qui établit une cartographie de cette dernière afin d'aider les programmeurs à comprendre et à visualiser le comportement de leurs applications. Nous validons ensuite cette méthode à l'aide d'exemples d'application.

Mots-clés : localité temporelle, localité spatiale, distance de réutilisation, working set, cartographie

---

1. Introduction

Plus de deux cents cycles de processeur peuvent être nécessaires pour accéder à la mémoire [5], une sur-utilisation ou une mauvaise utilisation de cette ressource peut faire chuter les performances d'une application. Les mémoires cache permettent d'améliorer la gestion de la mémoire, mais elles demeurent très petites (quelques Mio) comparées à la mémoire principale (de l'ordre du Gio). L'organisation des données dans la mémoire principale et la manière dont un programme y accède sont donc très importantes. Les outils qui permettent de comprendre l'utilisation faite de la mémoire par une application sont ainsi utiles à tout développeur qui souhaite optimiser ses applications.
Dans cette étude nous proposons de mettre en place un outil basé sur l'instrumentation binaire d'un programme pour cartographier son utilisation de la mémoire. Cette cartographie correspond à la répartition des accès aux différentes structures de données au fil du temps ; elle apporte une information temporelle qui permet de visualiser précisément quelle partie de l'application peut poser problème. Elle permet par ailleurs de différencier les lectures des écritures, et les données partagées entre plusieurs threads des données privées.
Nous discuterons dans la section 2 des travaux connexes. Nous détaillerons ensuite dans la section 3 les objectifs et le fonctionnement de notre outil, avant de le valider sur des exemples d'utilisation dans la section 4.

---

*. supervisé par Guillaume Huard et Swann Perarnau

2. État de l'art

De nombreuses techniques permettent d'utiliser efficacement les caches pour palier à la lenteur de la mémoire principale. Il est cependant presque toujours nécessaire pour ce faire d'avoir un certain nombre d'informations sur l'utilisation de la mémoire.

La *distance de réutilisation* (ou stack *distance*) est souvent utilisée comme métrique lors de l'optimisation du cache. La distance de réutilisation d'un accès mémoire a correspond au nombre d'adresses différentes accédées depuis le dernier accès à a ; elle permet de comprendre les besoins en caches réels pour une application [2], mais elle est toutefois très difficile à mesurer directement (il faudrait pour ce faire intercepter tous les accès mémoire). Il est néanmoins possible de s'en approcher avec d'autres mesures, telles que celle des *working sets* - c'est à dire la mesure des performances d'une application en fonction de la quantité de cache donnée à chacune de ses structures de données.

Il est possible de créer des *partitions* dans le cache afin d'empêcher deux structures de s'y gêner ; les working sets offrent alors des informations utiles au choix du contenu et de la taille des partitions [9]. Cette méthode permet des gains de performance de près de 40% (dans les meilleurs cas) mais la mesure est très lente : pour observer les variations de performances dues au cache il est nécessaire de travailler sur des problèmes de taille réelle. Il serait intéressant de pouvoir obtenir cette métrique en travaillant sur des entrées de taille réduite.

Certains outils tels que ULCC proposent à l'utilisateur de créer ses partitions, de spécifier pour chaque partition uniquement si les données qu'elle contient sont partagées ou non entre des threads, et si elles ont une forte ou une faible distance de réutilisation [4]. Un outil permettant d'obtenir ces informations pourrait donc faciliter le travail du développeur. Le logiciel Soft-OLP est capable de dresser un profil pour chaque objet de l'application à partir d'entrées tests en traçant tous les accès mémoire. Lors de l'exécution le logiciel décide a partir de ces données d'une partition du cache en fonction de l'entrée [7]. L'analyseur de Soft-OLP est proche de ce que nous cherchons à faire mais il ne fournit pas d'information sur le partage d'un objet entre différents threads.

Une autre approche de l'optimisation de cache consiste à modifier le code de l'application soit en utilisant des informations données par le compilateur lors de la phase d'optimisation [3] ; soit en prenant en compte le cache dans la conception de l'algorithme et en l'adaptant [10]. Avec cette approche les améliorations sont mesurées en comptant le nombre de défauts de caches gagnés. Ces mesures peuvent toutefois être faussées par des mécanismes tels que le *pre-fetching*. Une cartographie de la mémoire permettrait de visualiser clairement les modifications apportées par ces optimisations et éventuellement d'en concevoir de nouvelles.

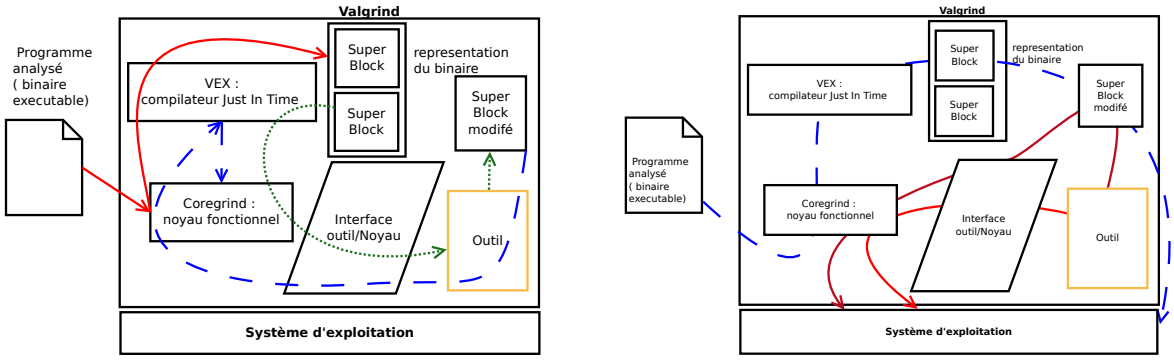
3. HeapInfo : un outil d'analyse de la mémoire

3.1. Outils existants

Il existe différents programmes notamment Valgrind[8] et Pin[6] qui permettent d'analyser la mémoire. Ces deux outils sont assez semblables dans leur mécanisme et sont composés de deux entités : le noyau qui gère l'exécution simulée du programme analysé (appelé client), et l'outil qui détermine l'instrumentation à faire du client. Dans cette étude, nous nous plaçons au niveau de l'outil.

Nous avons décidé de travailler sur Valgrind car il possède plusieurs outils permettant d'analyser la mémoire, et ce de différents points de vue. Son noyau est constitué de deux modules : Coregrind qui gère l'aspect fonctionnel (interaction avec le système d'exploitation et simulation du programme analysé), et la librairie VEX qui gère la compilation *Just In Time*. La figure 1a montre le fonctionnement de l'exécution d'un programme sur Valgrind. Le fichier binaire (du programme analysé) est d'abord traduit par Coregrind à l'aide de la librairie VEX dans un format interne (flèches pleines). Dans ce format l'exécutable est divisé en suites d'instructions séquentielles (sans instructions de type *jump*) appelées *super bloc*. Coregrind choisi alors le prochain super bloc à exécuter, le donne à l'outil (flèches en pointillés) qui va instrumenter le super bloc en y ajoutant des *handlers*, rendre la version modifiée à Coregrind (flèches discontinues) qui la compilera et l'exécutera. Cette méthode d'analyse au fil de l'exécution permet à l'outil de modifier très facilement son instrumentation au cours du temps. L'interface avec le noyau lui donne aussi la possibilité de s'abonner à des événements standards tels que l'allocation mémoire ou la création de threads. Enfin, comme le montre la figure 1b, lorsque l'outil a besoin d'utiliser des fonctions

de la librairie C, il doit utiliser l'interface proposée par le noyau (flèches pleines) afin que Coregrind les différencie des appels effectués par le client (flèche discontinues).



(a) Instrumentation d'un programme

(b) Déclenchement d'appel système

FIGURE 1 – Schémas du fonctionnement de Valgrind

Parmi les outils proposés par Valgrind certains semblent être utiles pour notre étude. Le premier d'entre eux, à savoir Cachegrind, dispose d'un simulateur de caches qui lui permet de tracer les défauts de caches d'un programme avec une précision variable (ligne de code, fonction). Bien que cet outil apporte des informations intéressantes, l'observation des défaut de cache ne permet pas d'avoir une vision globale de la mémoire : certaines informations telles que le nombre d'accès à un emplacement ou la localisation des accès en mémoire peuvent être masquées. Viennent ensuite Dhat et Massif qui permettent d'analyser le tas. Ces outils apportent des informations pertinentes -telles qu'un graphique indiquant la variation de la quantité de mémoire utilisée, mais sont trop peu précis par rapport aux informations que nous voulons obtenir. Enfin Lackey, l'outil exemple de Valgrind affiche simplement tous les accès mémoire (lecture, écriture ou lecture d'instruction) ainsi que leur taille. Il semble donc être à même de nous permettre d'obtenir une cartographie des accès. Mais cet outil est malheureusement trop lourd : l'analyse du benchmark NAS LU en classe S[1] sur une machine dotée d'un processeur Pentium 4 et de 2Gio de RAM exécutant a duré plus de quatre heures et généré un fichier de sortie de 40Gio, alors qu' en temps normal ce programme s'exécute en quelques secondes.

3.2. HeapInfo un nouvel outil pour Valgrind

Valgrind est un programme approprié pour notre analyse mais nous avons vu précédemment que les outils existants ne permettent pas de la réaliser en un temps raisonnable ; nous avons donc développé un outil : Heapinfo que nous présentons dans cette section.

Heapinfo intercepte les allocations mémoire (malloc, new etc.) et crée un bloc de méta données pour chaque allocation. Un tel bloc contient notamment l'identifiant du thread créateur, l'estampille temporelle de la création de la structure, et la liste des accès à cette structure. L'utilisateur a la possibilité de nommer ces structures en utilisant le mécanisme de requêtes client de Valgrind.

Chaque accès mémoire est intercepté mais seuls les accès à des structures allouées dynamiquement sont traités, enregistrés, et associés à la structure de données concernée. Cela permet de ne traiter que les informations utiles. Afin de faire ressortir l'information temporelle, une horloge logique est maintenue et incrémentée à chaque fois qu'un nouvel accès est pris en compte. Le traitement des accès mémoire qui constitue le point central de notre analyse est détaillé dans l'algorithme 1.

Nous avons vu dans la section 3.1 avec l'exemple de Lackey qu'il n'est pas envisageable de traiter tous

Algorithm 1 Algorithme de HeapInfo

```
function ACCESMEMOIRE(tid, addresse, taille, type)        ▷ Le thread tid a déclenché un
                                                          ▷ accès mémoire ,caractérisé par
                                                          ▷ sa taille, son type (lecture ou écriture)
                                                          ▷ et l'adresse ou il s'est produit.
    Bloc b ;
    if existe(b=trouverBlocContenant(adresse, taille)) then   ▷ Cet accès se situe sur une
                                                          ▷ structure de donnée observée, il
                                                          ▷ est ajouté aux données la concernant.
        ajouterAccessDansBloc(b, tid, adresse, taille, type) ;
    end if
end function
function AJOUTERACCESSDANSBLOC(b, tid, adrresse, taille, type)
    Accès a ;
    if existe(a=trouverAccesDansSeuils(adresse)) then
        fusionner(a, tid,adresse, taille) ;               ▷ Si le dernier accès à la même portion
                                                          ▷ (ex : page mémoire) du bloc s'est produit
                                                          ▷ à une date plus proche que le seuil de
                                                          ▷ fusion temporelle, il y a fusion des accès
    else
        a=NouvelAcces(tid, addresse, taille, type) ;      ▷ On initialise un nouvel accès
        Ajouter(b,a) ;                                    ▷ et on l'ajoute à ceux du bloc concerné
    end if
end function
```

les accès à la mémoire individuellement. Afin de réduire la quantité d'information à gérer, nous avons mis en place une fusion de ces accès que nous formalisons de la manière suivante.
Soit S l'ensemble des structures de données allouées.
Nous divisons chaque structure $S_i \in S$ en $k \in \mathbb{N}$ blocs de taille H notés $(S_{i0}, .., S_{ik-1})$.
Un accès a est caractérisé par trois valeurs :

1. $t(a)$, qui correspond à l'instant auquel a s'est produit ;
2. $tfin(a)$ défini de la manière suivante : si a est le résultat de la fusion de plusieurs accès, $tfin(a)$ est le moment auquel s'est produit le dernier des accès ayant été fusionné avec a, sinon, $tfin(a) = t(a)$;
3. et $bloc(a) = S_{ij}$ est le bloc accédé à l'instant $t(a)$. Nous notons $A_{ij} = \{a | bloc(a) = S_{ij}\}$ l'ensemble des accès au bloc S_{ij}.

Si un accès a_m se produit sur le bloc S_{ij} et que $\exists\ a_l \in A_{ij}$ tel que $t(a_m) - t(a_l) \leq T$ alors a_l et a_m seront fusionnés de la manière suivante :
– $t(a_{fusion}) = min(t(a_l), t(a_m))$;
– et $tfin(a_{fusion}) = max(t(a_l), t(a_m))$.
L'accès a_l est alors remplacé par l'accès a_{fusion}. La figure 2 représente graphiquement cette fusion
Les grandeurs T et H sont des constantes correspondant respectivement aux seuils de fusion temporelle et de fusion spatiale ; elles sont réglables par l'utilisateur.
Pour mesurer l'impact de la fusion, nous avons effectué plusieurs cartographies de la NAS LU en faisant varier ces seuils et nous avons noté le temps d'exécution ainsi que la taille de la sortie de HeapInfo. Le tableau 1 présente les résultats de cette expérience. Nous constatons que la fusion permet de réduire le temps d'exécution, cependant elle entraîne dans le même temps une perte d'information. Il est donc nécessaire de bien choisir les deux seuils afin d'obtenir une cartographie précise en un temps raisonnable.

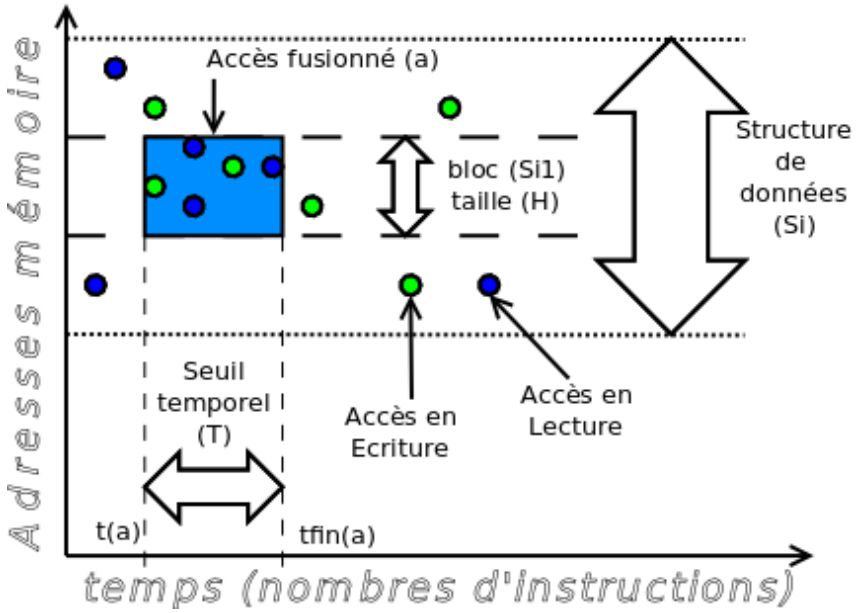


FIGURE 2 – Schéma de la fusion des accès mémoire dans Heapinfo

4. Exemples d'utilisations d'heapinfo

Dans cette partie, nous validerons notre outil en présentant la cartographie de différentes applications. Les expériences décrites ici ont été réalisées sur la plateforme grid5000, sur une machine du cluster genepi possédant 2 processeurs Intel Xeon E5420 QC, chacun dotés de quatre coeurs ainsi que 8Gio de RAM.

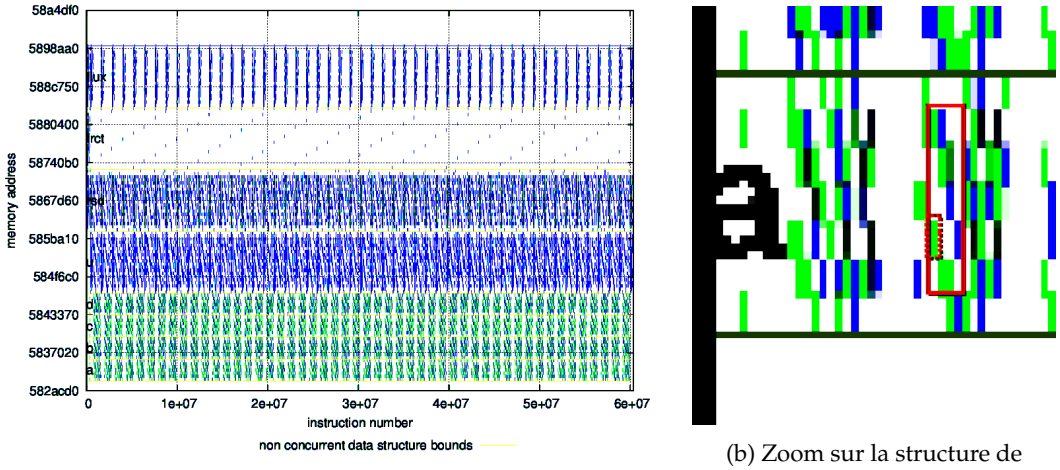
4.1. Mesure de working set

La figure 3a montre la cartographie de la NAS-bench LU en classe S et en version séquentielle ; on y voit les accès à la mémoire en fonction du temps. Chaque structure de données est délimitée par deux lignes horizontales. Les blocs colorés correspondent à un ensemble d'accès fusionnés ; la couleur d'un bloc dépend de deux paramètres : si le bloc est considéré concurrent (des accès venant de plusieurs threads ont été fusionnés) ou non, et la proportion de lecture ou d'écriture. Ainsi, un bloc non concurrent sera bleu

Expérience	Fusion spatiale	seuil de fusion temporelle	temps d'exécution (s)	taille de la sortie (Mio)
Lackey	aucune	aucune	>14400	> 4096
Heapinfo	adresses consécutives	0	1187	5046
	adresses consécutives	10	41	144
	adresses consécutives	100	20	58
	adresses consécutives	1000	9	1
	ligne de cache	0	361	1516
	ligne de cache	10	21	59
	ligne de cache	100	13	23
	ligne de cache	1000	9	1
	page mémoire	0	163	651
	page mémoire	10	13	20
	page mémoire	100	11	8
	page mémoire	1000	9	<1

TABLE 1 – Temps d'exécution et taille des fichiers de sortie de NAS LU mono-thread en classe S selon les seuils

foncé s'il ne contient que des lectures, vert clair s'il ne contient que des écritures, et s'il contient des lectures et des écritures, sa couleur se situera entre le bleu et le vert (et dépendra du ratio écriture/lecture). De même, pour un accès concurrent les couleurs vont du orange (écriture) au pourpre (lecture).



(b) Zoom sur la structure de donnée A

(a) cartographie complete

FIGURE 3 – Cartographie de NAS LU en classe S

Nous définissons la prochaine occurrence d'un accès a de la manière suivante :
- si $\exists\ a'$ tel que $\mathtt{bloc}(a) = \mathtt{bloc}(a')$ et $\forall\ a''$ tel que $\mathtt{bloc}(a'') = \mathtt{bloc}(a)$, $t(a'') < t(a)$ ou $t(a'') > t(a')$, alors $\mathtt{suiv}(a) = a'$)
- sinon $\mathtt{suiv}(a) = \mathtt{nil}$.

Un accès a peut être évincé du cache par l'ensemble $\mathtt{Int} = \{a_i \mid t(a) \leq t(a_i) \leq t(\mathtt{suiv}(a))\}$. Ainsi, pour que a bénéficie du cache, il faut que l'ensemble B des blocs b tels que $\exists a_i \in \mathtt{Int}$ et $\mathtt{bloc}(a_i) = b$ tienne dans le cache ; i.e. $\#(B) * H < C$ où C est la taille du cache.

Si nous nous concentrons sur la structure A, il est possible de trouver la distance de réutilisation d'un accès graphiquement. Observons par exemple l'écriture encadrée en pointillés sur la figure 3b. L'ensemble $\mathtt{Int}$ des accès intermédiaires correspond au rectangle continu. Or ce rectangle correspond à cinq blocs différents. La cartographie ayant été réalisée avec un seuil spatial de 4Kio, il faut $4 * 5 = 20\mathtt{kio}$ de cache pour réutiliser cet accès. La table 2 nous donne la taille des différentes structures de LU en fonction de leur classe. En classe S, nous avons besoin de 20Kio soit 71% de la taille de A. Nous en déduisons qu'en classe B, la structure A aura besoin de 1.35Mio de cache.

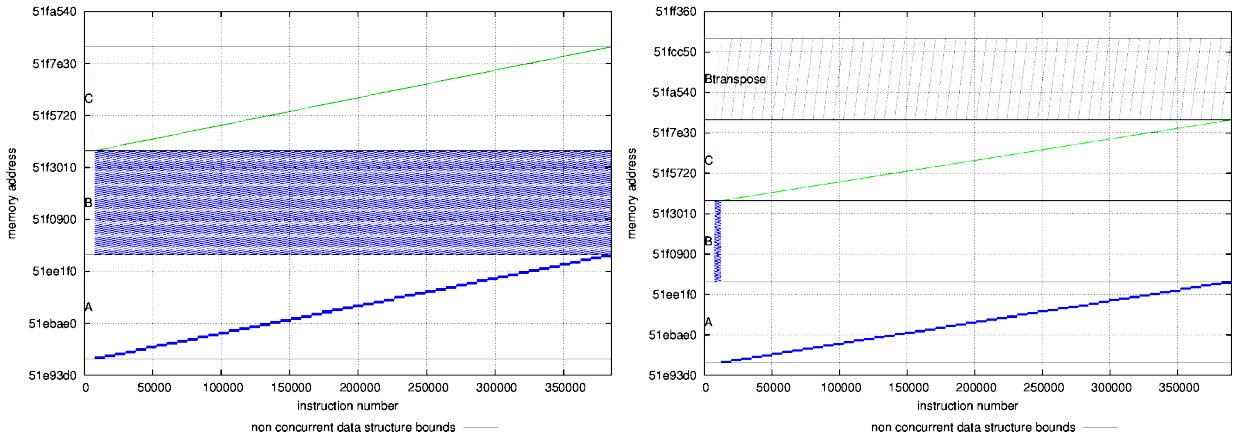
Classe	taille a,b,c,d	taille u,rsd, frct,flux
S (petite)	28,12 kio	79,22 kio
B (rélle)	1,98 Mio	4,13 Mio

TABLE 2 – taille des structures de NAS LU pour les classe S et B

Bien entendu cette méthode graphique n'est pas très précise ; c'est pourquoi Heapinfo fournit une sortie textuelle qui peut être facilement lue par un script utilisateur afin d'extraire automatiquement ce type d'information.

4.2. Reconnaître des schémas d'utilisation de la mémoire

Un exemple classique de l'importance de la gestion du cache est le produit matriciel : les matrices 2D sont en général représentées par des grands tableaux. Effectuons de manière naïve l'opération $C = C + A * B$ où A,B et C sont des matrices carrées de taille $N * N$. Nous allons parcourir les matrices A et C ligne par ligne et la matrice B colonne par colonne (ce qui implique des *sauts* dans la mémoire, comme le montre la figure 4a). Ces sauts sont une nuisance du point de vue du cache car ce dernier copie toute une ligne de la mémoire ($\simeq 128octets$) à chaque accès. Dans ce cas, non seulement il ne profite pas de ce système, mais en plus chaque accès à un élément de B évincera potentiellement un élément de A ou de C qui aurait pu être réutilisé. Il existe de nombreuses solutions algorithmiques pour résoudre ce problème, la plus simple consiste à transposer la matrice B. Ce procédé implique un surcoût mais peut s'avérer très rentable. La figure 4b correspond à la cartographie d'une telle version du produit matriciel : on voit que la matrice B n'est jamais parcourue plus d'une fois par colonne puis que la matrice $Btranspose$ est parcourue à plusieurs reprises en séquence.



(a) Produit matriciel *naïf*

(b) Produit matriciel transposé

FIGURE 4 – Cartographies de l'exécution de produits matriciels

5. Conclusion

Dans cette étude, nous avons développé un outil d'analyse de la mémoire d'une application. Ce dernier identifie les structures de données et maintient un historique des accès à ces structures en différenciant les écritures des lectures et les données partagées des données privées. Il permet d'obtenir une cartographie de la mémoire et offre par là même une vision globale de l'exécution ainsi qu'une sortie textuelle facilement exploitable par un script. Contrairement aux outils qui se reposent sur les compteurs matériels, Heapinfo offre la possibilité de travailler sur des entrées de petite taille, donc de travailler sur une exécution courte.

La première limite de notre outil provient de l'exécution en série des programmes parallèles par Valgrind. Ce mode d'exécution permet de simplifier l'analyse de la mémoire mais il ralentit de manière conséquente l'analyse de programmes parallèles. Le fait de travailler au niveau de la mémoire permet toutefois d'utiliser des petites instances du programme étudié et d'extrapoler nos observations pour l'exécution réelle. Une partie du problème est ainsi palliée.

Heapinfo apporte une cartographie de la mémoire où les différentes structures de données sont clairement identifiables. Mais pour les programmes utilisant des structures complexes (arbre, liste chaînées etc.) la cartographie peut devenir illisible, chaque allocation étant considérée comme une structure à part entière. Détecter lors de l'analyse les dépendances entre les structures de données peut s'avérer complexe, et risque de ralentir l'analyse. Valgrind fournit cependant un mécanisme de requêtes client grâce

auquel l'utilisateur peut fournir ce type d'information à l'outil. Nous prévoyons donc d'implémenter la gestion des structures de données complexes à l'aide de ce mécanisme pour les versions futures.
La principale perspective de travail pour la suite de cette étude consiste à employer Heapinfo afin d'optimiser des applications parallèles et à généraliser la méthode utilisée pour la rendre la plus automatique possible.

Remerciements

Je tiens à remercier particulièrement Guillaume Huard pour sa présence et sa patience, ainsi que Swann Perarnau qui, malgré la distance, prend toujours le temps de suivre mon travail et de répondre a mes questions.
Merci aussi à Marion Jeannin qui bien que n'étant pas informaticienne, a pris le temps de relire ce papier plusieurs fois.
Les expériences présentées dans cet article on été réalisées sur la plate-forme expérimentale Grid 5000, issue de l'Action de Développement Technologique (ADT) ALADDIN pour l'INRIA avec le support du CNRS, de RENATER, de plusieurs Universités et d'autres contributeurs (voir https://www.grid5000.fr).

Bibliographie

1. Bailey (D. H.), Barszcz (E.), Barton (J. T.), Browning (D. S.), Carter (R. L.), Dagum (D.), Fatoohi (R. A.), Frederickson (P. O.), Lasinski (T. A.), Schreiber (R. S.), Simon (H. D.), Venkatakrishnan (V.) et Weeratunga (S. K.). – The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, vol. 5, n3, Fall 1991, pp. 63–73.
2. Beyls (K.) et D'Hollander (E.). – Reuse distance as a metric for cache behavior. *In : Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, pp. 617–662. – Dallas, Texas, USA, Aug 2001.
3. Bugnion (E.), Anderson (J. M.), Mowry (T. C.), Rosenblum (M.) et Lam (M. S.). – Compiler-directed page coloring for multiprocessors. *SIGPLAN Not.*, vol. 31, September 1996, pp. 244–255.
4. Ding (X.), Wang (K.) et Zhang (X.). – Ulcc : a user-level facility for optimizing shared cache performance on multicores. *In : Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. pp. 103–112. – San Antonio, Texas, USA, Feb 2011.
5. Drepper (U.). – What every programmer should know about memory, 2007.
6. keung Luk (C.), Cohn (R.), Muth (R.), Patil (H.), Klauser (A.), Lowney (G.), Wallace (S.), Janapa (V.) et Hazelwood (R. K.). – Pin : Building customized program analysis tools with dynamic instrumentation. *In : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. pp. 190–200. – Chicago, Illinois, USA, Jun 2005.
7. Lu (Q.), Lin (J.), Ding (X.), Zhang (Z.), Zhang (X.) et Sadayappan (P.). – Soft-olp : Improving hardware cache performance through software-controlled object-level partitioning. *In : Proceedings of 18th International Conference on Parallel Architectures and Compilation Techniques (PACT 2009)*, pp. 246–257. – Raleigh, North Carolina,USA, Sep 2009.
8. Nethercote (N.) et Seward (J.). – Valgrind : A framework for heavyweight dynamic binary instrumentation. *In : Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pp. 89–100. – San Diego, California, USA, Jun 2007.
9. Perarnau (S.). – *Environnements pour l'analyse expérimentale d'applications de calcul haute performance*. – These, Université de Grenoble, Dec 2011.
10. Tchiboukdjian (M.), Danjean (V.) et Raffin (B.). – A Fast Cache-Oblivious Mesh Layout with Theoretical Guarantees. *In : International Workshop on Super Visualization (IWSV'08)*. – Kos, Greece, mai 2008.