Social network ordering to reduce cache misses

Thomas Messi Nguélé, Maurice Tchuente, Jean-François Méhaut

Numérotation des graphes sociaux

pour la réduction des défauts de cache

Thomas Messi Nguélé[1,2,3] — Maurice Tchuente[1,2] — Jean-Francois Méhaut[2,3]

[1] IRD, UMI 209 UMMISCO, Université de Yaoundé I, BP 337 Yaoundé Cameroun

[2] LIRIMA, Laboratoire d'Informatique et Applications, Fac. des Sciences, Dépt. d'informatique, BP: 812 Yaoundé, Cameroun

[3] Université de Grenoble Alpes, INRIA-LIG, Corse, BP: 38400 Grenoble, France

RÉSUMÉ. L'une des propriétés des graphes sociaux est leur structure en communautés, c'est-à-dire en sous-ensembles où les nœuds ont une forte densité de liens entre eux et une faible densité de liens avec l'extérieur. Par ailleurs, la plupart des algorithmes de fouille des réseaux sociaux comportent une exploration locale du graphe sous-jacent, ce qui amène à partir d'un nœud, à faire référence aux nœuds situés dans son voisinage. L'idée de cet article est d'exploiter la structure en communautés pour optimiser le stockage des grands graphes qui surviennent dans la fouille des réseaux sociaux. L'objectif étant de réduire le nombre de défauts de cache avec pour conséquence la réduction du temps d'exécution. Dans cet article, après avoir formalisé le problème de numérotation des nœuds des réseaux sociaux comme un problème d'arrangement linéaire optimal, nous proposons *NumBaCo*, une heuristique pour la numérotation des nœuds tenant compte de la struture en communautés des graphes sociaux. Nous présentons pour le score de Katz et le Pagerank, des simulations comparant les structures de données existantes (*bloc* et *yale*) à leurs versions exploitant *NumBaCo*. Les résultats obtenus sur une machine NUMA (de 32 cœurs) à partir des jeux de données amazon, dblp et web-google montrent que *NumBaCo* contribue à diminuer les défauts de caches et par conséquent à réduire le temps d'exécution. Par exemple, avec Katz et sur amazon, nous avons une diminution du temps d'exécution pouvant aller jusqu'à 21.6% (comparé à *bloc*) et 20.6% (comparé à *yale*); cette diminution du temps d'exécution correspond à une réduction des défauts de cache de 73%.

ABSTRACT. One of social graph's properties is the community structure, that is, subsets where nodes belonging to the same subset have a higher link density between themselves and a low link density with nodes belonging to external subsets. Futhermore, most social network mining algorithms comprise a local exploration of the underlying graph, which consists in referencing nodes in the neighborhood of a particular node. The idea of this paper is to use the community structure to optimise the storage of large graphs that arise in social network mining. The goal is to reduce cache misses and consequently, execution time. In this paper, after formalizing the problem of social network ordering as a problem of optimal linear arrangement, we propose *NumBaCo*, a heuristic for social network ordering using the community structure of social graphs. We present for Katz score and Pagerank, simulations that compare existing data structures (bloc and yale) to their corresponding versions that use *NumBaCo*. Results on a NUMA (32 cores) machine using amazon, dblp and web-google datasets show that, using *NumBaCo* allows to reduce cache misses and consequently execution time. For example with Katz and amazon dataset, we reduced execution time by up to 21.6% (compared to bloc) and 20.6% (compared to yale); this reduction of the execution time matches with the reduction by 73% of cache misses.

MOTS-CLÉS : Fouille de réseaux sociaux, Communauté, Défaut de cache

KEYWORDS : Social network mining, Community, Cache miss

1. Introduction

Lorsqu'un algorithme de fouille des réseaux sociaux (comme le score de Katz [10] ou le Pagerank [15]) s'execute, il opère sur chaque nœud x du graphe en faisant le plus souvent référence aux nœuds situés dans le voisinage de x. Les structures de données utilisées dans les langages spécialisés de graphes ou les plates-formes d'analyse des graphes (Galois [14], Green-Marl [8] ou Stinger [2, 5]) ne considèrent que le voisinage direct de x.

Dans ce rapport, nous nous intéressons à la prise en compte dans la structure de données utilisée, d'un voisinage allant au delà du voisinage direct de x, ici la communauté de x. En d'autres termes, peut-on augmenter les performances des programmes d'analyse des graphes sociaux si l'on tient compte, dans la structure de données utilisée, de l'organisation en communautés des nœuds du graphe ?

La suite de cet article comprend le background à la section 2, la section 3 présente notre approche pour résoudre le problème posé, la section 4 donne l'évaluation de l'approche, la section 5 les travaux connexes, et la section 6 la conclusion et quelques perspectives.

2. Background

Avant d'entamer la section 3, nous trouvons interessant (pour faciliter la compréhension) de présenter la structure en communautés des réseaux sociaux, la gestion de la mémoire cache et la représentation des graphes.

2.1. Structure en communautés des graphes sociaux

L'une des propriétes des graphes sociaux est leur structure en communautés. Une communauté dans un graphe social est un sous-ensemble du graphe dans lequel les nœuds ont une forte densité entre eux et une faible densité avec les nœuds de l'extérieur. La détection des communautés peut-être locale ou globale. Dans la détection locale, on cherche la communauté à laquelle appartient un noeud sans forcément connaitre tout le graphe [13]. Dans la détection globale, tout le graphe est connu et on cherche à le diviser en plusieurs communautés. Par exemple, à la figure 1, un algorithme de détection de communautés divisera le graphe en deux communautés C1 = {1, 2, 5, 7} et C2 = {3, 4, 6, 8}.

Dans cet article, nous utiliserons l'algorithme de Louvain [3] pour une détection globale des communautés. Cet algorithme recherche une partition du graphe en se basant sur une fonction de qualité appelée modularité. Celle-ci attribue à une partition une valeur comprise entre -1 et 1, en fonction de la densité de liens à l'intérieur des communautés comparée à la densité des liens à l'extérieur de la communauté.

L'algorithme de Louvain commence par une partition dans laquelle chaque nœud est une communauté. Puis, l'algorithme calcule les communautés en répétant les deux phases suivantes :

1) Pour chaque nœud i, évaluer le gain en modularité en déplaçant i de sa communauté courante vers la communauté de l'un de ses voisins. Placer i dans la communauté pour laquelle le gain est maximal (et positif).

2) Générer un nouveau graphe dans lequel les nœuds sont les communautés détectées à l'étape précédente. Revenir à la première étape avec en entrée ce nouveau graphe.

Ces deux phases sont répétées jusqu'à ce que le gain en modularité ne soit plus possible.

2.1.1. Structure imbriquée de Louvain

L'algorithme de Louvain retourne une structure hierarchique (ou imbriquée) qui est telle que les communautés obtenues à la dernier étape sont constituées des sous-communautés obtenues aux étapes précédentes. Cette structure sera utilisée dans la suite pour optimiser le stockage des nœuds.

2.2. Gestion de la mémoire cache

Un processeur qui veut accéder à une donnée pendant l'exécution d'un programme recherche d'abord l'entrée correspondante dans le cache. Si la donnée n'est pas présente, il y a défaut de cache.

Il existe trois principales catégories de défaut de cache qui sont :

– les défauts de cache obligatoires causés par la première référence à une donnée,

– les défauts de cache conflictuels causés par les données ayant la même adresse dans le cache,

– les défauts de cache capacitifs causés par le fait que les données d'un programme ne peuvent pas suffire dans le cache. C'est cette catégorie que nous considérons dans cet article.

Lorsque survient un défaut de cache, l'un des algorithmes classiques suivant est exécuté pour ramener la donnée de la mémoire dans le cache :

– algorithme optimal (la ligne de cache qui ne sera pas utilisée pour la plus grande période de temps est remplacée),

– algorithme aléatoire, – LRU *Least Recently Used*,

– FIFO *First In First Out*, – LFU *Least Frequently Used*, ...

Étant donné que les processeurs généralistes (Intel, AMD, ARM) implémentent directement l'un de ces algorithmes dans leur matériel, pour bénéficier de l'efficacité de la mémoire cache, l'utilisateur devrait s'assurer que les structures de données qu'il utilise (graphe dans notre cas) sont bien organisées pendant l'exécution du programme.

2.3. Réprésentation des graphes

Soit n le nombre de nœuds et m le nombre d'arêtes. Considérons le graphe $G1$ pondéré et non orienté de la figure 1. $G1$ a n=8 et m=11X2 (une arête est comptée double car le graphe est non-orienté).

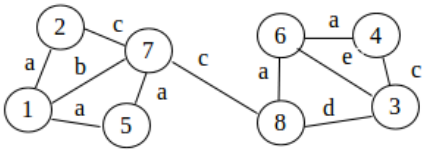


Figure 1 – Graphe $G1$

2.3.1. Représentation avec une matrice d'adjacence, $espace\ O(n^2)$

Une façon simple de représenter ce graphe est d'utiliser la représentation matricielle. Dans ce cas, on se sert d'une matrice d'adjacence M où $M(i,j) = m_{ij}$ est le poids de

l'arête entre les nœuds i et j. Mais celle-ci n'est pas très appropriée pour les graphes sociaux car les matrices résultantes sont souvent creuses. Par exemple, pour le graphe de la figure 1, on utilise 8X8 = 64 cases alors que 42 cases (66%) sont gaspillées pour stocker les zéros (voir tableau 1).

	1	2	3	4	5	6	7	8
1		a			a		b	
2	a						c	
3				c		e		d
4			c			a		
5	a						a	
6			e	a				a
7	b	c			a			c
8			d			a	c	

Tableau 1 – Matrice représentant le graphe (G)

2.3.2. Représentation sous forme de Yale, $espace\ O(n + 2m)$

La représentation souvent adoptée par certains langages spécialisés de graphe (à l'instard de Galois [14] et Green-Marl [8]) est celle de Yale [6]. Cette représentation se sert de trois vecteurs permettant de simuler la matrice d'adjacence (M pour notre exemple) :

– Un vecteur A utilisé pour stocker le poids de chacune des arêtes. Dans le tableau 2, le vecteur A est utilisé pour stocker les poids des 22 arêtes du graphe G1.

– Un autre vecteur JA donnant l'extrémité j de chacune des arêtes dont le poids est stocker dans A. De ce fait, A et JA ont la même taille (22 pour le graphe G1).

– Et un dernier vecteur IA qui donne l'indice dans A du premier élément non nul de chaque ligne de la matrice simulée (M). La dernière case contient le nombre d'arête + 1.

Par exemple, le premier élément non nul de la première ligne de M est stocké à case 1 de A ; le premier élément non nul de la deuxième ligne de M est stocké à la case 4 de A ; le premier élément non nul de la troisième ligne de M est stocké à la case 6 de A ; ... ; le premier élément non nul de la huitième ligne de M est stocké à la case 20 de A et la dernière case contient le nombre d'arêtes 22 + 1 = 23.

A	a	a	b	a	c	c	e	d	c	a	a
-	a	e	a	a	b	c	a	c	d	a	c

JA	2	5	7	1	7	4	6	8	3	6	1
-	7	3	4	8	1	2	5	8	3	6	7

IA	1	4	6	9	11	13	16	20	23

Tableau 2 – Répresentation de Yale du graphe $G1$

2.3.3. Représentation avec des listes d'adjacence, $espace\ O(n + 2m)$

D'autres représentations peuvent être utilisées : le graphe est alors représenté par un vecteur de nœuds, chaque nœud pouvant être relié

1) à un bloc de ses voisins, la taille du bloc étant variable [17] (voir figure 2 a)-)

2) à une liste chainée de blocs (de taille fixe) de ses voisins, adaptée aux graphes dynamiques et utilisée par la plateforme Stinger [2, 5] (voir figure 2 b)-).

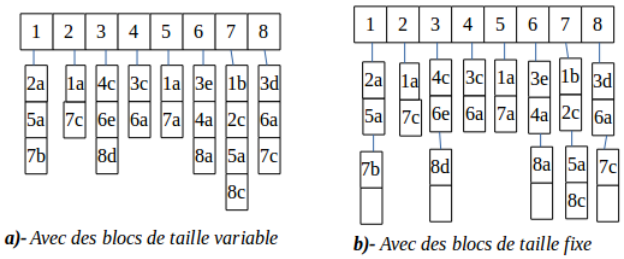


Figure 2 – Adjacency list representations of $G1$

Aucune de ces représentations ne tire profit du "regroupement en communautés" des nœuds du graphe pour réduire le temps d'exécution des algorithmes des réseaux sociaux, car ce n'était pas leur but.

3. Comment exploiter la structure en communautés des graphes sociaux pour diminuer les défauts de cache ?

3.1. Idée

L'idée est de faire en sorte que, chaque fois qu'un nœud est en cours de traitement, les autres nœuds membres de sa communauté se retrouvent dans le même cache mémoire que ce nœud. Ainsi, les données correspondant à une communauté doivent être consécutives en mémoire.

3.1.1. Structure en communautés du graphe

Le graphe est maintenant perçu comme un ensemble de communautés (détectées par l'algorithme de Louvain [3]). On procède ainsi à une renumérotation du graphe en suivant la structure imbriquée de Louvain (voir section 2.1.1) : les nœuds appartenant à une même communauté ou à une même sous-communauté ont des numéros consécutifs. Ceci aura pour effet de les rapprocher en mémoire pendant l'exécution d'un programme d'analyse des réseaux sociaux.

À la figure 3, le graphe comporte 3 communautés : C1, C2 et C3. À la partie droite, les nœuds ont des numéros consécutifs dans chaque communauté et dans chaque sous-communauté. Par exemple C3 est subdivisée en deux sous-communautés ; et les nœuds de chaque sous-communautés sont consécutifs.

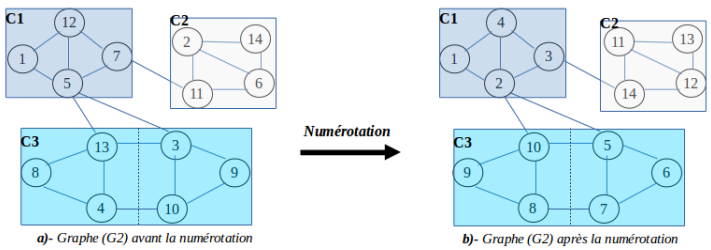


Figure 3 – Représentation de (G2) tenant compte des communautés

3.1.2. stockage du graphe

La structure en communautés est incorporée dans les modes de répresentation du graphe (rappelés à la section 2.3). Dans la nouvelle représentation obtenue, les voisins sont stockés de sorte que, lors d'un traitement, la priorité est accordée aux nœuds situés dans une même sous-communauté ou dans une même communauté.

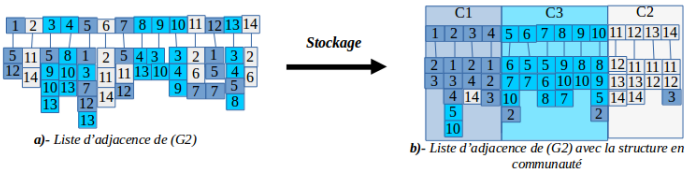


Figure 4 – Liste d'adjacence de (G2) tenant compte des communautés

La figure 4 donne une représentation par liste d'adjacence du graphe (G2) avant et après la prise en compte de la structure en communautés. On peut remarquer qu'après cette prise en compte, les nœuds appartenant à la même communauté sont maintenant plus groupés (distinction avec les couleurs).

Concernant le stockage des voisins, en considerant le nœud 10 (après numérotation), on remarque que ses voisins sont stockés dans cet ordre 8, 9, 5 et 2 : 8 et 9 sont d'abord stockés parce qu'ils sont dans la même sous-communauté que 10 (voir figure 3) ; ensuite on stocke 5 parce qu'il est dans la même communauté que 10 ; 2 est stocké en dernier parce qu'il n'est pas dans la même communauté que 10.

L'astuce utilisée pour rapprocher les nœuds en mémoire est la renumérotation du graphe en tenant compte des communautés. Dans la suite, nous la présenterons en détail après avoir montré qu'elle est une heuristique d'un problème plus général, *le problème de numérotation des graphes sociaux pour la réduction des défauts da cache*.

3.2. Formalisation du problème

3.2.1. Modélisation des défauts de cache

Notre modèle ne considère que les défauts de cache capacitifs causés par le fait que les données d'un programme ne peuvent pas suffire dans le cache mémoire (introduit à la section 2.2). On considère ici un cache de données avec une seule ligne. La Figure 5 présente notre modèle de mémoire : la mémoire principale a une taille t blocs, un bloc ayant la même taille qu'une ligne de cache.

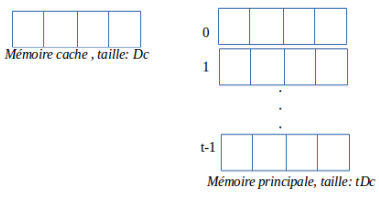


Figure 5 – Une ligne dans le cache et t blocks dans la mémoire principale

Soit

– D_c : la taille de la mémoire cache,

– N : l'ensemble des nœuds

– π : une permutation dans l'ensemble des nœuds

$$\begin{array}{rccc} \pi : & N & \longrightarrow & N \\ & x & \longmapsto & \pi(x) \end{array}$$

– b : une fonction qui donne le bloc auquel appartient un nœud en mémoire.

$$\begin{array}{rccc} b : & N & \longrightarrow & \{0,1,...,t-1\} \\ & x = aD_c + r & \longmapsto & b(x) = a \end{array}$$

– σ : la fonction qui définit un défaut de cache

$$\begin{array}{rccc} \sigma : & N \times N & \longrightarrow & \{0,1\} \\ & (x,y) & \longmapsto & \sigma(x,y) = \begin{cases} 0 & si\ b(x) - b(y) = 0 \\ 1 & sinon \end{cases} \end{array}$$

Intuitivement, le défaut de cache porte sur le nœud y : le processeur est sur le nœud x (qui se trouve donc dans le cache) et il essaye d'accéder au nœud y. Si les deux nœuds étaient dans le même bloc mémoire $b(x) = b(y)$, alors y se trouve aussi dans le cache et il n'y a pas de défaut de cache $\sigma(x,y) = 0$, sinon x et y n'étaient pas dans le même bloc mémoire $b(x) \neq b(y)$ et donc y n'est pas présent dans le cache ; il y a un défaut de cache $\sigma(x,y) = 1$.

On considère qu'un programme P dans son exécution fait reference à une séquence ordonnée de nœuds $N_k = (n_1, n_2, n_3, ..., n_k)$. Ainsi l'ensemble de défauts de caches provoqués par une séquence N_k de P est donné par :

$CM(N_k) = 1 + \sum_{i=2}^{k} \sigma(n_i, n_{i-1})$

3.2.2. Problème de numérotation des graphes sociaux

Soit $G = (N, E)$ un graphe social (N est l'ensemble des nœuds et E l'ensemble des arêtes). On aimerait trouver une numérotation (permutation) π des nœuds de ce graphe pour qu'un programme s'exécutant sur une séquence ordonnée N_k produise le minimum de défauts de cache (min_{dc}). Autrement dit,

$$Soit\ G = (N,E), min_{dc} \in \mathbb{N}, \begin{cases} - & on\ cherche & \pi : N \longrightarrow N \\ & & \quad\ n \longmapsto \pi(n) \\ - & pour\ que & \sum_i^k \sigma(\pi(n_i), \pi(n_{i-1})) \leq min_{dc}, \\ & avec\ N_k = (n_1, ..., n_k)\ et\ n_i \in N. \end{cases}$$

Théorème 3.1 (relatif à la complexité du problème) *Le problème de numérotation des graphes sociaux (PNGS) tel que défini plus haut est NP-complet.*

Démonstration :*Nous nous servons du problème d'arrangement linéaire optimal (PALO) [7]. En rappel, ce problème est défini comme suit :*

$$Soit\ G = (N,A), min \in \mathbb{N} \begin{cases} - & on\ cherche & \pi : N \longrightarrow N \\ & & \quad\ n \longmapsto \pi(n) \\ - & pour\ que & \sum_{\{n_i,n_j\} \in A} |\pi(n_i) - \pi(n_j)| \leq min \end{cases}$$

À partir de cette définition, on voit aisement que le PNGS est une instance du PALO où l'on recherche un π permettant d'obtenir le nombre minimum de défauts de cache dans un parcours des nœuds du graphe social.

3.2.3. Numérotation basée sur les communautés

Théorème 3.2 *Le problème de numérotation des graphes sociaux (PNGS) peut être résolu avec une heuristique basée sur la structure en communautés des graphes sociaux.*

Preuve: *L'algorithme 1 (NumBaCo) est une heuristique pour PNGS.*

Algorithm 1 : **Num**érotation **Ba**sée sur les **Co**mmunautés (**NumBaCo**)

Entrée : $G = (N, E)$, un graphe social
Sortie : $G' = \pi(G)$, π est un ordonnancement (permutation), le voisinage de chaque nœud x' de G' est stocké en respectant la structure imbriquée de Louvain

1: $Com \leftarrow detect\_comm\_Louvain(G)$
2: $Com_{cl} \leftarrow comm\_des\_comm(Com)$
3: $\pi \leftarrow numerotation\_graphe(Com_{cl}, G)$
4: $G_1 \leftarrow stockage\_des\_voisins(Com, G)$
5: $G' \leftarrow nouveau\_graph(G_1, \pi)$
6: **return** G'

3.2.3.1. $Com \leftarrow detect\_comm\_Louvain(G)$

Il s'agit de l'algorithme de Louvain tel que décrit plus haut (voir section 2.1). Il prend en paramètre un graphe (G) et retourne l'ensemble de communautés de (G) sous forme hierarchique ou imbriquée (voir section 2.1.1). Cet algorithme s'exécute en $O(nlogn)$, n étant le nombre de nœuds de (G). Une exécution de cet algorithme pourra donner la configuration à la figure 3a) où le graphe (G2) est divisé en 3 communautés C1, C2 et C3.

3.2.3.2. $Com_{cl} \leftarrow comm\_des\_comm(Com)$

Cet algorithme permet de classer les communautés en fonction de leur affinité (nombre d'arêtes qu'elles partagent entre elles). Il fonctionne comme l'algorithme de Louvain :

– La communauté i est placée dans la communauté voisine avec laquelle elle partage le plus grand nombre d'arêtes.

– On repète l'opération précedente jusqu'à ce qu'il ne reste qu'une seule communauté.

– Le classement des communautés initiales est donné par leur ordre d'inclusion dans la communauté finale Com_{cl}. L'opération la plus couteuse ici le calcul des affinités.

L'algorithme s'exécute alors en $O(nk + C^2)$, où k est le dégré moyen des nœuds du graphe et C le nombre de communautés contenues dans Com.

3.2.3.3. $\pi \leftarrow numerotation\_graphe(Com_{cl}, G)$

Cette fonction est utilisée pour générer une nouvelle numérotation du graphe :

– Les nœuds appartenant à la même communauté ou sous-communauté ont des numéros consécutifs

– Com_cl est utilisé pour décider quelle communauté (ou sous-communauté) vient avant l'autre en mémoire ; les numéros des nœuds des communautés qui viennent en premier ont des numéros plus petits.

Cette étape s'exécute en $O(n)$.

3.2.3.4. $G_1 \leftarrow stockage\_des\_voisins(Com, G)$

Cette fonction permet de changer l'ordre de stockage des voisins.

– Le voisinage de chacun nœud suit désormais la structure imbriquée de Louvain (voir section 2.1.1).

– Par exemple, le nœud 10 de la figure 3b) a l'ordre de stockage de ses voisins modifié à la figure 4b). Dans un stockage classique, on respecte plutôt l'ordre chronologique des nœuds.

La complexité de cette étape est de $O(nkC)$.

3.2.3.5. $G' \leftarrow nouveau\_graph(G_1, \pi)$

Ici, le nouveau graphe est généré avec une nouvelle numérotation et avec un stockage des voisins qui respecte la structure imbriquée de Louvain. La complexité ici est de $O(nk)$.

Ainsi la complexité totale de l'algorithme est de $O(nlogn + (nk + C^2) + n + nkC + nk) = O(nlogn + n(K(2 + C) + 1) + C^2)$.

3.2.4. Quelques propriétés : gain dû à la numérotation

Étant donné un programme P d'analyse des réseaux sociaux, le gain désigne ici la réduction du nombre de défauts de cache provoquée par l'exécution de P lorsque les nœuds ont été stockés en mémoire en suivant l'algorithme de numérotation présenté plus haut.

Propriéte 3.1 (Exploration des nœuds) *Le gain obtenu par cette numérotation depend de l'exploration des noeuds du graphe pendant l'exécution du programme :*

1) Il est grand pour une exploration locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés dans son voisinage).

2) Il est petit pour une exploration non locale du graphe (à partir d'un nœud, on fait réference aux nœuds situés en dehors de son voisinage).

Indications : *Dans le premier cas, comme les nœuds successifs sont plus rapprochés en mémoire, on a moins de défauts de cache ; le gain est donc grand.*

Dans le deuxième cas, les nœuds successifs étant éloignés en mémoire, il y a plus de défauts de cache ; le gain est donc plus petit.

Propriéte 3.2 (Rangement initial des nœuds) *Le gain obtenu par cette numérotation depend du rangement initial (avant l'usage de NumBaCo) des noeuds dans le graphe :*

1) Il est grand lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros non consécutifs et très éloignés.

2) Il est petit lorsque les noeuds inter-communautaires (de chacune des communautés) ont des numéros consécutifs (gain nul) ou proche (gain petit).

Indications : *Dans le premier cas, le rapprochement des nœuds voisins est effectif. Ainsi dans une exploration locale du graphe, il y aura moins de défauts de cache avec la numérotation générée par l'algorithme 1.*

Dans le deuxième cas, la numérotation générée par l'algorithme 1 sera presqu'identique à la numérotation initiale. Ainsi le gain sera faible.

4. Évaluation

Les expérimentations ont été menées sur la machine NUMA32, 4 nœuds (*numa node*) de 8 cœurs chacun, soit au total 32 cœurs pour 64 Go. Chaque noeud est de type Intel Xeon avec les caractéristiques 2.27GHz, L1 32KB, L2 256KB, L3 24MB, pas d'*Hyper-Threading*. La figure 6 présente un *numa node*.

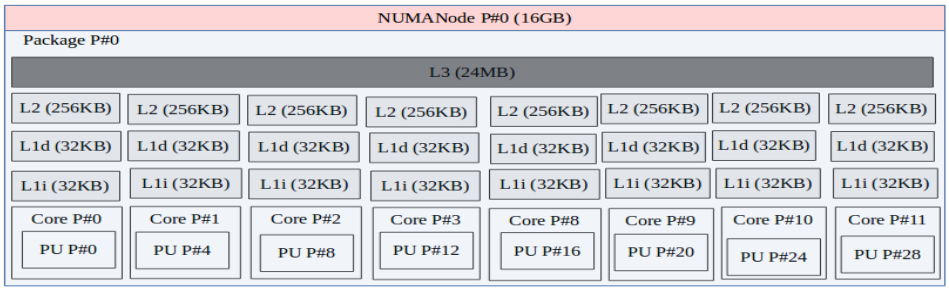


Figure 6 – *Numa node* : 8 cœurs, L1 et L2 privé, L3 partagé

Pour cette évaluation, nous avons utilisé deux applications d'analyse des réseaux sociaux : le score de Katz [10] et le Pagerank [15]. Chacune de ces applications a été implémentée avec deux structures de données pour la repésentation des graphes :

– par liste d'adjacence, chaque nœud étant relié à un bloc (de taille variable) de ses voisins : on l'appellera *bloc*

– sous forme de Yale : on l'appellera *yale*

Chacune de ces deux structures a ensuite été réorganisée en utilisant l'algorithme NumBaCo présentée à la section 3.2.3. Nous notons chacune des structures résultantes par *b_numbaco* et *y_numbaco*.

4.1. Score de katz

Le score de Katz [10] est utilisé comme une mesure de similarité basée sur les distances entre les nœuds. Le score de Katz entre deux nœuds x et y est donné par la formule :

$$katz\_score(x,y) = \sum_{l=1}^{L} (\beta^l . |paths_{x,y}^{<l>}|) \quad (1)$$

Où :

– L représente la taille maximale d'un chemin.

– $paths_{x,y}^{<l>}$ est l'ensemble des chemins de longueur l entre x et y, et $|paths_{x,y}^{<l>}|$ représente leur nombre.

– $0 < \beta < 1$. β est choisi tel que les chemins avec un l grand contribuent moins à la somme que les chemins avec un l petit.

Nous avons réalisé une implémentation multithreadée de l'algorihme de katz. Les nœuds sont rangés dans une liste chainée qui est ensuite parcourue en parallèle. Pour chaque nœud, la fonction **computeKatzNode** est invoquée (voir algorithme 2). L est responsable de la différence de temps d'exécution pour un même graphe.

Algorithm 2 Katz multi-threadé

```
1: Global nodeList, G, β, L, K_sc
2: do_work()
3: while nodeList ≠ ∅ do
4:     x ← atomic_dequeue(nodeList)
5:     K_sc[x] ← computeKatzNode(x, G, β, L)
6: end while
7:
8: main()
9: nodeList ← generate_nodeList(G)
10: for i = 1 to n_threads do
11:     spawn_thread(do_work())
12: end for
13: wait_every_child_thread()
14: output(K_sc)
```

Pour tout nœud x de G, on peut démontrer que les nombres de chemins à partir de ce nœud vers les autres nœuds se calculent ainsi qu'il suit ($N_i,\ et\ L_i$ représentent respectivement les voisins et les nombres de chemins d'ordre i) :

$$\begin{cases} i = 1 & N_i \quad = G.neighbors(x) \\ & L_i[y] = 1, \forall y \in N_i \\ \\ 2 \leq i \leq L & N_i \quad = \{z / z \in G.neighbors(y) \wedge y \in N_{i-1}\} \\ & L_i[z] = \sum_y L_{i-1}[y]_{/\{y \in N_{i-1} \wedge z \in G.neighbors(y)\}} \end{cases} \tag{2}$$

Ceci nous permet d'établir l'algorithme 3. La clé réside dans le calcul du vecteur de nombre de chemins $cLenPath$ à la ligne 6 avec la fonction **updateLenPath**() développée entre les lignes 15 et 26). À chaque valeur de l, avant de passer à la valeur suivante, le score de katz est mis à jour (lignes 7 à 10) ; l'ensemble des nœuds courants et le tableau des nombres de chemins sont également mis à jour (lignes 11 et 12).

Algorithm 3 Score de Katz entre x et tout nœud atteignable avec L

```
1: computeKatzNode(x, G, β, L)
2: dNeig ← G.neighbors(x)
3: pNeig ← dNeig
4: pLenPath[{dNeig}] ← 1
5: for l = 2 → L do
6:     [cNeig, cLenPath] ← updateLenPath(pNeig[ ], pLenPath[ ])
7:     for all (t ∈ cNeig) and (t ∉ dNeig) do
8:         katz[t] ← katz[t] + β^l cLenPath[t]
9:         accessibleNeig.add(t)
10:    end for
11:    [pNeig, cNeigh] ← [cNeig, empty()]
12:    [pLenPath, cLenPath] ← [cLenPath, empty()]
13: end for
14: return buildLign(x, katz[ ], accessibleNeig[ ])
15: updateLenPath(pNeig[ ], pLenPath[ ])
16: for all y ∈ pNeig do
17:    for all z ∈ G.neighbors(y) do
18:        if z ∈ cNeig then
19:            cLenPath[z] ← cLenPath[z] + pLenPath[z]
20:        else
21:            cLenPath[z] ← pLenPath[z]
22:            cNeig.add(z)
23:        end if
24:    end for
25: end for
26: return [cNeig, cLenPath]
```

4.2. Pagerank

Pagerank [15] est un algorithme utilisé par Google pour classifier les pages dans le web. Le Pagerank d'une page x est donné par la formule suvante :

$$PR(x) = (1-d) + d \sum_{y \in N_{in}(x)} \frac{PR(y)}{|N_{out}(y)|} \quad (3)$$

Où :

– d est la probabilité de suivre cette page, $(1-d)$ la probabilité de suivre une autre page.

– $N_{in}(x)$ est l'ensemble des voisins entrant de x.

– $N_{out}(y)$ est l'ensemble des voisins sortant de y.

Nous avons utilisé une implémentation *posix thread* proposée par *Nikos Katirtzis*[1]. Cette implémentation utilise la structure de données *bloc*.

4.3. Résultats

Pour cette évaluation, nous nous sommes servis des graphes amazon, dblp et web-google [19].

4.3.1. Résultats sur amazon

Les nœuds représentent les produits vendus en ligne par le site amazon. Il existe un lien entre deux produits si ceux-ci sont fréquemment achetés ensemble. Les communautés (qui seront détectées ici par l'algorithme de Louvain) peuvent être considérées comme des ensembles de produits appartenant à la même catégorie. Le graphe considéré ici (après suppression des nœuds isolés) a 269906 nœuds et 1851744 arêtes.

4.3.1.1. Diminution du temps d'exécution

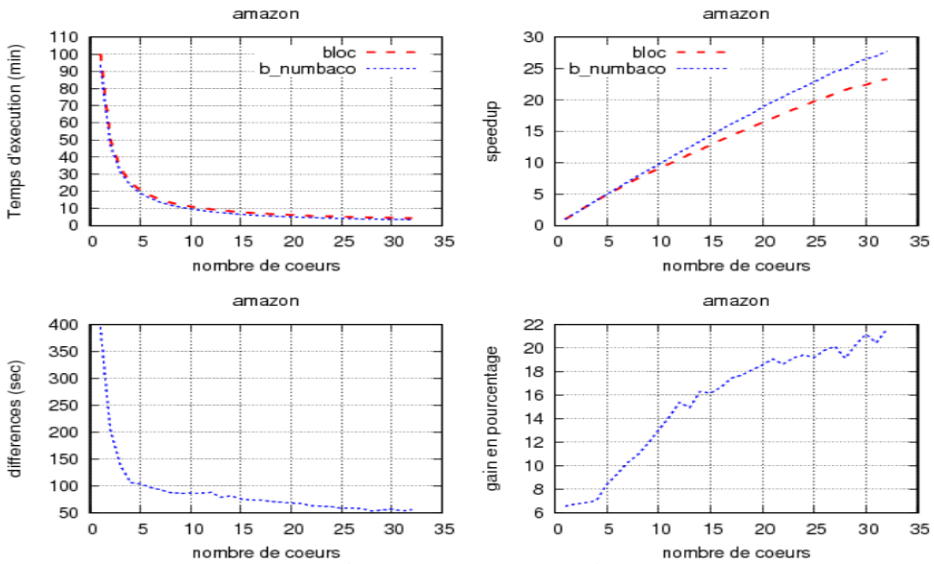
La figure 7 présente les temps d'exécution obtenus sur la machine décrite plus haut. En haut, nous avons la comparaison avec la structure *bloc* et en bas la comparaison avec la structure *yale*. Nous lisons chaque partie de la gauche vers la droite et de haut en bas. Considérons la première partie (*a-comparaison avec bloc*). Au premier cadrant, la courbe *b_numbaco* reste en dessous de la courbe *bloc*. Ceci traduit bien le fait que l'algorithme *NumBaco* contribue à réduire le temps d'exécution. Dans le troisième cadrant, on peut voir les différences de temps (en seconde) entre les deux structures. Ces différences se réduisent avec le nombre de cœurs (même variation que les temps d'exécution). Elles sont plus significatives en observant la courbe du quatrième cadrant qui présente les pourcentages des temps reduits en fonction du nombre de cœurs.

Cette courbe montre que les performances augmentent en fonction du nombre de cœurs, variant de 6.5% (avec 1 cœur) à 21.6% (avec 32 cœurs). Ceci peut s'expliquer par l'architecture utilisée (voir figure 6), plus précisement par le cache L3 qui est partagé entre les cœurs : les cœurs profitent de plus en plus des données chargées par les autres. Cette explication justifie aussi le speedup obtenu (deuxième quadrant, 32 cœurs) qui est de 27.8 avec *NumBaco* comparé à 23.3 sans *NumBaco*.

Nous faisons des observations similaires dans la deuxième partie (*b-comparaison avec yale*). Cette fois ci, *NumBaco* permet de réduire le temps d'exécution jusqu'à 20.6%. Et le speedup est de 27.7 avec *NumBaco* comparé à 23.4 sans *NumBaco*.

1. `https://github.com/nikos912000/parallel-pagerank`

a-comparaison avec bloc



b-comparaison avec yale

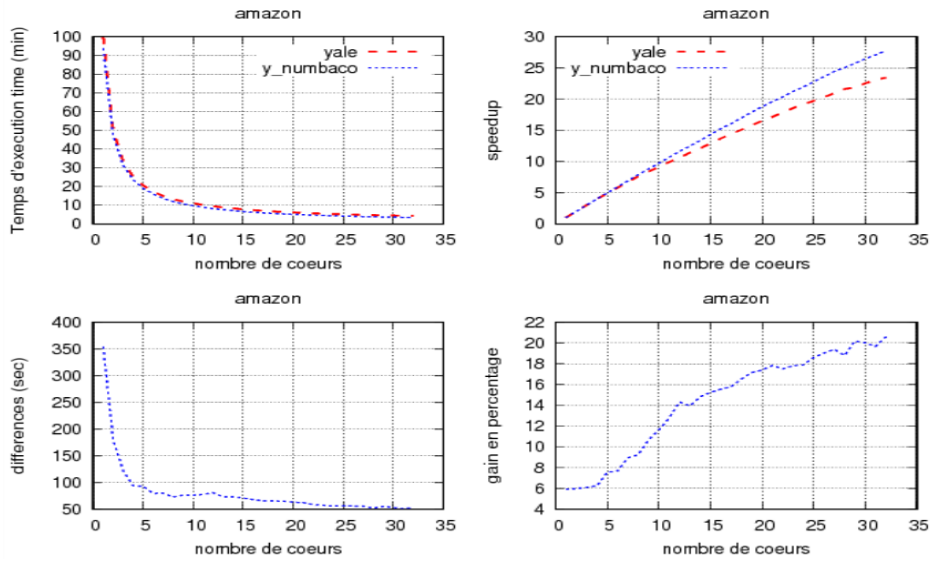
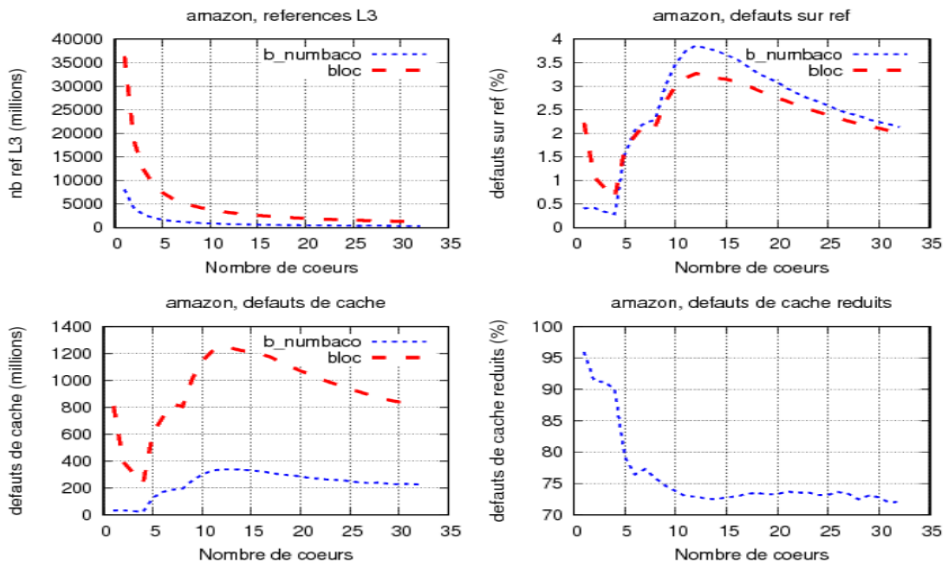


Figure 7 – Diminution du temps d'exécution – amazon, Katz

a-comparaison avec bloc



b-comparaison avec yale

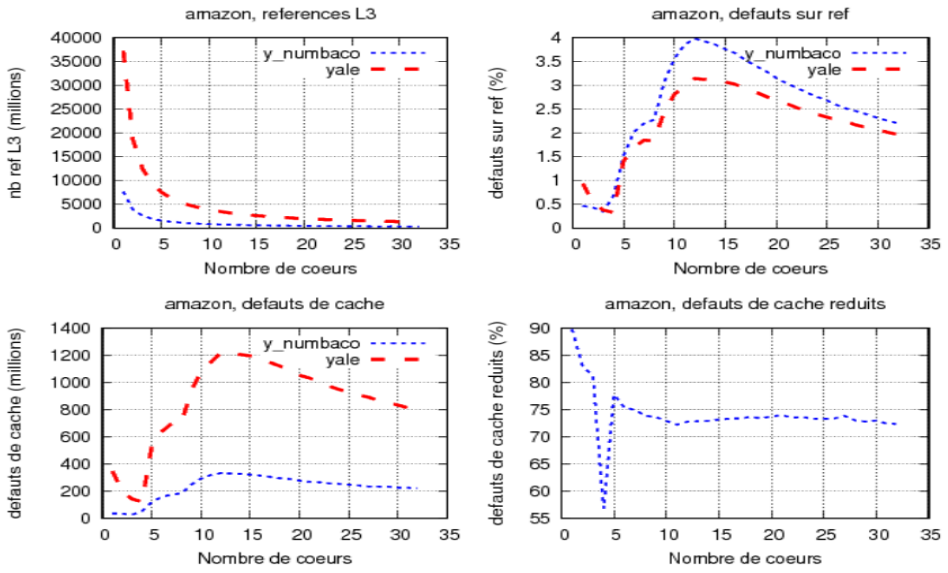


Figure 8 – Diminution des défauts de cache – amazon, Katz

4.3.1.2. Diminution des défauts de cache

Pour vérifier que les temps d'exécution observés étaient liés au nombre de défauts de caches causés par le programme, nous avons lancé le programme avec l'outil *perf* [2]. La figure 8 présente les résutats obtenus pour les évenements "cache-references" et "cache-misses".

Dans chaque partie, le premier cadrant correspond aux courbes des évènements "cache-references", le troisième cadrant correspond aux courbes de l'évènements "cache-misses".

2. `https://perf.wiki.kernel.org/index.php/Tutorial`

Le deuxième cadrant correspond au rapport (en %) entre les évènements "cache-misses" et "cache-references". Le quatrième cadrant donne le pourcentage des nombres de défauts de caches réduits l'usage de *NumBaCo*.

Considérons la première partie (*a-comparaison avec bloc*). Dans le premier cadrant, la courbe *b_numbaco* reste en dessous de la courbe *bloc*. Ceci signifie que *NumBaCo* permet de réduire le nombre de références au cache L3. En effet, les données (les nœuds) étant mieux organisées (avec la structure *b_numbaco*), les caches L2 et L1 se retrouvent plus sollicités. Ceci contribue à moins referencer le cache l3.

L'effet direct de la réduction du nombre de reférences au cache L3 est la diminution du nombre de défauts de cache observable au troisième cadrant. Ici, on observe bien que la courbe *bloc* reste au dessus de la courbe *b_numbaco* ; ce qui montre qu'en prenant en compte la structure en communautés, on réduit le nombre de défauts de cache. Le quatrième cadrant nous permet de voir qu'on réduit les defauts de caches de 95% (1 cœur) à 73% (32 cœurs).

Des observations similaires sont effectuées dans la deuxième partie (*b-comparaison avec yale*).

4.3.2. Résultats sur dblp

Les nœuds correspondent aux auteurs des articles scientifiques en informatique. Il existe un lien entre deux auteurs s'ils ont été co-auteurs d'au moins un article. Les communautés (qui seront détectées ici par l'algorithme de Louvain) correspondent aux auteurs qui ont publié dans un même journal ou dans une même conférence. Le graphe considéré ici (après suppression des nœuds isolés) a 195310 nœuds et 2099732 arêtes.

4.3.2.1. Diminution du temps d'exécution

La figure 9 présente les résultats les temps d'exécution obtenus.

Les courbes ont les mêmes allures que dans le cas précédent (cas d'amazon). La différence se fait au niveau des performances observées. Avec le jeu de données dblp, l'usage de *NumBaco* permet de réduire le temps d'exécution jusqu'à 17% par rapport à *bloc* et 15% par rapport à *yale*. Dans ce cas, le speedup à 32 cœurs est de 27.3 (comparé à 23.5 avec *bloc*) et 26.7 (comparé à 23.5 avec *yale*).

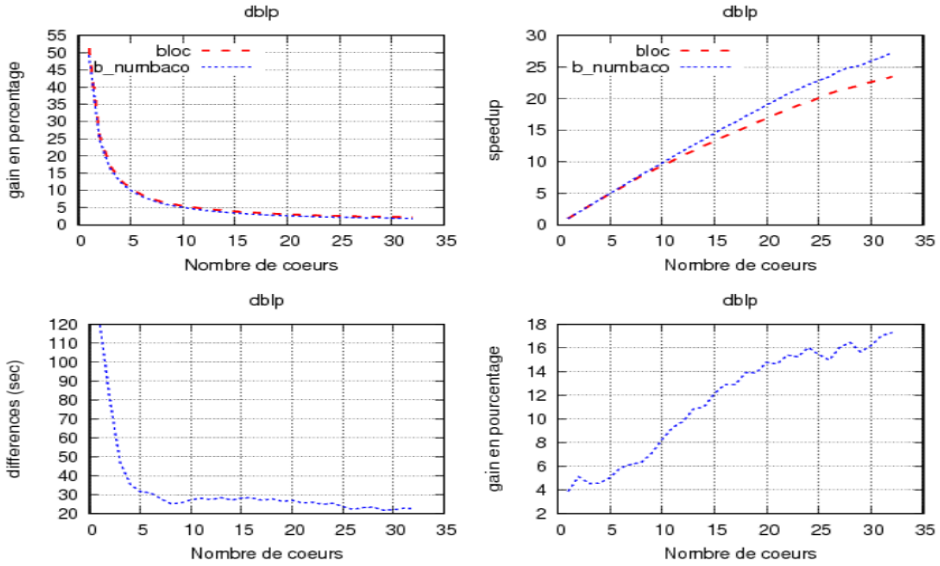
4.3.2.2. Diminution des défauts de cache

Comme dans le cas du jeu de données amazon et comme le montre la figure 10, la diminution des temps d'exécution est aussi liée à la diminution des défauts de cache. Ici à 32 cœurs, les defauts de cache sont réduits de 64% (comparé à *bloc*) et 62% (comparé à *yale*).

4.4. Résultats sur Web-goole avec Pagerank

La figure 11 montre que NumBaCo permet de rédure plus de 50% du temps d'exécute. Cette réduction du temps d'exécution est due à la reduction de près de 80% des défauts de cache que l'on peut observer à la figure 12.

a-comparaison avec bloc



b-comparaison avec yale

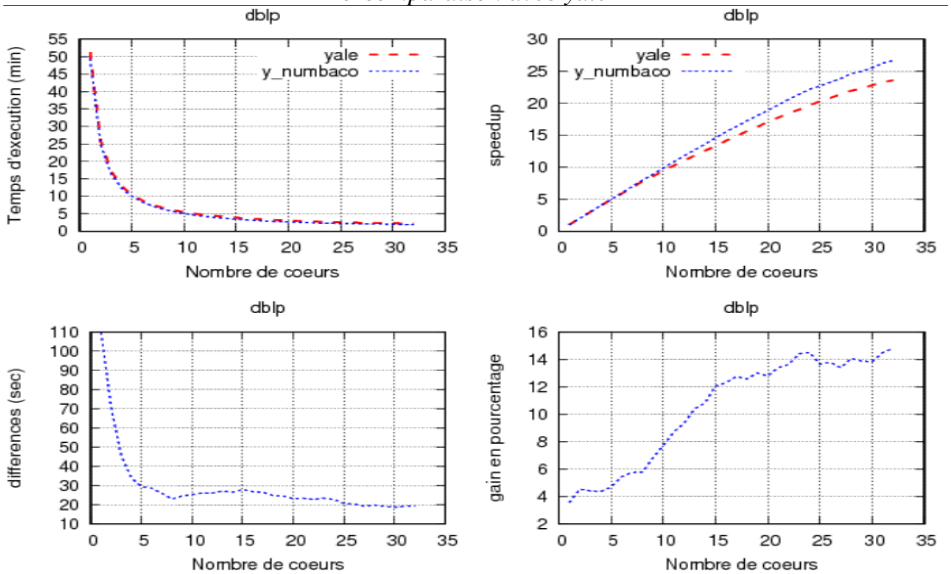
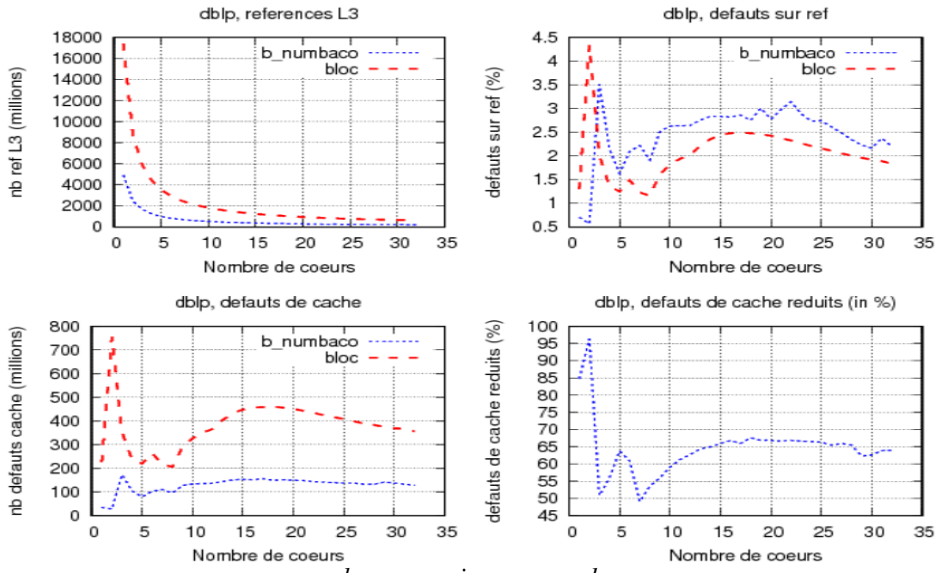


Figure 9 – Diminution du temps d'exécution – dblp, Katz

a-comparaison avec bloc



b-comparaison avec yale

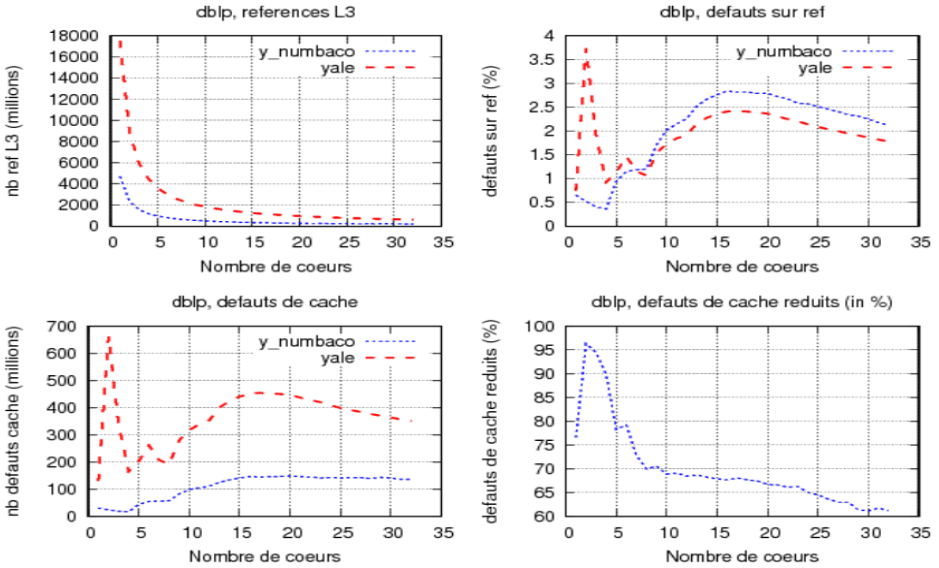


Figure 10 – Diminution des défauts de cache – dblp, Katz

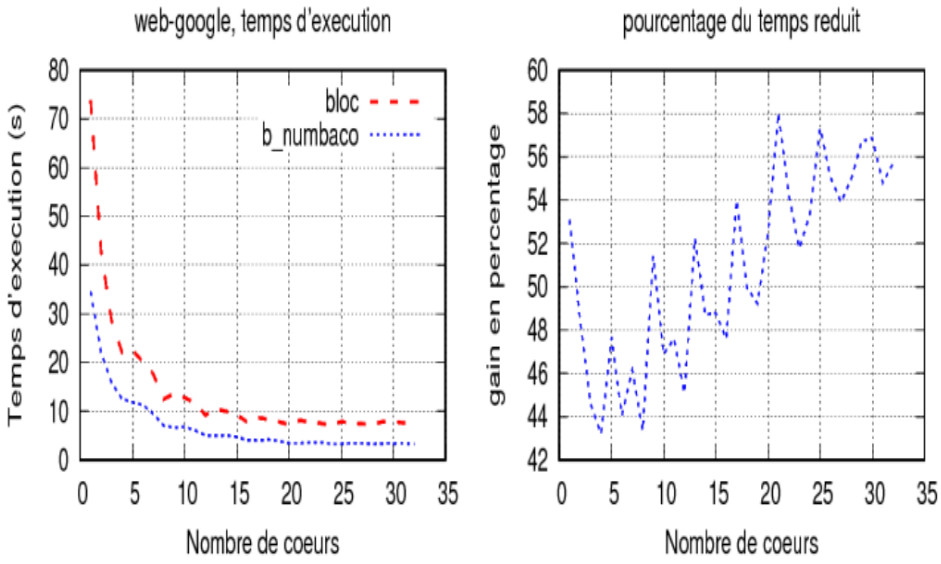


Figure 11 – Réduction du temps d'exécution – web-google, Pagerank

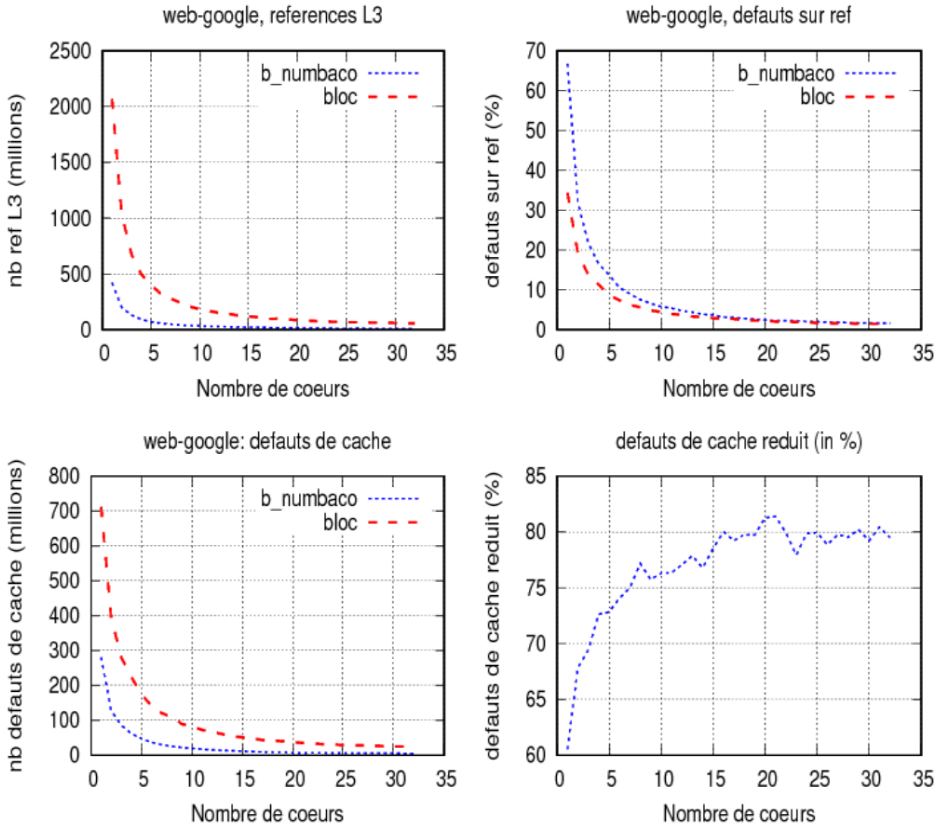


Figure 12 – Réduction des défauts de cache – web-google, Pagerank

4.5. Discussion

4.5.1. Différence de performance

La différence de performance observée entre les jeux de données amazon et dblp s'explique par le rangement initial des nœuds (Propriété 3.2) : les nœuds inter-communautaires du graphe initial d'amazon ont des numéros plus éloignés que les numéros des nœuds inter-communautaires du graphe initial de dblp. (Pour le vérifier, nous avons calculé la différence de coût $\sum_{(n_i,n_j)} |\pi(n_i) - \pi(n_j)|$ avant et après usage de l'algorithme de numérotation, cette différence était plus élévée avec le graphe amazon ; ce qui traduit un meilleur gain).

La différence de perfomance entre le Pagerank (80% des défauts de cache pour 50% du temps d'exécution) et le score de Katz (73% des défauts de cache pour 21% du temps d'exécution) s'explique l'exploration locale (Propriéte 3.1) : dans le cas du Pagerank, les nœuds successifs auxquels le programme fait reférence sont plus rapprochés en mémoire (ce qui cause moins de défauts de cache) ; et dans le cas du score de Katz, les noeuds successifs sont moins rapprochés (ce qui cause plus de défauts de cache).

4.5.2. Surcoût induit par NumBaCo

L'algorithme de numérotation présenté plus haut a pour vocation d'être exécuté avant un programme d'analyse des réseaux sociaux. Ceci afin d'avoir une organisation des nœuds en mémoire permettant de réduire au maximum les défauts de cache. Les performances du programme sont alors améliorées. Toutefois, pour atteindre cet objectif, il faudra que le programme en question ait une complexité plus élévée que l'algorithme de numérotation. Dans les expérimentations que nous proposons dans ce rapport, l'algorithme de katz a une plus grande complexité ($O(n^3)$ dans le pire des cas, comparé à $O(nlogn)$ de l'algorithme de numérotation).

Plus concrètement, en relevant le temps mis par *NumBaCo*, nous avons obtenu *1675 ms* pour amazon et *1683 ms* pour dblp (voir tableau 3). Ce qui est relativement petit par rapport au gain obtenu. Par exemple avec le jeu de données amazon (voir tableau 4), en comparant avec la structure *bloc*, on a obtenu le gain *389276 ms* sur 1 cœur, *72668 ms* sur 19 cœurs et *54252 ms* sur 32 coeurs (ce qui correspondent à environ *232 fois*, *43 fois* et *32 fois* le temps perdu).

Jeu de données	amazon	dblp
Temps (*NumBaCo*)	1675 ms	1683 ms

Tableau 3 – Temps d'exécution de l'algorithme de numérotation

Tableau 4 – Observed gain with *b_comm++*

	amazon			dblp		
nb de cœurs	1 cœur	19 cœurs	32 cœurs	1 cœur	19 cœurs	32 cœurs
sans numérotation (ms)	6020869	384892	258333	3079606	190321	131220
avec numérotation (ms)	5631593	312224	204081	2922478	167365	112695
Gain (ms)	389276	72668	54252	157128	22956	18525
(n fois coût *NumBaCo*)	***232,4 fois***	***43,4 fois***	***32,4 fois***	***93,4 fois***	***13,6 fois***	***11 fois***

L'algorithme *NumBaCo* proposé ici peut aussi être utilisé pour le pré-processing dans des applications où le même graphe est utilisé pour plusieurs exécutions. C'est le cas par exemple des *benchmarks* pour lesquels plusieurs expérimentations sont effectuées avec le même graphe. Le graphe est alors traité et stocké uniquement à la première expérimenta-

tion. Dans ce cas, le coût de *NumBaCo*, même s'il est élévé par rapport au *benchmark*, est amortit par le nombre d'expérimentations.

Une autre utilisation de *NumBaCo* peut être perçue dans les plateformes de *streaming* comme Stinger [5]. Ici, on peut imaginer que le coût est élévé à l'initialisation du système et moins élévé durant le reste de la vie du système. Nous n'avons pas étudié ce cas dans cet article, mais au regard de la théorie developpée ici, le gain généré par *NumBaCo* permettrait d'augmenter les permeformances de ces plateformes.

5. Travaux connexes

Dans de récents travaux, Junya Arai et ses co-auteurs [1] se servent d'une modification de l'algorithme de Louvain pour proposer une numérotation basée aussi sur la structure en communautés des graphes. Toutefois, comparé à leurs travaux, dans notre article (qui étend [12]), nous restons les seuls :

– à avoir proposé un classement des communautés avant la numérotation, ce qui contribue à augmenter les performances ;

– à avoir proposé un stockage des voisins en suivant l'ordre de la structure imbriquée de Louvain (voir section 2.1.1), ce qui donne la priorité aux nœuds appartenant à la même communauté ou sous-communauté, augmentant ainsi les performances ;

– à avoir montré que l'algorithme proposé est une heuristique d'un problème plus général, le problème de numérotation des graphes sociaux. Nous avons aussi montré que ce problème est NP-complet.

D'autres auteurs se sont aussi servis de la structure en communautés des graphes pour une bonne organisation des données :

- Duong et co-auteurs [4] formalisent le problème de répartition des réseaux sociaux sur un système distribué de machines. Ils proposent ensuite un algorithme de répartition qui tire profit de la structure en communautés des réseaux sociaux. Leur objectif est de réduire le nombre de requêtes à la base de données. Dans notre cas, nous recherchons un algorithme de numérotation des nœuds dans une mémoire (partagée) et permettant de réduire les défauts de cache.

- Hoque et co-auteurs [9] ont conçu une technique d'organisation du disque dur en se basant sur le regroupement en communautés des données issues des graphes sociaux. Cette technique leur a permis de diminuer le nombre de déplacements de la tête de lecture et ainsi d'améliorer l'accès au disque (48% plus rapide). Nous agissons plutôt sur la mémoire vive (tandis qu'ils agissent sur le disque dur) ; et nous cherchons à réduire le nombre de défauts de cache.

Pour améliorer le *prefetching*, Li et co-auteurs [11] utilisent l'algorithme de recherche des itemsets (fréquents fermés) pour fabriquer leurs propres algorithmes (c-miner et c-miner*). Ces algorithmes sont ensuite utilisés pour trouver la corrélation entre les blocs d'une mémoire : les blocs sont perçus comme des items, les règles d'association issues de ces items permettent de faire du *prefetching*. Dans notre cas, nous contribions aussi à améliorer le *prefetching* mais en se servant de la détection des communautés.

Plusieurs travaux visent l'usage d'une meilleure représentation des matrices creuses pour accroitre les performances de certaines applications (dans la résolution des systèmes d'équations linéaires) :

- Lukas Polok et co-auteurs [16] proposent une structure de données basée sur la représentation en sous-blocs d'une matrice creuse. Cette représentation permet de réduire

les défauts de cache lors des opérations arithmétiques effectuées sur la matrice pendant l'exécution.

- Rukhsana S. et Anila U. [18] proposent une représentation en sous-blocs (d'éléments tous non nuls) d'une matrice creuse. Les auteurs montrent que, non seulement cette représentation permet d'économiser plus d'espace, mais aussi permet d'obtenir une meilleur perfomance (lors de la multiplication matrice-vecteur).

Dans notre cas, nous recherchons la représentation mémoire des graphes sociaux la plus appropriée pour réduire les défauts de cache des programmes d'analyse des réseaux sociaux.

6. Conclusion

Dans cet article, il était question de voir comment exploiter la structure en communautés pour diminuer les défauts de cache et le temps d'exécution des programmes de fouille des réseaux sociaux. Nous avons proposé *NumBaCo*, une numérotation des nœuds permettant de tenir compte de la struture en communautés des graphes sociaux. Des expérimentations effectuées sur le score de katz et le Pagerank avec les jeux de données amazon, dblp et web-google ont montré que les performances sont améliorées lorsqu'on utilise cette numérotation.

En perspective, nous envisageons d'intégrer cet algorithme dans les langages dédiés et les plates-formes d'analyse de graphes afin d'améliorer leurs performances. Une autre perspective est de se servir de la structure en communautés des graphes sociaux lors de l'optimisation des boucles (pendant la phase de compilation) ; ceci pourrait contribuer à augmenter les performances.

Références

[1] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order : Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 22–31. IEEE, 2016.

[2] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger : Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics : Theory and Experiment*, 2008(10) :P10008, 2008.

[4] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. Sharding social networks. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 223–232. ACM, 2013.

[5] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. Stinger : High performance data structure for streaming graphs. *HPEC*, pages 1–5, 2012.

[6] Stanley C Eisenstat, MC Gursky, MH Schultz, and AH Sherman. Yale sparse matrix package. i. the symmetric codes. Technical report, DTIC Document, 1977.

[7] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3) :237–267, 1976.

[8] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl : a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[9] Imranul Hoque and Indranil Gupta. Social network-aware disk management. 2010.

[10] Leo Katz. A new status index derived from sociometric analysis. *Psychometrica-vol.18, No.1*, 18(1), March 1953.

[11] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2) :213–245, 2005.

[12] Thomas Messi Nguélé, Maurice Tchuenté, and Jean-François Mehaut. Exploitation de la structure en communautés pour la réduction de défauts de cache dans la fouille des réseaux sociaux. In *Conférence de Recherche en Informatique (CRI)*, Yaoundé, Cameroon, December 2015.

[13] Blaise Ngonmang, Maurice Tchuente, and Emmanuel Viennet. Local community identification in social networks. *Parallel Processing Letters*, 22(01) :1240004, 2012.

[14] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[15] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : bringing order to the web. 1999.

[16] Lukas Polok, Viorela Ila, and Pavel Smrz. Cache efficient implementation for block matrix operations. In *Proceedings of the High Performance Computing Symposium*, page 4. Society for Computer Simulation International, 2013.

[17] Jason Riedy, David A. Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. *IEEE Computer Society Washington, DC, USA 2012*, 18(1) :1619–1628, 2012. IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

[18] Rukhsana Shahnaz and Anila Usman. Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers. *Int. Arab J. Inf. Technol.*, 8(2) :130–136, 2011.

[19] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1) :181–213, 2015.