Un partitionnement d'arêtes à base de blocs pour les algorithmes de marches aléatoires dans les grands graphes sociaux

Yifan Li [1], **Camelia Constantin** [1], **Cédric du Mouza** [2]

1. *Univ. Pierre et Marie Curie, 2 Place Jussieu, 75005 Paris, France*
   yifan.li@lip6.fr,camelia.constantin@lip6.fr
2. *CNAM Paris, 2 rue Conté, 75141 Paris, France*
   dumouza@cnam.fr

RÉSUMÉ. *Des résultats récents (Bourse* et al.*, 2014 ; Gonzalez* et al.*, 2012 ; Xin* et al.*, 2013) montrent que le* partitionnement des arêtes *(vertex-cut) s'avère être plus efficace que le* partitionnement des sommets *(edge-cut) traditionnellement utilisé pour les calculs sur des graphes réels comme les graphes des réseaux sociaux. Compte tenu de la distribution de type « power-law » de la plupart des graphes, le partitionnement des arêtes (vertex-cut) permet d'éviter des calculs asymétriques. Par conséquent, plusieurs systèmes de calcul de graphe basés sur cette approche ont été proposés, tels que PowerGraph (GraphLab2) (Gonzalez* et al.*, 2012) et GraphX (Xin* et al.*, 2013). Leurs stratégies de partitionnement sont génériques et ne dépendent pas des algorithmes utilisés pour effectuer les différents calculs. Nous proposons dans cet article une nouvelle stratégie de partitionnement de graphe (de ses arêtes) optimisant l'exécution des algorithmes basés sur les marches aléatoires qui prennent en compte les propriétés topologiques des graphes. Notre stratégie est basée sur la construction de blocs qui modélisent des* communautés locales*. Notre approche de partitionnement en blocs fournit une distribution équilibrée des arêtes sur les noeuds de calcul et permet de la maintenir dynamiquement. Les coûts de communication pour les calculs utilisant des marches aléatoires sont réduits de manière significative par rapport aux approches existantes.*

ABSTRACT. *Recent results (Bourse* et al.*, 2014 ; Gonzalez* et al.*, 2012 ; Xin* et al.*, 2013) prove that* edge partitioning *approaches (also known as* vertex-cut*) outperform* vertex partitioning *(*edge-cut*) approaches for computations on large and skewed graphs like* social networks*. These vertex-cut approaches generally avoid unbalanced computation due to the* power-law degree distribution *of the graphs. However, these methods, like* evenly random assigning *(Xin* et al.*, 2013) or* greedy assignment strategy *(Gonzalez* et al.*, 2012), are generic and do not consider any computation pattern for specific graph algorithm. We propose in this paper a vertex-cut partitioning dedicated to random walks algorithms which takes advantage of graph topological properties. It relies on a blocks approach which captures* local communities*. Our split and merge algorithms allow to achieve load balancing of the workers and to maintain*

it dynamically. Our experiments illustrate the benefit of our partitioning since it significantly reduce the communication cost *when performing* random walks*-based algorithms compared with existing approaches.*

MOTS-CLÉS : *partitionnement de graphes, réseaux sociaux, performance.*

KEYWORDS: *graph partitioning, social networks, performance.*

DOI:10.3166/ISI.22.3.89-113

1. Introduction

Les algorithmes basés sur les marches aléatoires, tels que PageRank Personnalisé (PPR) (Jeh, Widom, 2003) et SALSA Personnalisé (Bahmani *et al*., 2010) se sont avérés être particulièrement efficaces pour les systèmes de recommandation personnalisés en raison de leur capacité à passer à l'échelle. Certaines propositions récentes s'appuient sur plusieurs marches aléatoires commencées à partir de *chaque sommet* du graphe, comme par exemple l'algorithme *Fully Personalized PageRank* qui utilise l'approximation de Monte Carlo (Bahmani *et al*., 2011). Nous appelons ce calcul intensif *Fully Multiple Random Walks* (FMRW).

Le partitionnement de graphe est d'un intérêt primordial pour le traitement distribué des graphes. Il joue un rôle de plus en plus important pour les calculs centrés sur les sommets, comme dans le modèle *Pregel* qui s'est largement imposé comme le modèle de référence de calculs centrés sommets, et dans l'évaluation de requêtes. Les résultats récents montrent que le partitionnement des arêtes (*vertex-cut*) s'avère être plus efficace (Bourse *et al*., 2014 ; Gonzalez *et al*., 2012) que le partitionnement des sommets (*edge-cut*) pour les calculs sur des graphes réels comme les graphes des réseaux sociaux. Par conséquent, plusieurs systèmes de calcul sur des graphes basés sur cette approche ont été proposés, tels que PowerGraph (GraphLab2) (Gonzalez *et al*., 2012) et GraphX (Xin *et al*., 2013). Toutefois, pour la plupart de ces systèmes, les stratégies de partitionnement sont génériques et indépendantes des algorithmes de calcul qui les utilisent. Les stratégies de partitionnement distribuent les arêtes, soit uniformément entre les partitions, soit de manière aléatoire, en utilisant une fonction de hachage sur les identifiants des sommets (comme dans Giraph(Apache, s. d.) et Graphx) ou un algorithme glouton ou dynamique (comme dans PowerGraph et GPS(Salihoglu, Widom, 2013)). Ce partitionnement peut générer une distribution déséquilibrée de la charge de calcul entre les ressources.

De plus, contrairement à des algorithmes à messages « légers » comme PageRank dont les messages transmis entre les sommets ne sont que des valeurs représentant des scores, les algorithmes tels que *Fully (multiple) Random Walks* considérés dans cet article ont un coût de communication plus important puisque (i) certaines informations supplémentaires sur les chemins doivent également être transmises (ensemble des chemins avec leur longueur et score par exemple) et (ii) plus d'un message (marche aléatoire) partent de chaque sommet à chaque itération. Dans ce cas, la réduction des coûts de communication est cruciale pour garantir les performances du calcul. Ceci

est particulièrement vrai dans un cluster informatique à large échelle reposant sur une infrastructure réseau machine à machine complexe, ou les systèmes distribués à calcul en mémoire comme Spark où la communication pourrait être un goulot d'étranglement pour atteindre une performance globale élevée.

Nous proposons dans cet article une nouvelle technique de partitionnement de graphe basée sur des blocs prenant en compte la charge de calcul et qui offre une répartition équilibrée et une réduction des coûts de communication pour les calculs basés sur des marches aléatoires. À notre connaissance, c'est la première fois qu'une stratégie de partitionnement dédiée aux marches aléatoires multiples est proposée dans le modèle Pregel. Enfin, les expériences montrent que notre partitionnement apporte des gains significatifs en termes de coût de communication et de temps d'exécution.

Contributions

En résumé, les contributions de cet article sont les suivantes :

– une stratégie de partitionnement s'appuyant sur la notion de blocs qui prend en compte les spécificités des algorithmes de calcul sur des graphes et les propriétés topologiques des grands graphes du monde réel, ainsi qu'un algorithme de sélection des *graines* pour la construction des blocs ;

– des algorithmes pour fusionner et diviser les blocs afin de parvenir à un équilibrage dynamique des partitions ;

– une comparaison expérimentale sur de grands graphes réels des réseaux sociaux de notre approche de partitionnement avec plusieurs méthodes de partitionnement aléatoire existantes.

2. État de l'art

Pregel (Malewicz *et al*., 2009) est devenu un *framework* distribué populaire de calcul de graphe en raison des facilités qu'il offre aux développeurs pour réaliser des calculs sur les grands graphes comparativement à d'autres systèmes de calcul de données parallèles, par exemple Hadoop. Pregel s'inspire du modèle de calcul *Bulk Synchronous Parallel* (Valiant, 2008), un modèle où les calculs sur un graphe sont réalisés en plusieurs itérations, également appelées *super-steps*. Au cours d'une itération, chaque sommet reçoit d'abord tous les messages qui lui ont été adressés par d'autres sommets pendant l'itération précédente. Chaque sommet effectue ensuite en parallèle les actions définies par une fonction spécifiée par l'utilisateur, à savoir *Vertex.compute ()* (Salihoglu, Widom, 2013) ou *Vertex.program ()* (Gonzalez *et al*., 2012), en utilisant les nouvelles valeurs reçues dans les messages. Puis chaque sommet peut décider d'arrêter le calcul ou d'envoyer à d'autres sommets les messages à utiliser dans la prochaine itération. Lorsqu'aucun message n'est transmis dans le graphe pendant une itération (*c'est-à-dire* chaque sommet a décidé de s'arrêter) le calcul s'arrête. En raison du succès du modèle de calcul de *Pregel*, plusieurs optimisations ont été récemment proposées dans la littérature comme par exemple la fonction

Master.compute () (Salihoglu, Widom, 2013) pour incorporer des calculs globaux ou *Mirror Vertices* (Low *et al*., 2012) pour réduire la communication.

Les méthodes de partitionnement de graphes traditionnelles, par exemple *2-way cut* par recherche locale ou les approches multi-niveaux, comme Kernighan-Lin (Kernighan, Lin, 1970), PageRank Vectors (Andersen *et al*., 2006) et les algorithmes basés sur METIS (Karypis, Kumar, 1998), suivent une stratégie de partitionnement de sommets (*edge-cut*). Ils proposent des partitions qui attribuent (presque) uniformément les sommets entre les partitions tout en minimisant le nombre d'arêtes coupées (arêtes entre deux partitions). Ces algorithmes sont efficaces pour les petits graphes, qui ne sont pas sensibles au déséquilibre de la charge des calculs. Cependant ceci conduit à une charge déséquilibrée sur les partitions pour des graphes du monde réel, compte tenu de leur grande taille et de leur distribution suivant une loi de puissance. Des propositions de partitionnement plus récentes dans des systèmes semblables à Pregel, comme Giraph, GPS, Gelly et Chaos (Roy *et al*., 2015), coupent le graphe en utilisant une stratégie *edge-cut* qui génère également un déséquilibre pour les graphes dont la distribution suit une loi de puissance, comme cela a été montré dans (Gonzalez *et al*., 2012).

À la différence du partitionnement de sommets pour lequel il existe une importante littérature et plusieurs implémentations, peu de travaux récents proposent un partitionnement d'arêtes. Les deux principales propositions sont GraphX (Xin *et al*., 2013) et PowerGraph (Gonzalez *et al*., 2012). Cependant GraphX propose seulement un partitionnement aléatoire/par hachage où les arêtes sont réparties uniformément entre les partitions en prenant en compte certaines contraintes de communication entre les sommets. Certaines propriétés du graphe sous-jacent, comme par exemple les *communautés locales* dans les réseaux sociaux, ne sont pas prises en considération. Contrairement au partitionnement par hachage, l'approche proposée par PowerGraph s'appuie sur une heuristique gloutonne, *Greedy Vertex-Cuts*, qui a montré des performances significativement meilleures que le placement aléatoire dans tous les cas (Gonzalez *et al*., 2012). Cependant cette approche ignore également les propriétés topologiques du graphe et se concentre uniquement sur la minimisation de la communication lors de la distribution des arêtes entre les partitions. De plus, contrairement à notre proposition, les partitions obtenues par GraphX et par PowerGraph ne peuvent pas être mises à jour dynamiquement lorsque le graphe évolue.

Notre approche bénéficie également de l'existence de communautés dans les graphes sociaux. Pour des graphes sociaux avec des degrés de noeuds qui suivent des distributions de type *heavy-tailed*, ainsi que des grands coefficients de regroupement, les voisins directs d'un sommet suffisent pour construire de bons regroupements (communautés) avec une faible conductance (Gleich, Seshadhri, 2012). Une amélioration de cette méthode de détection de communautés est proposée par (Whang *et al*., 2013), mais cette proposition ne prend pas en compte le partitionnement des arêtes ou l'équilibrage de la charge. En outre, le partitionnement de graphe lorsqu'il existe des communautés qui se recouvrent n'est pas adapté aux systèmes similaires à Pregel.

3. Partitionnement de graphe basé sur des blocs

Cette section introduit d'abord l'idée principale de notre stratégie de partitionnement de graphe basée sur des blocs, puis donne des détails sur la construction des partitions.

3.1. Principes

La plupart des méthodes existantes de partitionnement d'arêtes, comme par exemple les approches aléatoires (Xin *et al.*, 2013) ou gloutonnes (Gonzalez *et al.*, 2012), permettent d'obtenir une charge de travail équilibrée, ce qui signifie que chaque partition a le même nombre d'arêtes. Notre objectif est d'aller au-delà de l'équilibrage de la charge de travail et de réduire le temps de traitement des calculs sur le graphe en réduisant la communication entre les partitions. Dans le cadre d'une approche de *partitionnement par arêtes* un sommet est éventuellement attribué à plusieurs partitions et les communications entre les partitions se produisent lors de la mise à jour des différentes répliques (sommets « miroirs ») à chaque super-étape de Pregel. Par conséquent, le facteur de réplication des sommets (*Vertex Replication Factor (VRF)*) défini en premier dans (Gonzalez *et al.*, 2012) est souvent utilisé pour mesurer le coût de communication. Ainsi, étant donné un partitionnement des arêtes, le coût de la communication est généralement estimé, dans les approches similaires à celles de Pregel, comme étant :

DÉFINITION 1 (Coût de communication). — *Le coût de communication pour un calcul de graphe avec une approche Pregel est défini comme étant :*

$$cost_{Comm} = O(L * (VRF * |V|)) \quad (1)$$

où L désigne le nombre de super-étapes (itérations) lors du calcul sur le graphe, V est l'ensemble des sommets et VRF est le facteur de réplication des sommets.

Cependant, dans la plupart des graphes réels, comme les réseaux sociaux, il existe de nombreux *clusters* (communautés). Notre objectif est de considérer cette caractéristique topologique dans notre approche de construction de blocs. Le *modèle d'accès local (Local Access Pattern (LAP))* est décrit par (Yang *et al.*, 2012) étant comme l'un des trois types de motifs de requêtage dans les graphes. Notre stratégie de partitionnement pour des algorithmes basés sur les marches aléatoires s'appuie sur ce modèle et considère les communautés des graphes pour réduire les coûts de communication.

Bien que *VRF* soit un bon estimateur du coût de la communication pour certains algorithmes de calcul sur des graphes, il ne convient pas aux algorithmes basés sur les marches aléatoires qui suivent un modèle *LAP*, car pour ces algorithmes le nombre de visites de chaque sommet est différent au cours d'une même super-étape. En d'autres termes, les communications sont menées de façon inégale dans le graphe. Notre objectif est donc de concevoir une nouvelle stratégie de partitionnement des arêtes dédiée à des algorithmes basés sur les marches aléatoires et qui prend en compte à la fois la

topologie donnée par la distribution en loi de puissance du graphe et le modèle *LAP* qui caractérise ces algorithmes.

Notre approche

Un bloc correspond à un cluster bien connecté du graphe, par exemple une communauté dans un réseau social. Dans l'approche *Pregel*, nous considérons le bloc comme étant un ensemble d'arêtes qui sont "proches" l'une de l'autre et ces blocs deviennent les unités de base utilisées dans la construction de chaque partition, mais aussi les unités d'allocation pour la charge de travail sur les machines. De façon similaire à la méthodologie adoptée dans le partitionnement de sommets dans (Gleich, Seshadhri, 2012 ; Whang *et al.*, 2013), nous proposons de calculer un ensemble de K blocks en explorant le graphe. Une arête est attribuée à un bloc en fonction de sa distance par rapport à ce bloc. Nous commençons une exploration en largeur (*breadth-first search (BFS)*) à partir d'un ensemble prédéfini de K sommets respectant certaines propriétés, appelés graines. Pour chaque arête rencontrée, nous mettons à jour sa distance par rapport à tous les blocs. Lorsque la phase d'exploration s'est terminée, nous attribuons les arêtes au bloc le plus proche.

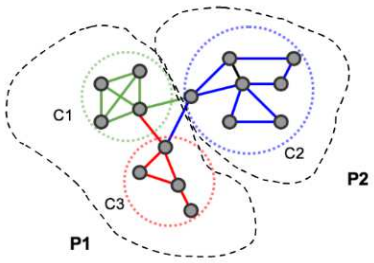

Figure 1. Exemple de blocs et de partitions

EXEMPLE 2. — La figure 1 présente un exemple qui illustre notre approche à deux étapes. Tout d'abord, nous regroupons les arêtes en fonction de leur distance des différentes graines (ici trois graines). Nous obtenons ainsi trois communautés (blocs d'arêtes): c_1, c_2 et c_3. Ensuite, nous fusionnons des blocs pour obtenir des partitions de taille similaire. Ainsi, dans cet exemple, nous construisons la partition P_1, composée des blocs c_1 et c_3, et la partition P_2 qui correspond au bloc unique c_2. □

3.2. Distance d'une arête

Le calcul des mesures de proximité entre deux sommets a été étudiée dans de nombreux travaux. Ces mesures de proximité permettent de détecter des clusters dans le graphe (voir la section 3.3). Plusieurs approches ont étendu la détection de clusters dans des graphes pour effectuer des partitionnements en se basant sur l'observation que pour plusieurs algorithmes de graphes comme ceux reposant sur des marches aléatoires, la recherche des plus proches voisins, l'exploration en largeur (BFS), etc., les communications lors des calculs se produisent principalement entre des sommets

appartenant au même cluster. Par exemple (Andersen *et al.*, 2006) a proposé une méthode basée sur l'utilisation de vecteurs PageRank par rapport à un sommet initial et plusieurs configurations pré-établies afin de trouver une « bonne » partition. La construction des partitions en utilisant des marches aléatoires est décrite dans (Sarkar, Moore, 2010).

Pour notre approche de partitionnement, nous proposons ici d'estimer la distance entre une arête et un sommet. Nous adaptons le score P-distance inverse (Jeh, Widom, 2003) (inverse P-distance) utilisé pour le calcul de distance entre deux sommets.

3.2.1. Distance sommet-à-sommet

La mesure P-distance inverse capture la connectivité : deux sommets d'un graphe sont proches s'ils sont reliés par de nombreux chemins qui sont courts. Ainsi, la distance $dist_v(i,j)$ entre le sommet *i* et le sommet *j* dans un graphe G dirigé peut être calculée en prenant en compte les chemins qui les relient, de la manière suivante :

$$dist_v(i,j) = \sum_{p \in P_{ij}} S(p) \tag{2}$$

où P_{ij} indique l'ensemble des chemins de *i* à *j*, *S(p)* est la valeur de la P-distance inverse correspondante au chemin *p*, définie ci-dessous.

Similairement à l'idée de distance P-distance inverse, nous introduisons le concept d'accessibilité dans le calcul de distance entre deux sommets. L'accessibilité signifie la probabilité d'une marche aléatoire débutant au sommet i d'arriver au sommet j.

DÉFINITION 3 (Distance sommet-à-sommet). — *Pour un chemin p: v_0, v_1, ..., v_k de longueur k, la distance sommet-à-sommet S(p) est définie par :*

$$S(p) = (1-\alpha)^k \cdot \prod_{i=0}^{k-1} \frac{1}{outDeg(v_i)} \tag{3}$$

où $\alpha \in (0,1)$ est la probabilité de téléportation, c.-à.d.*, la probabilité de retourner au sommet d'origine et $outDeg(v_i)$ est le degré sortant du sommet v_i.*

3.2.2. Distance sommet-à-arête

Sur la base de la distance sommet-à-sommet présentée ci-dessus, nous définissons une distance entre un sommet et une arête comme suit :

DÉFINITION 4 (Distance sommet-à-arête). —

La distance $dist_e(a,b)$ d'un sommet a à une arête $b = (i,j)$ est définie comme étant :

$$dist_e(a,b) = \theta(dist_v(a,i), dist_v(a,j))$$

où θ est une fonction d'agrégation qui dépend de la distance entre le sommet a et le sommet i ainsi que de la distance entre le sommet a et le sommet j.

Dans nos expériences nous choisissons la fonction θ comme étant la *moyenne*, mais d'autres fonctions comme min ou max peuvent également être considérées.

3.3. Algorithme d'allocation des arêtes

Nous pouvons maintenant concevoir un algorithme d'allocation d'arêtes basé sur la distance sommet-à-arête définie précédemment. Notre algorithme peut être décomposé en trois étapes :

$i)$ sélection d'un sous-ensemble de sommets, appellées par la suite des *graines* ;

$ii)$ calcul de distance de chaque arête à toutes les graines ;

$iii)$ allocation des arêtes aux différents blocs.

3.3.1. Sélection des graines

Nous considérons pour notre partitionnement en blocs une stratégie d'expansion à partir de graines : nous sélectionnons un sommet comme graine pour chaque bloc puis nous ajoutons chaque arête à l'un des blocs existants. Naturellement le résultat du partitionnement, en termes d'équilibrage de taille ou de communication pendant le calcul, dépend fortement du choix des graines. Ce problème a été étudié dans la littérature par exemple dans (Whang *et al*., 2013) qui propose un algorithme pour détecter les communautés dans des graphes ou dans (Dahimene *et al*., 2014) où plusieurs stratégies de sélection de graines sont proposées pendant l'étape de pré-calcul de l'algorithme de recommandation.

Dans cet article les graines sont sélectionnées suivant la méthode *Spread Hubs* (voir (Whang *et al*., 2013)) facilement déployable dans les systèmes existants de calcul sur des graphes. Nous utilisons deux mesures principales pour la sélection des graines : $1)$ le degré du sommet pour le choisir comme étant une graine candidate et $2)$ sa distance à d'autres graines existantes. Notre algorithme de sélection de graines considère en priorité les sommets avec un degré global important, conformément aux étapes suivantes:

$i)$ nous trions tout d'abord les sommets du graphe par ordre décroissant de leurs degrés globaux (degré entrant + degré sortant) ;

$ii)$ pour chaque sommet présent dans la liste des sommets triée, s'il est *trop proche* de n'importe quelle autre graine existante il est ignoré et le sommet suivant et pris en considération.

Notre algorithme de sélection de graines (voir l'Algorithme 1) considère en priorité les sommets avec un degré global élevé car dans ce cas son score de centralité est élevé. Les sommets de son voisinage direct sont susceptibles de rejoindre son bloc. En prenant en compte une distance minimale entre les graines nous assurons une meilleure répartition des graines dans le graphe. Nous utilisons l'algorithme BFS pour

Algorithme 1 : sélection des graines

entrée : un graphe dirigé G = (V,E), nombre de graines *snum*, la profondeur de l'exploration BFS pour chaque graine *dep*
sortie : l'ensemble des graines $\{s_1, s_2, ..., s_{snum}\}$

```
Initialisation:
Pour chaque u ∈V, C[(u, deg, tag)] ← (u, u.getDegree(), true);
C.sortBy(C.deg, descending);
ensemble des graines: S.empty[VertexId];
pour i ←0 à C.length faire
    si S.size < snum alors
        si C[i].tag alors
            S.add(C[i].u.id);
            pour chaque q ∈ C[i].BFS(dep) faire
                q.tag = false;
        sinon
            break;
    sinon
        break;
Retourner S;
```

mesurer la *distance* entre les graines car il est efficacement implémenté dans Pregel. Pour chaque graine candidate, l'algorithme BFS qui débute avec cette graine calcule le nombre de sauts (itérations) requis pour rencontrer d'autres graines existantes. Les expériences montrent que cet algorithme permet d'obtenir un bon partitionnement, y compris lorsque la profondeur de l'exploration BFS de chaque cluster est fixée à 1 (c'est-à-dire lorsqu'on ne peut pas choisir comme nouvelle graine un voisin direct d'une graine existante).

3.3.2. Nombre de graines

Chaque graine détermine un bloc, par conséquent le nombre total de graines doit être supérieur ou égal au nombre de partitions finales souhaitées. Cependant, nous pouvons obtenir un meilleur partitionnement lorsque le nombre de graines est plus élevé que le nombre final de partitions pour les raisons suivantes :

– *l'expansion* de chaque bloc peut être traitée de manière indépendante, elle peut par conséquent être déployée facilement sur une architecture de type Pregel ;

– la fusion de petits blocs est beaucoup moins coûteuse que le fractionnement (c'est-à-dire le raffinement) de gros blocs lorsqu'on essaie de minimiser le facteur de réplication ;

– niveau de réutilisation de notre partitionnement sera plus élevé lorsque le nombre de blocs pré-calculés augmente, puisqu'il pourra servir pour un nombre de

partitions différents.

3.3.3. Calcul des distances

Pendant la deuxième étape de notre algorithme nous calculons d'abord la distance P-distance inverse de chaque sommet à toutes les graines existantes. Ce calcul est réalisé efficacement dans notre architecture de type Pregel par une exploration BFS en parallèle à partir de chaque graine. Pour un ensemble de graines $\mathcal{S} = (s_1, s_2, \ldots, s_N)$, nous maintenons pour chaque sommet ν un vecteur de distances $dist(\nu) = (d_1, d_2, \ldots, d_N)$ où $d_i = dist_v(s_i, \nu)$ est la P-distance inverse de ν à la graine s_i. Ce vecteur est mis à jour pour chaque arête rencontrée durant l'exploration de l'algorithme BFS.

Étant donnée que l'exploration BFS dans des grands graphes est très coûteuse, nous proposons de limiter la distance d'exploration de l'algorithme BFS. En effet, dans la plupart des grands graphes (comme les graphes sociaux) les communautés peuvent être capturées en sélectionnant les graines parmi les sommets aux degrés les plus élevés, qui représentent le centre de ces communautés. Intuitivement, la distance entre le centre d'une communauté et les autres sommets à l'intérieur de la communauté est faible. Les résultats expérimentaux, en accord avec la théorie de « Six degrés de séparation » (Newman *et al.*, 2006a), montrent que le rayon d'un bloc (autrement dit la distance entre la graine et les membres potentiels de la communauté) est faible et, par conséquent, la profondeur d'exploration de l'algorithme BFS peut être choisie comme ayant une petite valeur. Les expériences sur le réseau social Livejournal (Leskovec *et al.*, 2008) par exemple montrent que le nombre de sommets/arêtes atteints en partant d'un ensemble de *200 graines* est d'environ 88 % et 96 % en limitant l'exploration BFS uniquement à 3 et 4 sauts respectivement.

Finalement nous calculons un vecteur de distances pour chaque arête du graphe. Considérons une arête $\varepsilon(\nu, \nu')$ et les vecteurs de distances pour ses deux sommets correspondants, $dist(\nu) = (d_1, d_2, \ldots, d_N)$ et $dist(\nu') = (d'_1, d'_2, \ldots, d'_N)$. En appliquant la Définition 4 nous calculons le vecteur de distances d'arête $dist(\varepsilon) = (D_1, D_2, \ldots, D_N)$ comme étant :

$$\forall i \in [1..N],\ D_i = dist_e(s_i, \varepsilon)$$

3.3.4. Affectation des arêtes aux blocs

Enfin, nous pouvons allouer les différents arêtes aux blocs en fonction de leur vecteur de distances d'arête calculé précédemment. Nous décidons qu'une arête appartient au bloc dont la graine est la *plus proche* de cette arête. Pour les arêtes sans valeur de distance (ce qui signifie que ses deux sommets n'ont pas été atteints par aucune graine pendant l'exploration BFS), nous les allouons à un bloc supplémentaire.

EXEMPLE 5. — Nous illustrons le processus d'allocation d'arête avec l'exemple de la figure 2. Supposons que nous avons déjà calculé les vecteurs de distance pour les

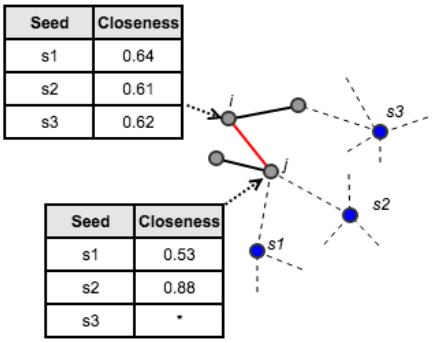

Figure 2. Exemple d'allocation d'arête

sommets i et j en considérant trois graines s_1, s_2 et s_3. Notons que la valeur '*' signifie que le sommet actuel ne peut pas être atteint par la graine s_3 pendant l'exploration BFS. Nous additionnons (ou faisons la moyenne) les deux vecteurs pour déterminer le vecteur de distances d'arête pour $e(i,j)$: $dist(i,j) = (0.64+0.53, 0.61+0.88, 0.62+0.0) = (1.17, 1.49, 0.62)$. Dans notre exemple nous pouvons clairement observer que l'arête e devrait être attribuée à s_2 car elle a une valeur de proximité maximale pour cette graine. □

Observons que certaines optimisations sont possibles pour stocker les vecteurs de distances de sommet et pour le calcul du vecteur de distances d'arête. Par exemple, nous pouvons éviter de garder toutes les valeurs de distance à chaque graine, car dans cette étape d'allocation d'arêtes seule la valeur maximale est utilisée pour allouer une arête à un bloc. Nous pouvons donc ne conserver que les *top-k* valeurs pour chaque sommet, avec $k \leq |\ calS|$. Le résultat final est évidemment plus précis pour des valeurs de k plus élevées.

4. Algorithmes de fusion et d'éclatement de blocs

Notre partitionnement en blocs respecte les propriétés topologiques du graphe (social), comme par exemple les communautés locales et la distribution en loi de puissance des degrés des sommets, pour réduire considérablement les coûts de communication par rapport à une stratégie d'allocation aléatoire (Gleich, Seshadhri, 2012 ; Whang *et al.*, 2013).

Étant donné un certain nombre de serveurs P, nous devons déterminer comment répartir les différents blocs sur ces serveurs en tenant compte de deux critères :

– minimiser les coûts de communication globalement ;

– équilibrer la charge de travail, de stockage et de calcul entre les serveurs.

Ces conditions peuvent être prises en compte par la définition suivante.

DÉFINITION 6 (Partitionnement équilibré d'arêtes). — *Considérons un graphe $G(V, E)$ où V est l'ensemble des sommets et E l'ensemble des arêtes, ainsi qu'un ensemble de blocs $\mathcal{B}$ et de serveurs P. Le* partitionnement équilibré d'arêtes $\mathcal{A}(\mathcal{B}, P)$ *est défini comme:*

$$\mathcal{A}(\mathcal{B}, P) \in 2^{\mathcal{B}},\ tel\ que \begin{cases} \forall \mathcal{A}' \in 2^{\mathcal{B}}, \frac{1}{|V|} \sum_{v \in V} |alloc(v, \mathcal{A})| \\ \qquad\qquad \leq \frac{1}{|V|} \sum_{v \in V} |alloc(v, \mathcal{A}')| \\ \\ \forall i \in [1..P], \eta \frac{|E|}{P} \leq |Edge(p_i)| \leq \lambda \frac{|E|}{P} \end{cases}$$

où p_i désigne une partition (serveur) et $Edge(p_i)$ les arêtes qu'elle contient, $alloc(e, \mathcal{A})$ est l'ensemble de partitions auxquelles l'arête e *est allouée avec le partitionnement $\mathcal{A}$ (plus d'une partition si le sommet est répliqué) et $(0 \leq \eta \leq 1 \leq \lambda)$ sont des constantes de valeur faible pour contrôler le stockage de chaque partition.*

La première partie de la définition signifie que le partitionnement $\mathcal{A}$ est celui qui minimise le facteur de réplication des arêtes *(VRF)*. La mesure *VRF* adoptée par exemple dans (Gonzalez *et al.*, 2012) signifie que moins le sommet est partagé par des partitions en moyenne, moins il y a de communication initiée par le système entre les partitions pour la synchronisation des sommets avant de débuter la prochaine itération. La seconde partie de la définition permet de contrôler la taille d'une partition pour s'adapter à la capacité du serveur et d'avoir une distribution d'arêtes quasi-équilibrée.

En s'appuyant sur la Définition 6, nous pouvons procéder au partitionnement final en fonction des différents blocs que nous avons construits.

4.1. Éclatement de bloc

Étant donné que l'attribution des arêtes aux blocs est basée uniquement sur un critère de distance, certains blocs peuvent ne pas correspondre à la taille maximale autorisée pour une partition (deuxième partie de la Définition 6). Par conséquent, nous proposons une stratégie d'éclatement simple. Supposons que la taille d'une partition p_i soit $(\beta - 1)\lambda \frac{|E|}{P} \leq |Edge(p_i)| < \beta\lambda \frac{|E|}{P}$. Nous appliquons ensuite notre algorithme de construction de blocs à la partition p_i avec β seeds pour la diviser en β sous-blocs. Nous pouvons éventuellement itérer le processus pour l'un des sous-blocs qui dépasse la taille de la partition.

4.2. Fusion de blocs

Notre construction de blocs peut également entraîner la construction de certains blocs dont la taille est inférieure à la taille minimale (c'est-à-dire $\eta \frac{|E|}{P}$, voir Définition 6). Pour un tel bloc, nous ré-affectons ses arêtes en ne considérant plus sa graine. Observons que cela peut conduire par la suite à l'éclatement de certains blocs.

4.3. Allocation de blocs

Nous supposons que, après les éventuels éclatements requis, la taille de tous les blocs respecte la limite de taille des partitions. Pour attribuer les blocs aux différentes partitions, deux stratégies peuvent être envisagées: en se basant uniquement sur l'équilibrage des tailles de partition ou sur la minimisation du facteur de réplication entre les partitions.

Si nous considérons cette dernière approche, nous pouvons identifier les inconvénients suivants :

– la complexité est exponentielle pour trouver la meilleure allocation de blocs en tenant compte de ce critère,

– la taille finale de chaque partition peut différer fortement de l'une à l'autre,

– réduire le facteur de réplication global ne réduira pas dans une même proportion le coût des algorithmes de marches aléatoires car un chemin commençant dans un bloc et finissant dans un autre est peu probable (selon notre construction de blocs) et enfin,

– ce partitionnement ne peut pas évoluer de manière dynamique et le partitionnement doit être reconstruit lorsque plusieurs arêtes sont ajoutées ou supprimées.

Par conséquent, nous décidons d'adopter une allocation de blocs en considérant uniquement le critère de taille, pour parvenir à un partage équilibré. Nous proposons un algorithme *glouton* simple mais efficace. Nous allouons le plus grand bloc à la partition de plus petite taille, et nous itérons cette stratégie jusqu'à ce que tous les blocs soient alloués. Par conséquent, cette allocation est en $O(|\mathcal{B}|)$ où $\mathcal{B}$ représente l'ensemble des blocs.

L'algorithme complet est présenté dans Algorithme 2 où *split* fait référence à une fonction qui réalise la division de bloc présentée ci-dessus, *sortSize* est une fonction qui trie un ensemble de blocs selon leur taille, du plus grand au plus petit, et *first* renvoie le premier élément d'un ensemble ordonné.

4.4. Gérer la dynamicité du graphe

Les grands graphes, en particulier pour les applications de réseaux sociaux, se caractérisent souvent par une forte dynamicité. Un aspect important de notre algorithme de partitionnement est sa capacité à gérer cette dynamicité. En effet, lors de l'ajout d'une nouvelle arête (par exemple, lors de l'ajout d'un ami sur Facebook ou d'une URL sur un site Web), nous devons simplement agréger les deux vecteurs de distances de sommet des deux sommets de l'arête si les deux sommets étaient déjà présents dans le graphe pour calculer son vecteur de distances d'arête. Ensuite, nous attribuons l'arête au bloc, et par conséquent à la partition, avec le score de distance le plus élevé. Si l'un des sommets est nouveau, nous devons d'abord effectuer l'exploration BFS à partir de ce sommet et calculer son vecteur de distances de sommet.

Cette allocation d'arête peut conduire éventuellement à une division de blocs qui peut être traitée avec notre algorithme d'éclatement. À l'inverse, avec l'élimination

Algorithme 2 : Algorithme d'allocation de blocs

entrée : un ensemble $\mathcal{B} = \{b_1, \ldots, b_n\}$ de blocs, un ensemble $\mathcal{P} = \{p_1, \ldots, p_m\}$ de partitions
sortie : chaque bloc est alloué à un des $p_j \in \mathcal{P}$

// Initialisation pour éviter les grands blocs
$\mathcal{B}' = \emptyset$
pour chaque $b_i \in \mathcal{B}$ **faire**
  si $b_i.size > \lambda\frac{|E|}{n}$ **alors**
    $\mathcal{B}' = \mathcal{B}' \cup split(b_i)$
  $\mathcal{B}' = \mathcal{B}' \cup b_i$
// Tri de l'ensemble des blocs par taille décroissante
$\mathcal{B}' = sortSize(\mathcal{B}')$
$b = first(\mathcal{B}')$; **tant que** $\mathcal{B}' \neq \emptyset$ **faire**
  $p_i = smallest(\mathcal{P})$;
  $p_i = merge(p_i, b)$; //fusionne b avec la plus petite partition
  $\mathcal{B}' = \mathcal{B}' - \{b\}$;
  $b = first(\mathcal{B}')$;
Retourne $\mathcal{P}$;

d'une arête la taille d'un bloc peut devenir trop petite et nous procédons à notre algorithme de fusion de blocs.

5. Experiences

Cette section présente des expériences sur notre stratégie de partitionnement basée sur les blocs. Nous la comparons avec les méthodes de partitionnement d'arêtes existantes: les approches basées sur le hachage (Xin *et al*., 2013) et sur un algorithme glouton (Gonzalez *et al*., 2012).

5.1. Données et compétiteurs

Les calculs sont effectués à l'aide des API GraphX (Xin *et al*., 2013) dans Spark (Zaharia *et al*., 2012) (version 1.3.1), sur un cluster de 16 nœuds. Chaque machine dispose de 22 cœurs avec 60 Go de RAM fonctionnant sous Linux OS. Pour nos expériences, nous fixons la probabilité de téléportation α à une valeur classique de 0,15. La profondeur de l'exploration BFS (c'est-à-dire, la longueur maximale considérée pour les chemins de la graine jusqu'aux autres sommets) est fixée à 4. Donc, nous souhaitons calculer uniquement pour des sommets voisins de la graine considérée, plutôt que pour tous les sommets du graphe, en accord avec la théorie des *Six Degrés de Séparation* (Newman *et al*., 2006b).

5.1.1. Jeux de données

Nous validons notre approche sur deux ensembles de données: LiveJournal (Chierichetti *et al.*, 2009) avec 4,8M de sommets et 68,9M d'arêtes, et Pokec (Takac, Zabovsky, 2012) avec 1,6M de sommets et 30,6M d'arêtes. Ces jeux de données peuvent être téléchargés à partir de *SNAP* [1].

5.1.2. Compétiteurs

Partitionnement par hachage. Il y a quatre méthodes de partitionnement aléatoires (par hachage) largement utilisées [2], introduit dans GraphX :

– RandomVertexCut : alloue les arêtes aux partitions en hachant les identifiants de sommet origine et de destination.

– CanonicalRandomVertexCut : affecte des arêtes aux partitions en hachant les identifiants des sommets d'origine et de destination dans une direction canonique.

– EdgePartition1D : alloue les arêtes aux partitions en utilisant uniquement l'ID du sommet de la source, co-localisant les arêtes possédant le même sommet origine.

– EdgePartition2D : alloue les arêtes aux partitions à l'aide d'un partitionnement 2D de la matrice creuse d'adjacence des arêtes.

Coupe des sommets gloutonne (Greedy Vertex-Cuts). PowerGraph propose une heuristique gloutonne pour le processus de placement d'arêtes qui repose sur l'allocation précédente des sommets pour déterminer la partition à laquelle la prochaine arête devrait être attribuée.

5.1.3. Algorithmes de graphe considérés

Notre partitionnement est adapté aux algorithmes basés sur les marches aléatoires, qui nécessitent des coûts de communication importants entre les partitions. Nous avons donc implanté deux de ces algorithmes pour la validation: *PageRank* et *Fullly Multiple Random Walks*. Cependant, nous souhaitons également prouver que notre approche offre également de bonnes performances pour d'autres algorithmes de graphes. Par conséquent, nous avons également effectué des expériences avec un algorithme de calcul de composantes connexes. Donc, plus précisément, nous comparons différentes stratégies de partitionnement pour les algorithmes suivants :

– Fully Multiple Random Walks(FMRW) : Plusieurs algorithmes pour le *classement* et la *recommandation* proposés dans la littérature (comme le calcul fully Personalized PageRank (PPR) présenté dans (Bahmani *et al.*, 2011 ; Sarma *et al.*, 2008)) simulent plusieurs marches aléatoires à partir de chaque sommet pour leur calcul. Donc, pour nos expériences, nous décidons d'effectuer deux marches aléatoires indépendantes de longueur 4 à partir de chaque sommet. Observons que nous pouvons obtenir des marches plus longues en combinant des marches aléatoires courtes comme

1. https://snap.stanford.edu/data/index.html
2. voir les détails et implantations sur http://spark.apache.org/docs/latest/api/scala/index.html

dans (Bahmani *et al.*, 2011 ; Sarma *et al.*, 2008). Ainsi, pour ce calcul, nous avons 4 super-étapes (itérations) dans notre implantation Pregel. Pour chaque super-étape, chaque marche aléatoire repartant d'un sommet se dirigera à l'un des voisins directs choisi au hasard (avec une distribution uniforme). Notons ici que l'opération *restart* n'est pas introduite. Enfin, nous collectons les $|V| * 2$ marches aléatoires quand toutes les super-étapes sont réalisées.

– PageRank : les messages transmis entre les sommets sont uniquement des valeurs de score. Ainsi, le coût global en E/S n'est pas très élevé puisque la taille du message est petite, même si le nombre de messages est important. Plus particulièrement dans nos expériences nous avons implanté des versions statiques et dynamiques de PageRank. Le nombre de super-étapes est fixé pour la version statique, afin de contrôler le temps global de calcul sur le graphe. La version dynamique repose sur un seuil de convergence, traditionnellement 0,005 ou 0,001, ce qui fournit des résultats plus précis.

– Composantes connexes (CCs) : un algorithme pour calculer les composantes connexes qui peuvent habituellement être utilisées pour approximer les structures de clusters dans un graphe. En d'autres termes, nous pouvons considérer cela comme un algorithme sensible à la localité. Par exemple, dans sa mise en œuvre sur le modèle de Pregel, nous étiquetons chaque sommet avec l'identifiant de sommet le plus petit de la composante à laquel il appartient. Lors du démarrage de l'algorithme, nous définissons l'étiquette initiale de chaque sommet comme son identifiant propre. À chaque super-étape, le sommet recevra des messages (étiquettes) de ses voisins s'ils ont une étiquette inférieure. Ensuite, chaque sommet présente son étiquette à l'aide de la valeur d'étiquette la plus basse reçue pendant cette super-étape (ou conserve son étiquette si elle ne reçoit aucun message). Le calcul se termine après un nombre pré-défini de super-étapes, ou lorsqu'il n'y a plus de message à envoyer. Enfin, les sommets avec la même étiquette constitue une composante connexe.

– Composantes fortement connexes *(Strongly Connected Components(SCC)* : La version graphe-dirigé de l'algorithme CC précédent.

– Plus courts chemins *(Shortest Paths(SP))* : Un algorithme de graphe classique pour calculer la distance la plus courte entre chaque sommet et un ou plusieurs sommets-requêtes. Avant de commencer à exécuter l'algorithme, un ensemble de sommets est sélectionné en tant que cibles. Ensuite, pendant chaque super-étape, le sommet recevra les informations de distance des voisins à chaque cible pour déterminer si sa valeur doit être mise à jour. Le calcul se termine quand aucun nouveau message n'est transmis sur le graphe. En général, il se produit plus de communication aux premières super-étapes, car presque tous les sommets acquièrent des informations auprès des voisins et mettent à jour leur distance. Cependant ils atteignent rapidement la convergence en raison d'une diminution des mises à jour sur les sommets.

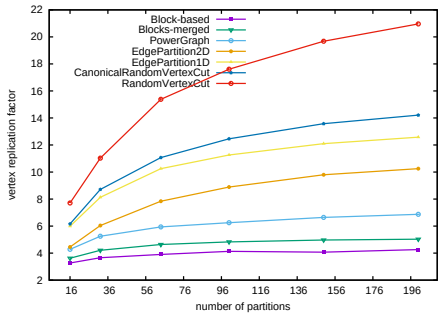

Figure 3. VRF w.r.t. edge partitioning methods on LiveJournal

Tableau 1. Messages transmis pour les FMRW (LiveJournal)

	Random-Vertex-Cut		Block-based partitioning			
#Partitions	VRF	mess. réels	VRF	mess. réels	mess. attendus	ratio
64	15,38	303,5m	3,90	55,3m	76,9m	0,72
100	17,61	381,9m	4,13	61,8m	89,4m	0,69
150	19,68	464,8m	4,07	70,6m	96,1m	0,73
200	21,12	525,6m	4,26	76,0m	106,0m	0,72

5.2. Communication

Notre approche vise à réduire le temps d'exécution des calculs sur les grands graphes grâce à une réduction significative des coûts de communication.

5.2.1. Facteur de réplication des sommets (VRF)

VRF est traditionnellement utilisé pour comparer deux partitionnements du point de vue des coûts de communication, indépendamment de l'algorithme exécuté. Nous comparons le VRF de notre partitionnement basé sur les blocs avec nos concurrents

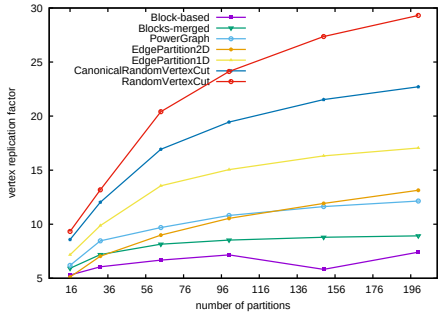

Figure 4. VRF en fonction de la méthode de partitionnement d'arêtes sur Pokec

pour différents nombres de partitions. Les résultats sont représentés sur les figures 3 et 4. Nous observons, comme cela avait été remarqué dans (Gonzalez *et al*., 2012), que les stratégies de partitionnement basées sur la topologie dépassent comme attendues les méthodes basées sur le hachage : le VRF a diminué de 30-60 % pour PowerGraph et 60-80 % pour notre stratégie de blocs par rapport aux stratégies utilisées dans GraphX. Cette expérience illustre également le bénéfice de notre approche globale pour l'allocation des arêtes par rapport à une approche gloutonne avec en moyenne un VRF de 40 % inférieur.

Nous observons également que le VRF présente une croissance sous-linéaire en fonction du nombre de partitions pour toutes les stratégies. En effet, un faible nombre de partitions conduit généralement à un grand nombre de sommets partagés par deux arêtes de différentes partitions. Lors de l'augmentation du nombre de partitions, nous augmentons également le nombre de sommets répliqués, mais notre partitionnement respectant les « communautés » permet de limiter ce nombre. Observons que la croissance reste cependant toujours plus importante pour les stratégies de GraphX.

5.2.2. Nombre de messages

VRF est un critère général pour comparer deux stratégies de partitionnement indépendamment des algorithmes, mais nous nous attendons à ce que notre partitionnement présente des résultats encore meilleurs pour les algorithmes basés sur les marches aléatoires. Par conséquent, pour estimer le bénéfice de notre approche, nous simulons des marches aléatoires multiples (FMRW) et nous mesurons le nombre de messages échangés entre les partitions. À partir de chaque sommet, nous effectuons 2 marches aléatoires de longueur 4 et nous reportons les résultats des expériences dans le tableau 2. Nous observons que notre méthode réduit considérablement le nombre de messages échangés entre les partitions. Par exemple, avec 100 partitions, 61,8 millions de messages sont nécessaires pour traiter le FMRW avec notre méthode, tandis que 381,9 millions sont transmis avec la méthode de coupe aléatoire des sommets (*Random-Vertex-Cut*), soit une baisse de 84 %. Ce résultat était attendu puisque le VRF était 3-4 fois plus bas avec notre méthode qu'avec la méthode *Random-Vertex-Cut*. Mais on constate également que si la réduction du nombre de messages et du VRF était proportionnelle, le système devrait échanger 89,4 millions de messages. Cette différence de 30 % du nombre de messages transmis valide notre intuition selon laquelle les marches aléatoires ont tendance à rester dans un *cluster* local (communauté). Ainsi, les sommets peu répliqués (près de la graine dans le bloc) sont accédés plus de fois et, à l'inverse, peu de marches aléatoires atteignent les sommets les plus éloignés et les plus répliqués. Des résultats similaires sont obtenus à partir d'expériences sur Pokec. Nous pouvons par conséquent affirmer que notre partitionnement peut non seulement économiser beaucoup de coûts de communication dans le cas général en raison du faible VRF, et que cette diminution est encore plus importante que celle attendue pour les algorithmes reposant sur des marches aléatoires dans des réseaux dont la distribution des degrés entrants/sortants est biaisée.

Tableau 2. Nombre exact de messages transmis pour FMRW suivant la stratégie de partitionnement d'arêtes

Nb. de partitions	Random-Vertex-Cut	Block-based
64	303,5m:15,38	55,3m(76,9m):3,90
100	381,9m:17,61	61,8m(89,4m):4,13
150	464,8m:19,68	70,6m(96,1m):4,07
200	525,6m:21,12	76,0m(106,0m):4,26

5.3. Temps d'exécution

Nous proposons d'évaluer dans quelle mesure le temps d'exécution des différents algorithmes de traitement du graphe est impacté par notre partitionnement comparé aux autres approches. Tout d'abord, nous lançons FMRW, un algorithme de communication lourde, sur les jeux de données LiveJournal et Pokec, respectivement, avec 3 marches aléatoires de longueur 4 générées à partir de chaque sommet. À partir des résultats de la figure 5, nous voyons que notre partitionnement peut économiser entre 20 à 65 % du temps d'exécution, par rapport aux autres partitionnements, pour les deux jeux de données. Nous constatons également que, comme prévu, le temps d'exécution augmente avec le nombre de partitions, car le traitement de l'algorithme nécessitera plus de communication. En ce qui concerne les compétiteurs, nous observons que les performances de PowerGraph s'améliorent avec le nombre de partitions, contrairement aux méthodes de partitionnement par hachage de GraphX. En effet, l'heuristique gloutonne de PowerGraph est basée sur la réduction des coûts de communication, de sorte que plus nous avons de partitions, plus important est le gain comparé à GraphX.

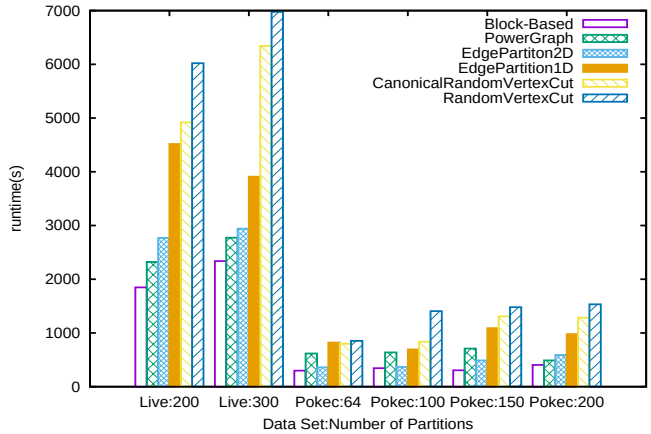

Figure 5. Temps d'exécution pour FMRW avec les différents méthodes de partitionnement pour LiverJournal et Pokec

Nous testons également notre méthode avec l'algorithme classique de PageRank. Nous considérons les approches statique (nombre fixe d'itérations) et dynamique (avec

convergence et seuil). Nous considérons qu'il y a 200 partitions. Nous procédons à respectivement 30, 50 et 100 itérations pour le PageRank statique et choisissons pour le PageRank dynamique respectivement 0,005 et 0,001 comme facteur de convergence. Les figures 6 et 7 présentent les résultats pour les jeux de données LiveJournal et Pokec, et confirment que notre méthode de partitionnement est supérieure aux autres. Bien que nous observions un gain moins important de 5-20 % pour l'implantation statique de PageRank, nous atteignons un gain de 20-55 % pour l'implantation dynamique.

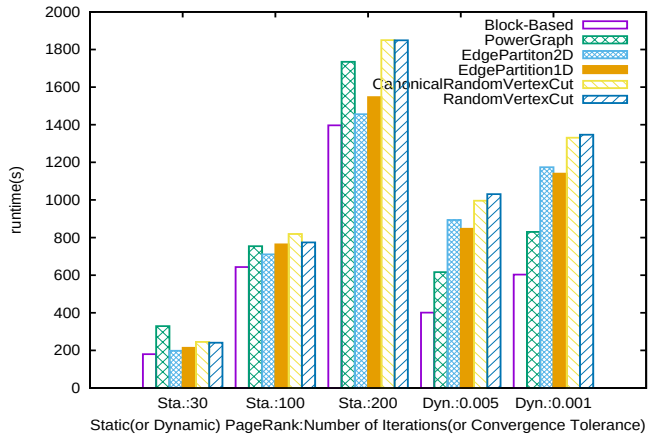

Figure 6. Temps d'exécution pour le PageRank statique et dynamique pour LiverJournal

Donc, pour le PageRank statique, contrairement à FMRW alors que nous procédons également à un nombre fixe de super-étapes, les performances des stratégies basées sur le hachage sont proches de celles obtenues avec PowerGraph et avec notre méthode. Les principales différences avec FMRW sont la taille des messages échangés entre les partitions et l'importance du calcul effectué. Alors qu'un message pour FMRW contient l'information sur les différentes marches aléatoires que nous construisons, un message dans PageRank ne contient qu'une valeur de score et un nœud ne transmet qu'une agrégation de scores. Pour le PageRank dynamique (avec le seuil de convergence), notre méthode fournit un gain significatif pour la performance. Ces expériences illustrent que nous fournissons un partitionnement qui suit les communautés existantes dans le graphe. En effet, les marches aléatoires sont plus susceptibles de rester à l'intérieur d'une partition donnée, ce qui entraîne une convergence plus rapide des scores.

Par ailleurs nous proposons d'utiliser un algorithme sensible à la localité permettant de construire des composantes connexes, pour vérifier si notre partitionnement fournit également de bonnes performances sur d'autres algorithmes qui ne sont pas basés sur des marches aléatoires. Nous avons implanté l'algorithme classique de détection de composantes connexes ainsi que l'algorithme de détection de composantes fortements connexes (qui considère un graphe orienté). Les résultats des figures 8 et

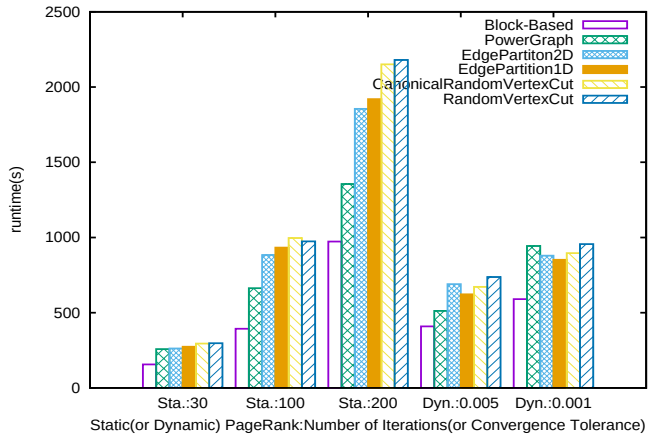

Figure 7. Temps d'exécution pour le PageRank statique et dynamique pour Pokec

9 montrent que notre méthode de partitionnement surpasse également d'autres approches pour cet algorithme. La raison est que nos partitions sont basées sur la topologie des graphes et plus précisément sur la connectivité et la proximité. Ainsi, avec notre approche, les messages entre les partitions sont moins fréquents qu'avec d'autres stratégies.

Nous avons également implanté un autre algorithme de calcul classique non-aléatoire : l'algorithme de calcul des plus courts chemins (SP) sur nos partitions. Nous exécutons les différentes stratégies de partitionnement mentionnées dans cet article sur les deux jeux de données pour différents nombres de partitionns. Nos résultats présentés figure 10 confirment que notre approche présente de meilleurs temps d'exécution par rapport à d'autres en raison de la réduction significative de la communication au cours du calcul de graphes.

6. Conclusion et travaux futurs

Nous présentons dans cet article un partitionnement d'arêtes (vertex-cut) pour les algorithmes basés sur les marches aléatoires en s'appuyant sur la topologie du graphe pour créer des blocs qui respectent les communautés locales. Nous proposons des algorithmes d'éclatement et de fusion pour obtenir et maintenir le partitionnement final. Nous démontrons expérimentalement que notre proposition surpasse les solutions existantes.

Comme travail futur, nous prévoyons d'étudier différents algorithmes de sélection de graines. Bien que ce problème ait été étudié dans des contextes différents (voir (Whang *et al.*, 2013 ; Dahimene *et al.*, 2014)), nous pensons que la nature des algorithmes de calcul sur des graphes, ici des algorithmes basés sur les marches aléatoires, doit être prise en compte lors du choix des graines. Nous avons également l'intention de concevoir une stratégie permettant d'allouer aux blocs existants les 5 à

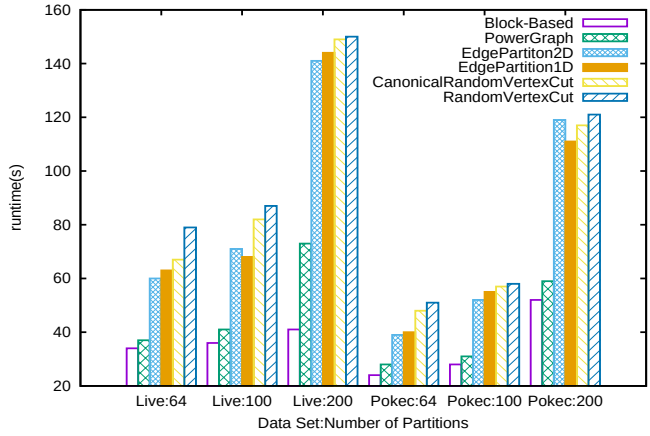

Figure 8. Temps d'exécution pour le calcul des composantes connexes (CC) pour LiverJournal et Pokec

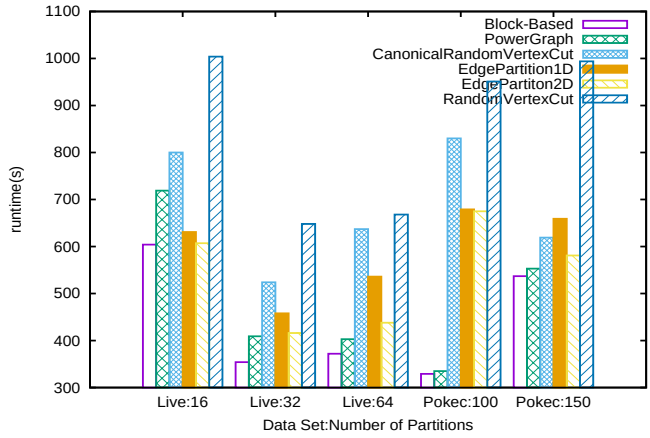

Figure 9. Temps d'exécution pour le calcul des composantes fortement connexes (SCC) computation pour LiverJournal et Pokec

10 % de sommets qui ne sont actuellement pas atteints par l'exploration BFS exécutée depuis chaque graine. Ces sommets, qui sont situés à la périphérie du graphe social, sont actuellement classés dans un bloc supplémentaire.

Bibliographie

Andersen R., Chung F. R. K., Lang K. J. (2006). Local Graph Partitioning using PageRank Vectors. In *Proc. of the IEEE symposium on foundations of computer science (focs)*, p. 475–486.

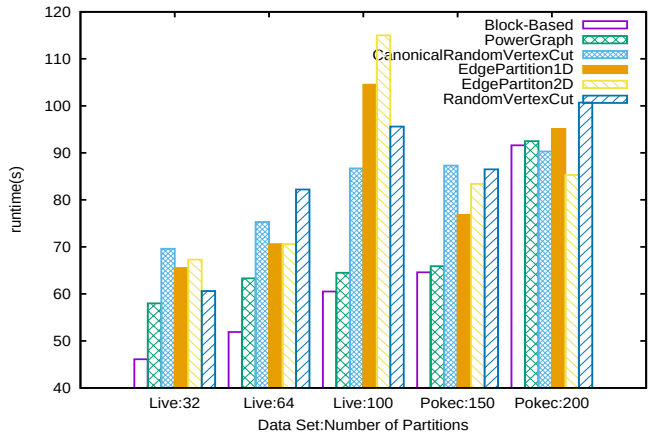

Figure 10. Temps d'exécution pour le calcul des plus courts chemins (SP) pour LiverJournal et Pokec

Apache. (s. d.). *Giraph.* Consulté sur http://giraph.apache.org

Bahmani B., Chakrabarti K., Xin D. (2011). Fast Personalized PageRank on MapReduce. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, p. 973–984.

Bahmani B., Chowdhury A., Goel A. (2010). Fast Incremental and Personalized PageRank. *Proc. of the VLDB Endowment (PVLDB)*, vol. 4, n° 3, p. 173–184.

Bourse F., Lelarge M., Vojnovic M. (2014). Balanced Graph Edge Partition. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (kdd)*, p. 1456–1465.

Chierichetti F., Kumar R., Lattanzi S., Mitzenmacher M., Panconesi A., Raghavan P. (2009). On Compressing Social Networks. In *Proc. of the ACM SIGKDD international conference on knowledge discovery and data mining (kdd)*, p. 219–228.

Dahimene R., Constantin C., Mouza C. du. (2014). RecLand: A Recommender System for Social Networks. In *Proc. of the ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, p. 2063–2065.

Gleich D. F., Seshadhri C. (2012). Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods. In *Proc. of the ACM SIGKDD international conference on knowledge discovery and data mining (kdd)*, p. 597–605.

Gonzalez J. E., Low Y., Gu H., Bickson D., Guestrin C. (2012). PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, p. 17–30.

Jeh G., Widom J. (2003). Scaling Personalized Web Search. In *Proc. of the international world wide web conference (www)*, p. 271–279. Consulté sur http://doi.acm.org/10.1145/775152.775191

Karypis G., Kumar V. (1998). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Scientific Computing*, vol. 20, n° 1, p. 359–392. Consulté sur http://dx.doi.org/10.1137/S1064827595287997

Kernighan B. W., Lin S. (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, vol. 49, n° 2, p. 291-307.

Leskovec J., Lang K. J., Dasgupta A., Mahoney M. W. (2008). Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR*, vol. abs/0810.1355.

Low Y., Gonzalez J., Kyrola A., Bickson D., Guestrin C., Hellerstein J. M. (2012). Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment (PVLDB)*, vol. 5, n° 8, p. 716–727.

Malewicz G., Austern M. H., Bik A. J. C., Dehnert J. C., Horn I., Leiser N. *et al*. (2009). Pregel: a System for Large-scale Graph Processing. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC)*, p. 6.

Newman M., Barabasi A.-L., Watts D. J. (2006a). *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press.

Newman M., Barabasi A.-L., Watts D. J. (2006b). *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press.

Roy A., Bindschaedler L., Malicevic J., Zwaenepoel W. (2015). Chaos: Scale-out Graph Processing from Secondary Storage. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, p. 410–424.

Salihoglu S., Widom J. (2013). GPS: a Graph Processing System. In *Proc. of the Conference on Scientific and Statistical Database Management (SSDBM)*, p. 22:1–22:12.

Sarkar P., Moore A. W. (2010). Fast Nearest-neighbor Search in Disk-resident Graphs. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, p. 513–522.

Sarma A. D., Gollapudi S., Panigrahy R. (2008). Estimating PageRank on Graph Streams. In *Proc. of the ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (pods)*, p. 69–78.

Takac L., Zabovsky M. (2012). Data Analysis in Public Social Networks. *Present Day Trends of Innovations*, p. 1-6.

Valiant L. G. (2008). A Bridging Model for Multi-core Computing. In *Proc. of the Annual European Symposium Algorithms - ESA* , p. 13–28.

Whang J. J., Gleich D. F., Dhillon I. S. (2013). Overlapping Community detection Using Seed set Expansion. In *Proc. of the ACM international conference on information and knowledge management (cikm)*, p. 2099–2108.

Xin R. S., Gonzalez J. E., Franklin M. J., Stoica I. (2013). GraphX: a Resilient Distributed Graph System on Spark. In *Proc. of the International SIGMOD Workshop on Graph Data Management Experiences and Systems (GRADES)*, p. 2.

Yang S., Yan X., Zong B., Khan A. (2012). Towards Effective Partition Management for Large Graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, p. 517–528.

Zaharia M., Chowdhury M., Das T., Dave A., Ma J., McCauly M. *et al*. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the USENIX symposium on networked systems design and implementation (nsdi)*, p. 15–28.