Rapport de Calculs Intensifs
Implémentation du PageRank avec MPI

Auteur : Ayaz Badouraly
Date : 13 avril 2016

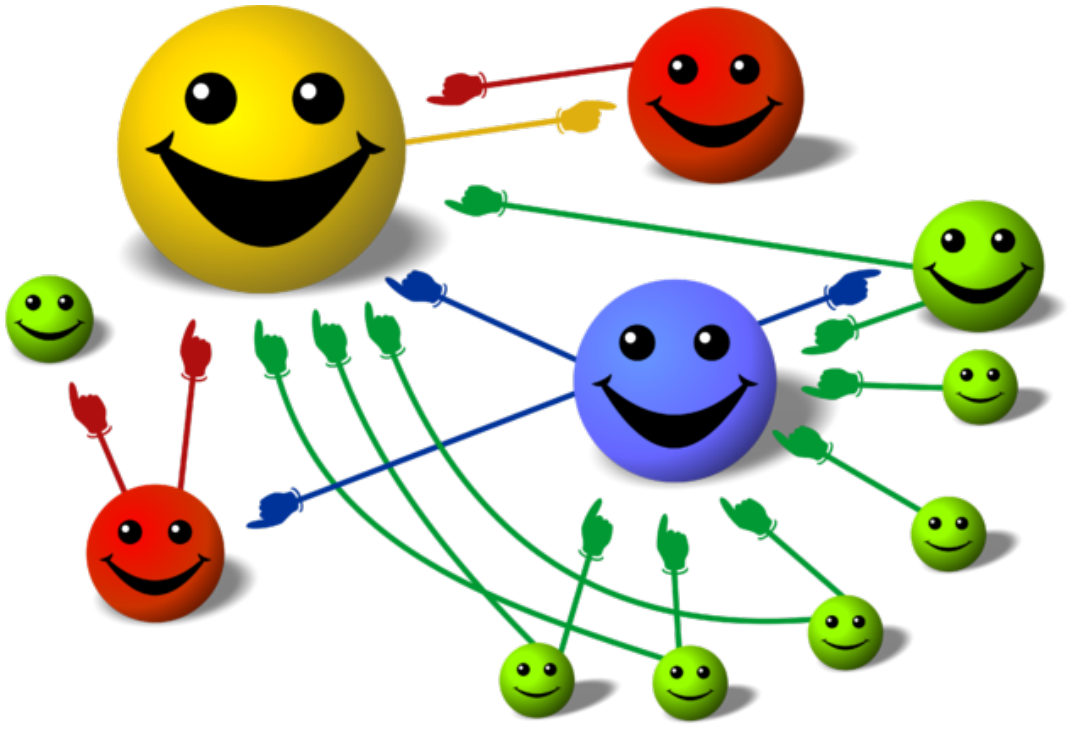

Table des matières

1 Introduction

Pour classer les pages web indépendamment de la sémantique, Sergey Brin et Lawrence Page ont conçu le *pagerank* en 1998. L'idée était d'attribuer une note à une page, vue comme un sommet du graphe orienté du web. Le critère de classement sélectionné est alors purement quantitatif et repose sur les liens entre les différents nœuds.

Malgré la simplicité de l'idée sous-jacente au pagerank, la taille du graphe du web ne permet pas de calculer son pagerank en un temps raisonnable sur un seul cœur d'une seule machine, aussi puissante soit-elle.

Je propose ici une implémentation parallèle et distribuée de l'algorithme du pagerank, utilisant la librairie `MPI` (*Message Passing Interface*).

Ce rapport a pour but de présenter le travail que j'ai effectué entre le vendredi 4 mars et le mercredi 13 avril. Il s'agit d'un support au code source lui-même. Les points les plus subtiles du code source sont expliqués sous forme de commentaires directement dans les fichiers sources.

2 Algorithme du PageRank

2.1 Définition du *pagerank*

On considère un graphe orienté $G = (U, L)$ où U est l'ensemble des sommets du graphe, et L l'ensemble des arcs.

Par analogie à la matrice d'adjacence de G, on définit la transposée de la matrice d'adjacence normalisée $A = (a_{ij})$ de G telle que :

$$\forall (i,j) \in U \times U : a_{ij} = \begin{cases} \frac{1}{\#(j)} & \text{si } (j,i) \in L \\ 0 & \text{sinon} \end{cases}$$

où $\#(j)$ est le nombre de liens sortants du sommet j. A représente le surfeur qui navigue de proche en proche en cliquant sur un lien au hasard de la page dans laquelle il se trouve.

Pour modéliser le surfeur qui arrive sur une page puit (*ie.* sommet sans arc sortant) et qui va sur une page quelconque ensuite, on introduit la matrice $D = (d_{ij})$ telle que :

$$\forall (i,j) \in U \times U : d_{ij} = \begin{cases} \frac{1}{\text{card}(U)} & \text{si } d^+(j) = 0 \\ 0 & \text{sinon} \end{cases}$$

où $d^+(j)$ est le degré sortant du sommet j.

Enfin, pour modéliser le surfeur qui se lasse et qui va sur une nouvelle page au hasard, on introduit la matrice $E = (\frac{1}{\text{card}(U)})$ de la même dimension que les matrices précédentes et qui ne contient que le terme $\frac{1}{\text{card}(U)}$.

La matrice de changement de distribution M est alors définie comme :

$$M = dA + dD + (1-d)E$$

où d est le *damping factor* ; d vaut usuellement $0,85$.

Le vecteur pagerank p est définit comme l'unique vecteur à valeurs positives, unitaire pour la norme $\|\cdot\|_1$ et vérifiant :

$$Mp = p \tag{1}$$

2.2 Explication de l'algorithme

Plutôt que de résoudre analytiquement l'équation (1), nous allons approximer la solution.

On définit la suite récurrente :

$$\begin{cases} p^0 = \begin{pmatrix} \frac{1}{\text{card}(U)} \\ \vdots \\ \frac{1}{\text{card}(U)} \end{pmatrix} \\ \forall n \in \mathbb{N} : \quad p^{n+1} = Mp^n \end{cases}$$

On se donne un seuil de précision ε.

L'algorithme consiste à calculer petit à petit la suite $(p^0, p^1, p^2, \cdots)$, jusqu'à trouver $N \in \mathbb{N}$ tel que :

$$\left\| p^{N+1} - p^N \right\| < \varepsilon$$

p^N est alors une bonne approximation du pagerank p.

Puisque nous sommes en dimension finie, les normes sont équivalentes. J'ai donc choisi d'utiliser la norme infinie pour le calcul des distances, afin que la précision du calcul du pagerank soit directement liée à la précision de chacune de ses composantes.

2.3 Justification mathématique

Par construction, ${}^tM$ est une matrice stochastique. Donc 1 est valeur propre de ${}^tM$. De plus, ${}^tM$ est de norme subordonnée 1. On en déduit que le rayon spectral de ${}^tM$ vaut 1.

Les éléments de la matrice ${}^tA + {}^tD$ ne sont pas nécessairement tous strictement positifs, mais au moins positifs. Les éléments de la matrice ${}^tE$ sont tous strictements positifs. On en déduit que la matrice ${}^tM$ est primitive.

Le théorème de Perron-Frobenius permet de conclure sur l'existence et l'unicité du pagerank : c'est le vecteur de Perron-Frobenius.

Par ailleurs, cela montre que 1 est une valeur propre simple de ${}^tM$. La convergence de la suite (p^n) vers p est donc garantie, en tant que probabilité stationnaire de la chaîne de Markov de matrice de transition de ${}^tM$ irréductible.

3 Implémentation

Concernant l'arborescence du répertoire, les fichiers sont séparés enre le dossier `src/` pour les sources principales (`src/main.c` et `src/pagerank.c`), le dossier `lib/` pour les librairies (`lib/file.c`, `lib/matrix.c` et `lib/vector.c`) et le dossier `tests/` pour les tests unitaires.

Les entrées du programme se trouvent dans le dossier `input/` et les sorties dans le dossier `output/`.

La compilation [1] est automatisée grâce à un `Makefile`. Les exécutables sont générés dans le dossier `bin/` avec l'extension `.run`.

3.1 Structures de données

Dans tout le projet, les quantités sont des nombres flottants à double précision. En C, ils sont implémentés avec le mot-clé : `double`.

3.1.1 Représentation des vecteurs

Les vecteurs sont représentés sous forme de tableaux unidimensionnels de `double`.

Le fichier `lib/vector.h` contient la définition de la structure `Vector`. Elle contient un entier pour le nombre de composantes et un tableau alloué dynamiquement.

```
struct Vector
{
        int dim;
        double *vector;
};
```

1. *N.B. :* travaillant sous Linux, je garantis le fonctionnement sur cet OS — et plus généralement sur tout OS certifié POSIX — mais pas sur Windows (j'ai tout de même ajouté quelques instructions préprocesseurs concernant cet OS). Si besoin, je suis en mesure de fournir une machine Linux fonctionnelle.

3.1.2 Représentation des matrices

Les matrices considérées dans le problème du PageRank sont grandes et creuses [2]. Les stocker sous forme de tableaux bidimensionnels est donc très coûteux en espace. J'ai donc fait une implémentation des matrices sous la forme COO (*coordinate format*).

Le fichier `lib/matrix.h` contient la définition de la structure `COO_Matrix`. Elle contient un entier pour le nombre de composantes non nulles, deux tableaux de `int` alloués dynamiquement contenant les index i et j des composantes non nulles, et un tableau de `double` alloué dynamiquement contenant la valeur des composantes.

```
struct COO_Matrix
{
        int nnz;
        int *rows;
        int *cols;
        double *values;
};
```

Puisque c'est la seule forme de matrices utilisée dans le projet, j'ai mappé le mot-clé `Matrix` au mot `COO_Matrix` :

```
#define Matrix COO_Matrix
```

3.2 Les librairies (`lib/`)

3.2.1 `lib/file.h`

Ce fichier contient toutes les fonctions utiles pour intéragir avec la mémoire dure : lecture du disque et écriture sur le disque.

3.2.2 `lib/matrix.h`

Ce fichier contient la définition de la structure `COO_Matrix` sour forme de trois tableaux unidimentionnels. Les tableaux sont alloués dynamiquement. J'ai donc écrit un constructeur et un destructeur.

2. *cf.* la visualisation des matrices sous forme d'images PNG.

```
struct COO_Matrix construct_coo_matrix ( const int nnz );
        /* create a null matrix with COO format */
void destruct_coo_matrix ( struct COO_Matrix *mtx );
        /* free memory space */
```

Il est aussi possible de charger une matrice à partir d'un fichier du dossier `input/`. Le fichier doit respecter certaines contraintes énumérées dans la section 3.4.1.

```
struct COO_Matrix from_graphfile_coo_matrix ( const char *filename,
        int *nodes_number ); /* create a null matrix with COO format
        from a graph written in a file */
```

Pour accéder à un élément de la matrice, on passe par la fonction `get_value_coo_matrix`. Cette fonction prend en entrée les index i et j. S'il existe $k <$ `nzz` tel que `rows`$[k] = i$ et `cols`$[k] = j$, alors la fonction renvoie `values`$[k]$; sinon elle renvoie `0`.

J'ai écrit d'autres méthodes pour pouvoir intéragir plus facilement avec la structure de données (ajout d'un élément à une matrice, sortie écran, enregistrement dans un fichier, création d'une copie).

Dans le problème du PageRank, avoir une visualisation graphique de la matrice de transition d'une page à une autre. Ne serait-ce que pour « voir » sur quoi l'on travaille. J'ai donc écrit une fonction créant une image en noir et blanc de la dimension de la matrice : chaque pixel noir représente une composante nulle de la matrice, et chaque pixel blanc représente une composante non nulle.

```
#ifdef WITH_LIBPNG
int draw_coo_matrix ( const struct COO_Matrix *mtx,
       const char *filename ); /* create a picture of a COO matrix */
#endif
```

Dans le `Makefile`, si la variable `WITH_LIBPNG` vaut `yes`, l'édition de liens est exécutée avec l'option `-lpng`. Dans le code, on ajoute les headers :

```
#ifdef WITH_LIBPNG
#include <png.h>
#endif
```

La boucle principale est :

```
1          for ( int i = 0 ; i < dim ; ++i ) {
2                  #pragma omp parallel for
3                  for ( int j = 0 ; j < dim ; ++j ) {
4                          if ( get_value_coo_matrix(mtx, i, j) ) {
5                                  row[j] = 0xff;
6                          }
7                          else {
8                                  row[j] = 0x00;
9                          }
10                 }
11                 png_write_row(png_ptr, row);
12         }
```

Expliquons ce code :

variables en jeu

`mtx` la matrice dont on trace une visualisation

`dim` dimension de la matrice

`row` une ligne de l'image, tableau de pixels de dimension **`dim`**

`png_ptr` pointeur vers l'image PNG que l'on est en train de construire

lignes 4, 5 et 6

si la matrice a une composante de coordonnées (i, j) non nulle, on ajoute un pixel blanc à la ligne `row`

lignes 7, 8 et 9

sinon, on ajoute un pixel noir à la ligne `row`

ligne 11

on écrit la ligne `row` dans l'image ; les appels à `png_write_row` doivent se faire dans le sens des i croissant, donc les boucles ne sont pas interchangeables et encore moins parallélisables

ligne 2

il n'y a aucune dépendance de données ou de sortie, donc la boucle de la ligne 3 à 11 est parallélisable ; pour cette petite boucle, j'ai voulu essayer `OpenMP` : dans le `Makefile`, on ajoute l'option `-fopenmp` au compilateur et à l'éditeur de liens

3.2.3 `lib/vector.h`

Il s'agit d'un librairie analogue à `matrix.h`. Même si les structures diffèrent, les méthodes sont quasiment identiques.

Ce fichier contient la définition de la structure `Vector` sour forme de tableau unidimentionnel. Le tableau est alloué dynamiquement. J'ai donc écrit un constructeur et un destructeur pour faciliter l'utilisation de la structure.

```
struct Vector construct_vector ( const int dim );
        /* create a null matrix */
void destruct_vector ( struct Vector *vec );
        /* free memory space */
```

Comme pour la structure `COO_Matrix`, j'ai écrit des méthodes pour pouvoir intéragir plus facilement avec la structure de données (accès à une composante, sortie écran, enregistrement dans un fichier, création d'une copie).

Pour l'algorithme du PageRank, nous avons aussi besoin de calculer la distance entre deux vecteurs :

```
double distance_vector ( const struct Vector *vec1,
                         const struct Vector *vec2 );
```

3.3 Les sources (`src/`)

Chacune des sources, contenue dans le dossier `src/`, utilise les librairies définies plus haut. Lorsqu'elles sont compilées, le `Makefile` crée un exécutable `pagerank.run` dans le dossier `bin/`.

J'ai choisi d'adopter une structure de communication maître/esclaves. À l'initiation de la session `MPI`, un processus `root` est défini et il agira en tant que maître. Il ne fait aucun calcul sur les données, il envoie seulement des paramètres et des ordres aux autres processus qui sont de fait esclaves.

3.3.1 `src/main.c`

Le fichier `src/main.c` met en place les différents paramètres de calcul.

Dans la boucle `main`, la session `MPI` est initialisée (*cf. lignes 30 à 50*) et le processus maître [3] est défini comme celui de plus grand numéro (*cf. ligne 40*).

Ensuite, les options envoyées sont parsées (*cf. lignes 51 à 119*).

`-b <buffer_size>`
Prend en argument la taille du buffer [4] ; c'est notamment utile lorsque le nom des sommets du graphe sont longs et que le parsing des fichiers `input/` échoue ;
Si non utilisée, la valeur `BUFSIZ` est choisie

`-i <input_filename>`
Prend en argument un nom de fichier, dont on calculera le pagerank du graphe ;
Si non utilisée, le nom du fichier est demandé interactivement (*cf. lignes 86 à 106*)

`-p <precision>`
Prend en argument la précision du pagerank souhaitée ;
Si non utilisée, la valeur par défaut est 10^{-6}

`-h` affiche l'aide

`-v` active le mode *verbose* [5]

Chaque processus aura besoin de connaître le nom du fichier d'entrée, on utilise donc un `MPI_Bcast` pour transmettre la variable `input_filename` (*cf. ligne 108*).

Après avoir écrit sur le disque dur tous les paramètres (*cf. lignes 120 à 139*), le calcul du pagerank est lancé (*cf. ligne 144*) :

```
compute_pagerank ( get_file_path(input_filename, INPUT), precision );
```

L'utilisateur reçoit, en fonction du résultat, un message de réussite ou un message d'échec (*cf. lignes 144 à 153*).

3. Il est sauvegardé en mémoire dans la variable globale `root` — cette forme de variable est la plus simple à utiliser, car elle est utilisée dans plusieurs fichiers du projet et la passer en argument des fonctions l'utilisant s'avère être lourd en termes d'écriture.
4. Elle est sauvegardée en mémoire dans la variable globale `buffer_size`
5. L'état *verbose* est sauvegardé en mémoire dans la variable booléenne globale `verbose`

3.3.2 `src/pagerank.c`

Le fichier `src/pagerank.c` exécute le calcul effectif du pagerank. Grâce à l'ensemble des librairies, le code est court et explicite.

La première fonction appelée est `compute_pagerank`. C'est la fonction principale. Elle renvoie `EXIT_SUCCESS` si l'algorithme a convergé, `4` sinon.

Elle appelle `load_graph` qui charge en mémoire la matrice COO transposée de la matrice d'adjacence du graphe du fichier `input_filename`. La matrice est rendue stochastique en colonnes grâce à l'appel à la fonction `normalize`.
Dans un premier temps, la matrice est chargée seulement dans le processus `root`.

La matrice est ensuite partagée entre tous les processus grâce à la fonction `distribute_matrix`.
Le processus `root` calcule dans un premier temps la taille des données à traiter par chaque processus[6] (*cf. lignes 131 à 140*).
Le processus `root` fait quatre appels `MPI_Send`, tandis que le processus esclave fait quatre appels `MPI_Recv`. Les données tranférées sont : le scalaire `nnz` avec le tag 0, le tableau `rows` avec le tag 1, le tableau `cols` avec le tag 2 et le tableau `values` avec le tag 3.

Un pré-calcul est fait pour tenir compte du *damping factor* avec la fonction `set_damping_factor`.

Une fois les données chargées en mémoire, le pagerank est calculé itérativement dans la fonction `launch_computing`. La boucle principale va de la ligne 258 à la ligne 286.
À chaque itération, tous les processus connaissent la valeur temporaire du pagerank dans la variable `pagerank`.
Chaque processus calcule le produit du pagerank par la matrice d'adjacence et le *damping factor* avec la fonction `multiply_matrix_vector` (*cf. ligne 260*). Chaque processus a donc en mémoire un vecteur dense `sub_pagerank`, de dimension la taille du

6. Le calcul est fait de manière intelligente, pour distribuer le plus également possible des données. Par exemple, s'il y a 10 données à distribuer sur 4 processus esclaves, alors les tailles calculées seront : $3 - 3 - 2 - 2$.

graphe, avec des composantes non nulles aux seules lignes dont le processus à la matrice. Le vecteur entier est donc reconstituable par un `MPI_Allreduce` avec l'opérateur `MPI_SUM` (*cf. ligne 263*).

Le terme complémentaire du *damping factor* est ajouté à la ligne 264.

À ce stade, tous les processus connaissent la valeur du pagerank à l'itération suivante dans la variable `next_pagerank`.

Le processus `root` calcule la distance entre `pagerank` et `next_pagerank`, et la communique à tous les processus (*cf. lignes 266 à 276*).

Si la distance entre les deux vecteurs est inférieure à la précision souhaitée, le calcul s'arrête (*cf. lignes 278 à 281*). Sinon, `pagerank` prend la valeur de `next_pagerank` et le calcul est réitéré (*cf. ligne 283 à 285*).

3.4 Les data (`input/` et `output/`)

3.4.1 `input/`

Le dossier `input/` contient toutes les données écrites en dur sur le disque et qui servent d'entrée aux différents programmes (*cf.* la librairie `lib/file.h`).

Il s'agit de toutes les matrices, enregistrées en respectant les contraintes suivantes :

- le fichier représente un graphe orienté, la matrice chargée en est la transposée du graphe d'adjacence
- les lignes commentées commencent par un `#`
- la première ligne contient un entier représentant le nombre de sommets du graphe
- chaque autre ligne représente un arc entre deux sommets et est de la forme : `node_name_1␣node_name_2`
- le graphe ne doit pas présenter de puit

3.4.2 `output/`

Le dossier `output/` contient toutes les données écrites en dur sur le disque en sortie des programmes (*cf.* la librairie `lib/file.h`).

Le dossier `output/` contient au moins trois fichiers :

— `settings.txt` qui contient les paramètres d'exécutions (le fichier d'entrée, la précision du calcul et l'état de sortie du programme)
— `mapping.txt` qui contient la correspondance entre les coordonnées du vecteur pagerank et le nom des nœuds (*cf.* les fichiers du dossier `input/`)
— `pagerank.txt` qui contient le vecteur pagerank calculé, dans l'ordre croissant des coordonnées

Si demandé, le dossier `output/` contient aussi une image au format PNG représentant la matrice d'entrée.

3.5 Les tests (`tests/`)

Le dossier `tests/` contient trois fichiers :

— `t_pagerank.c` qui contient tous les tests unitaires des sources `src/pagerank.h`
— `t_matrix.c` qui contient tous les tests unitaires de la librairie `lib/matrix.h`
— `t_vector.c` qui contient tous les tests unitaires de la librairie `lib/vector.h`

3.6 Codes d'erreur

J'ai suivi une convention commune à tous les fichiers de `src/` et `lib/`. Si l'exécution se passe bien, une fonction va retourner `EXIT_SUCCESS`, c'est-à-dire `0` sur la majorité des OS. Sinon :

code d'erreur 1
une erreur a été détectée sur les paramètres envoyés au programme

code d'erreur 2
une erreur a été détectée lors d'un traitement sur un fichier (ouverture, lecture, etc...)

code d'erreur 3
le programme n'a pas été lancé avec un nombre cohérent de processus (soit moins de deux, soit plus que la dimension du problème)

code d'erreur 4
l'algorithme du pagerank n'a pas convergé à la précision souhaitée après `MAX_ITERATION` itérations

Dans tous les cas, l'erreur est accompagnée d'un message explicatif pour faciliter le débuggage.

Une erreur provoque l'arrêt immédiat du processus par un `MPI_Abort` ou par un `exit` non nul, ce qui provoque en conséquence l'arrêt de tous les processus lancés par `mpirun`.

4 Résultats

4.1 Exemple de pagerank : `interstices_5`

On travaille sur le graphe suivant :

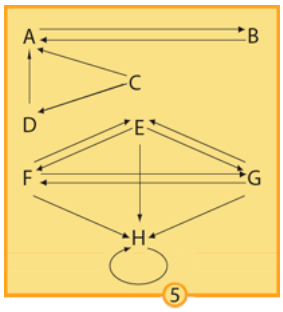

FIGURE 1 – Graphe interstices_5

On obtient le mapping :
- index 0 (coordonnée 1) $\longmapsto$ nœud A
- index 1 (coordonnée 2) $\longmapsto$ nœud B
- index 2 (coordonnée 3) $\longmapsto$ nœud C
- index 3 (coordonnée 4) $\longmapsto$ nœud D
- index 4 (coordonnée 5) $\longmapsto$ nœud E
- index 5 (coordonnée 6) $\longmapsto$ nœud F
- index 6 (coordonnée 7) $\longmapsto$ nœud G
- index 7 (coordonnée 8) $\longmapsto$ nœud H

On constate qu'il respecte l'ordre alphabétique. Il s'agit d'un hasard lié à l'ordre des arcs du fichier `input/interstices_5.txt`.

Avec un *damping factor* de $0,85$, on obtient le pagerank en 72 itérations :

$$\begin{pmatrix} 0,235557432433 \\ 0,218973817567 \\ 0,026718750000 \\ 0,018750000000 \\ 0,043269230769 \\ 0,043269230769 \\ 0,043269230769 \\ 0,370192307692 \end{pmatrix}$$

Le calcul théorique donne exactement ce pagerank avec une précision de $0,000000000001$.

4.2 Exemple de matrice creuse : `p2p-Gnutella04`

Voici un exemple de visualisation [7] de matrice creuse, représentant un graphe de 10876 nœuds et 39994 arcs. L'image totale (2a) contient si peu de points blancs qu'ils sont invisibles après redimensionnement de l'image. On les distingue sur l'extrait (2b) de celle-ci.

(a) Matrice totale 10876×10876

7. Un pixel noir représente un élément nul, et un prixel blanc représente un élément non nul.

(b) Extrait 500 × 500 de la matrice

FIGURE 2 – Matrice d'adjacence du graphe p2p-Gnutella04

5 Conclusion

5.1 Synthèse du travail accompli

Le code présenté dans le dossier est segmenté de sorte que ses différentes composantes indépendantes les unes des autres. Comme les librairies open-sources, elles pourront ainsi être réutilisées dans d'autres projets.

Il permet de répondre au cahier des charges en calculant le pagerank d'un graphe orienté, en utilisant la librairie `MPI`.

5.2 Difficultés rencontrées

Le langage C n'incluant pas la notion de surcharge de fonctions, il m'a été difficile de garder des notations claires, sans faire du code verbeux.

J'ai notamment essayé de cloisonner les librairies, en construisant des pseudo-objets à l'aide de structures et en fournissant de nombreuses fonctions s'apparentant aux méthodes du C++.

La partie sur la construction d'une image au format PNG a été la plus compliquée à mettre en place, à cause de l'absence de documentation claire. Néanmoins, je tenais à utiliser ce format d'images, notamment en raison des optimisations faites par les développeurs de la `libpng` qui permettent de stocker de grandes images sur peu d'espace mémoire.

5.3 Pour aller plus loin

Le travail présenté ici est un travail fait sur une petite échelle de temps. Il y a donc encore de nombreuses optimisations envisageables.

Du format de matrices COO que j'ai implémenté, on peut passer à un format CSR (*Compressed Sparse Row*), car le tableau `rows` est ordonné.

Il est alors aisé d'effectuer les calculs non plus sur CPU, mais sur GPU, grâce à la librairie `cuSPARSE` de la `CUDA Toolkit`. Elle fournit des kernels déjà optimisés pour la multiplication d'une matrice creuse par une vecteur dense : `cusparseDcsrmv()`. Mon implémenta-

tion a l'avantage majeur de pouvoir appeler directement la fonction `cusparseDcoo2csr()` pour calculer la matrice CSR à partir de la matrice COO.

Sans aller jusqu'à la programmation sur GPU, il est possible d'améliorer l'algorithme de répartition de la matrice sur les différents processus, pour prendre en compte que certaines lignes contiennent plus d'éléments que d'autres. Ainsi, on gagnerait en homogénéité des temps d'occupation de chaque processus.

Il est également possible d'améliorer sensiblement la fonction `draw_coo_matrix` pour prendre en compte la structure de matrice au format COO. Mais cela implique de perdre le caractère parallélisable des boucles.

Pour mesurer la performance du programme, il serait intéressant de calculer et afficher le temps de calcul du programme. Il est même envisageable d'afficher le temps de calcul de chaque étape du code (chargement des entrées, temps de chaque itération, etc...)

Enfin, il est possible de tendre vers l'algorithme utilisé en production chez Google, en calculant un pagerank logarithmique sur une échelle de 0 à 10, plutôt qu'un pagerank linéaire sur une échelle de 0 à 1.

6 Bibliographie

[1] *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Sergey Brin and Lawrence Page, 1998, `infolab.stanford.edu/~backrub/google.html`

[2] *Using your laptop to compute PageRank for millions of webpages*, Michael Nielsen, 2008, `michaelnielsen.org/blog/using-your-laptop-to-compute-pagerank-for-millions-of-webpages/`

[3] *Le theorème de Perron-Frobenius, les chaines de Markov et un célèbre moteur de recherche*, Bachir Bekka, 2007, `agreg-maths.univ-rennes1.fr/documentation/docs/Perron-Frobenius.pdf`

[4] *La formule de PageRank sur des exemples*, `interstices.info/encart.jsp?id=c_21839&encart=0&size=780,580`

[5] *A description on how to use and modify libpng*, Glenn Randers-Pehrson, 2016, `www.libpng.org/pub/png/libpng-manual.txt`

[6] *CUDA Toolkit Documentation*, `docs.nvidia.com/cuda/cusparse/`