Projet de Programmation sur Machine Parallèle

Analyse de Champ Lexical

Yann Kergutuil Rolih Meynard John Gliksberg
M1 MIHPS @ UVSQ

2014 – 2015

Sommaire

Le principal objectif du projet est de mettre en valeur des champs lexicaux dans un ensemble de corpus de texte. En effet, le travail à effectuer consiste à extraire des catégories de mots, à savoir des champs lexicaux à travers de nombreux documents. Les documents auquels on se réfère sont exclusivement des articles de Wikipédia.

Les démarches envisagées sont, pour l'essentiel, basées sur l'utilisation de matrices ayant différents paramètres. Les paramètres sont entre autres des mots, des distances et aussi des occurences. D'autres méthodes sont appliquées, notamment sur le calcul de distances et de matching par rapport à des dictionnaires de mots.

Le sujet de ce projet traite de l'analyse statistique et des méthodes de clustering.

1 L'extraction d'information

1.1 Introduction

Le coeur du projet repose sur la récupération d'information (*information retrieval*). Une des principales applications de cette méthode que l'on peut citer est la recherche de résumés d'articles scientifiques dans les bases de données. Ainsi, dans le cadre de la recherche médicale il peut être question de trouver l'ensemble des résumés d'articles dans une base de données qui traite d'un sydrôme particulier. Pour obtenir ces articles, la solution consistera à faire une requête à partir de mots clés qui sont pertinents au syndrôme. Ensuite, le système de récupération devra, d'une part, comparer la requête aux différents documents dans la base de données et, d'autre part, présenter à l'utilisateur les documents qui sont à la fois pertinents et rangés dans un ordre suivant le volume d'information contenu dans l'article.

Deux exemples concrets pour illustrer la récupération d'information :

Exemple 1.1 Requête pour la recherche dans une catégorie de résumés d'articles médicaux: *les symptômes d'une grippe*

Ainsi, tous les articles dont le contenu est lié aux symptômes

Exemple 1.2 Requête sur la recherche d'articles universitaires

Les revues sont un autre exemple d'applications de récupération d'information.

Ainsi, si nous lançons la requête suivante nous avons:

Mots clés	**Resultats**
Computer science engineering	*Nothing found*
Computing science engineering	IEEE: Computing in Science and Engineering

Table 1: Recherche d'articles universitaires

Le système de récupération d'information est censé être insensible à la moindre erreur ou le moindre changement de lettre dans la requête. Dans l'exemple 1.2, nous pouvons remarquer que la revue IEEE répond à la seconde requête. De cet exemple, nous pouvons inférer que dans beaucoup de cas la comparaison de mots peut générer des résultats inattendus.

Par ailleurs, un domaine bien connu dans la récupération d'information concerne les moteurs de recherche Web. Dans ce genre de systèmes, les requêtes sont très courtes, et de nombreux documents trouvés peuvent être des informations qui sortent du cadre du sujet attendu par

l'utilisateur. Ainsi, le classemment du resultat de la recherche est un critère d'efficacité du moteur de recherche.

1.2 Prétraitement des documents et des requêtes

Dans cette partie, il s'agit d'étudier le prétraitement qui est mis en oeuvre au préalable. Au niveau de la récupération d'information, les mots clés qui résument l'information à propos du thème d'un document sont appelés termes. De ce fait, une étape essentielle dans la mise en valeur de champs lexicaux est de créer une liste de tous les termes dans un recueil de documents, ainsi appelé index. Pour chaque terme, une liste de tous les documents contenant ce terme particulier est stockée.

1.2.1 Indexation inversée

La principale structure donnée au sein d'un système de recupération d'information s'agit de l'indexation inversée. De façon usuelle, un index inversée établit une relation entre les termes et leurs occurences dans un document donné. Nous pouvons ainsi illustrer un exemple d'index inversée:

Exemple 2.1 Un exemple d'index inversée qui représente les termes avec leurs occurences.

Terme	**Liste d'occurences**
programme	2301, 1268, ..., 74896, 47123, 741
matrice	4789, 4987, 1236, 7869, ..., 4682, 6321, 1978
parallèle	4781, 1247, 963, 4707, ..., 4136, 1247, 7841

Table 2: Dictionnaire

Dans l'exemple précédent, le dictionnaire fournit une courte liste des termes présents dans les documents, et aussi une liste correspondante aux placements de ces termes. Nous pouvons constater que l'index inversé présenté ne donne pas les identifiants des documents contenant des termes. On appelle ce type d'index un index de schéma-indépendant compte-tenu qu'aucune hypothèse n'est faite à propos du choix de stockage des documents.

Un index inversé peut être défini à partir de 4 méthodes:

`.premier(t)` retourne la première position à laquelle le terme t apparait dans l'ensemble du document

`.dernier(t)` retourne la dernière position à laquelle le terme t apparait dans l'ensemble du document

`.prochain(t, actuelle)` retourne la première occurence du terme t après la position actuelle

`.precedent(t, actuelle)` retourne la dernière occurence du terme t avant la position actuelle

Ce procédé est appelé un *inverted index*, littéralement index inversé.

Avant d'effectuer un index, deux étapes de prétraitements doivent être réalisées, à savoir l'élimination de tous les mots non pertinents et l'affichage du radical.

L'élimination de mots a pour principe de supprimer les mots qui n'apportent pas d'information sur ce document. On peut s'attendre à les trouver dans la majorité des documents — en

particulier, on peut s'attendre à ce qu'ils ne soient pas plus fréquents dans le document considéré que dans le corpus global de documents.

On peut rechercher ces types de mots en analysant en effet leur fréquence, ou exclure, sans analyse, des mots qu'on prévoit comme négligeables (par exemple les articles définis et indéfinis, les mots de liaison (*car*, *avant*, *aussi*, etc...)). Dans notre cahier des charges on doit avoir aussi peu d'à priori que possible sur le texte à analyser donc on se contente d'exclure les mots trop courts (*i.e.* moins de trois caractères) et on pourrait/devrait analyser les matrices de fréquences pour déterminer les mots sans information.

L'affichage du radical est la procédure qui vise à réduire chaque mot qui est conjugué ou qui a son propre suffixe. Par exemple,

programmation
programmable
programmer
programme

Dans le cadre de notre projet, on n'a pas effectué de détermination du radical car on traite des types de mots variés dans de nombreuses langues, ainsi les règles de construction sont trop complexes. De plus, comme c'est souvent le cas, cela pousserait dans de nombreux cas à réunir comme de la même famille des mots très différents. Par exemple,

to present	the present	il est présent
he presents	presents	live the present
a presentation	offre un présent	à présent
elle se présente		

1.3 Le modèle espace vertoriel

Le modèle espace vectoriel développe commme principale idée de créer une matrice de mot-document (`matrix_words`) dans laquelle chacun des documents est représenté par un vecteur colonne. Le vecteur a des valeurs non-nulles dans les positions qui font référence aux termes qui peuvent être présents dans les documents. Ainsi, concrètement chaque ligne détient un terme et les entrées sont nulles pour les positions correspondant aux documents contenant le terme. Chaque terme a un numéro identifiant. L'exemple ci-dessous représente une matrice de mot-document:

Document 1	Document 2	Document 3
1	2	3
4	5	6
7	8	9

Identifiants des mots:

calcul	= 1	bug	= 6
programmer	= 2	scalabilité	= 7
code	= 3	matrice	= 8
développement	= 4	vecteur	= 9
test	= 5		

Une fois la matrice de mot-document créée, il convient de calculer l'occurrence des termes dans les documents.

La matrice de mot-document que l'on a construit est pleine.

1.3.1 Requête de matching et modélisation de performance

La requête de matching consiste de trouver les documents qui sont pertinents à une requête q particulière. De manière générale, ce type de requête est réalisé à partir de la mesure distance cosinus. Par conséquent, un document $D(j)$ est considéré pertinent si l'angle entre la requête q et $D(j)$ est assez minime. Ainsi $D(j)$ est extrait si l'on a

$$cos\left[\theta(q, D(j))\right] = \frac{q^T \times D(j)}{||q||_2 \times ||D(j)||_2} > tol$$

tol est un seuil de tolérance prédéfini. Si la tolérance est réduite, les documents ont davantage retournés, ce qui signifie qu'ils sont pertinents à la requête. Cependant, le risque est qu'avec une tolérance réduite, des documents non pertinents peuvent être aussi retournés.

Dans l'extraction d'information, il est possible de mesurer la performance de la modélisation:

$$\text{Précision :} \quad P = \frac{D(r)}{D(t)}$$

$D(r)$ représente le nombre de documents pertinents extraits et $D(t)$ représente le nombre total de documents extraits.

1.4 Indexation sémantique latente

Définitions

L'analyse sémantique latente est une technique sur le traitement des langues naturelles et qui se repose sur la sémantique vectorielle.

L'indexation sémantique latente est un procédé qui effectue une relation entre un goupe de documents et les termes contenus dans ces documents.

Matrice des occurences

Dans le cadre de l'indexation sémantique latente, on est amené à construire une matrice des occurences. Cette matrice contient l'occurence des termes dans les documents. Il s'agit d'une matrice creuse.

Tous les "termes", venant de l'ensemble du corpus de texte, de la matrice sont des mots qui sont tronqués à leur radical.

2 Etat de l'art

2.1 Parallélisation de la recherche de similarité entre séquences protéiques sur GPU

Une approche pour améliorer la recherche de similarités entre protéiques a été mise en place. Celle-ci est un exemple de ce qui a déjà été développé en terme de parallélisation dans le domaine de comparaison de mots. La méthode proposée, notamment par les chercheurs Van Hoa Nguyen et Dominique Lavenier, se base sur une indexation complète des données en mémoire. Cette démarche est un moyen pour exhiber un parallélisme extrême sur des traitements simples. Chacun de ces traitements sont indépendants et s'adaptent au calcul sur GPU, Graphical Processing Units.

Les programmes en question sont:

- iBlASTP
- iBLASTN

Ces derniers ont été exécutés sur une carte NVIDIA GeForce 8800 GTX. Le travail effectué par ces deux codes sont, d'une part, la recherche de similarités entre deux blocs protéiques ou entre un bloc protéique et un génome complet. Les valeurs d'accélération obtenues s'échelonnent de 5 à 10 par rapport aux deux codes de référence dans le domaine, à savoir:

- BLASTP
- TBLASTN

2.1.1 Généralités

Dans le domaine de la bioinformatique, la recherche de similarités entre séquences génomiques est un point essentiel. L'objectif de cette recherche est de repérer des régions similaires dans des séquences d'ADN ou des séquences protéiques. Par exemple, la requête d'un bloc ayant un gène dont la nature est inconnue nécessite d'effectuer ce type de recherche de similarités. De façon générale, les résultats obtenus sont fondamentalement des segments similaires présentant une caractéristique identique élevé. Ainsi, l'information attendu doit présenter deux banques de séquence où sont précisément indiqués les couples de nucléotides (pour l'ADN) ou les couples d'acides aminés (pour les protéines).

Le développement des biotechnologies génère des données génomiques de plus en plus volumineuses. Nous pouvons citer, par exemple, le bloc d'aden GenBank qui possède plus de 180 milliards de nucléotides et sa taille est multipliée par un facteur entre 1.4 et 1.5 annuellement. Par conséquent, on constate le traitement de ces grandes quantités de données requiert davantages de temps, malgré l'amélioration des puissances des ordinateurs. En effet, d'après les calculs effectués la quantité de données en termes de nombre de nucléotides augmente plus rapidement que la puissance des machines exprimée en MIPS. Ainsi, la requête d'alignement entre séquence génomique est avant tout une requête par le contenu sur des données en vrac. Pour réaliser cette recherche, il est nécessaire de parcourir régulièrement l'ensemble des blocs, et ce de la première séquence jusqu'à la dernière séquence. De nombreux algorithmes pour faire ressortir des alignements ont été proposés. Parmis ces différents algrithmes, nous pouvons citer le pionnier Smith-Waterman qui en a réalisé un en 1981. L'algorithme de Smith-Waterman utilise des méthodes de programmation dynamique et ont une complexité quadratique. Les

autres algorithmes élaborés, principalement à partirde 1990, notamment le programme BLAST, reposent sur une méthode heuristique qui permet de localiser de petites banques identiques potentiellement pertinentes. Ces algorithmes, beaucoup plus efficace par rapport aux premiers, ont principalement été élaborés pour obtenir une requête performante dans les blocs génomiques. En revanche, leur utilisation dans le cadre d'une comparaison poussée n'est pas vraiment appropriée, car elle donne des temps d'exécution non optimisés. Cependant, pour parvenir à acclérer la recherche de ces blocs de séquence, plusieurs possibilités nous sont données, à savoir

- l'indexation des données en mémoire principale;
- la parallélisation à gros grain sur des machines de type cluster;
- la parallélisation à grain fin sur des structures spécialisées à base de VLSI/FPGA;
- la parallélisation sur des GPUs

2.1.2 Algorithme de recherche de similarités

Le coeur de l'idée se base sur une observation: la plupart des alignements significatifs ont une région de forte ressemblance se caractérisant par une suite de W caractères similaires entre les banques des deux séquences qui représentent cet alignement. Les zones dans lesquelles ces mots apparaissent sont alors repérées. Dès lors qu'un mot est trouvé sur deux portions de séquences distinctes, il est utilisé comme point d'ancrage pour déterminer un alignement plus important en essayant d'élargir, de part et d'autre, le nombre d'appariement (match) entre symboles similaires. Si l'élargissement amène à une forte similarité (déterminée en nombre de match/mismatch), sera efectué alors une dernière étape dont le principe est d'étendre encore plus l'alignement en tenant compte de l'inclusion ou de l'omission de symboles (gap).

L'exemple ci-dessous illustre comment l'alignement entre deux séquences protéiques est construit. Ainsi, le mot de 3 caractères ARV, qui apparaisse dans les deux séquences, est utilisé comme point d'ancrage. I s'agit alors de l'étape 1. A partir de ce point d'ancrage, les développeurs ont élargit de part et d'autres tant que le ration macth/mismatch est intéressant. Ceci correspond à l'étape 2. Enfin, l'équipe d'informatique a élargit de nouveau l'alignement en tenant en compte les erreurs de gap. Cette phase est l'étape 3.

Etape 1

```
K V I T A R V T G S A Q W C D N
        | | |
K L I S A R V K G S Q F C T N P
```

Étape 2

```
K V I T A R V T G S A Q W C D N
|   |   | | |   | |
K L I S A R V K G S Q F C T N P
```

Etape 3

```
K V I T A R V T G S A Q W C D N
|       | | |   | |       |   |
K L I S A R V K G S - Q F C T N P
```

Donc la recherche de similarité est une démarche qui s'effectue en 3 étapes:

- la recherche d'un point d'ancrage (étape 1)
- l'extension sans gap (étape 2)
- l'extension avec gap (étape 3)

Le programme BLAST implémente cette heuristique.

2.1.3 L'algorithme de BLASTP

L'algorithme de BLASTP prend en entrée un bloc protéique et une ou plusieurs banques de requête. L'algorithme s'effectue en 5 étapes:

Algorithme de BLASTP

```
1: indexation de la requête sur la base  - étape 1
       de mots de W caractères
2: pour tous les mots M de la banque
3:   si M appartient à la requête        - étape 2
4:     extension sans gap                - étape 3
5:   Si score >= S1
6:      extension avec gap               - étape 4
7:   Si score >= S2
8:      afficher l'alignement            - étape 5
```

Etape 1: indexation. 'équipe effectue un découpage de chaque requête en mots chevauchant de taille W. Ces mots sont alors stockés au sein d'une table de hachage

Etape 2: requête de point d'ancrage. La banque de séquence est lue séquentiellement, et ce du début jusqu'à la fin. Cette lecture prend en compte chaque mot chevauchant de W caractères. Pour chaque mot, l'équipe de développement recherche s'il apparait dans la table de hachage. Une fois qu'un mot existe, cela signifie que l'on est en présence d'un point d'ancrage.

Etape 3: extension sans gap. A partir du point d'ancrage les programmeurs ont cherché à élargir de part et d'autres afin d'entrainer une extension sans gap. De façon précise, BLASTP élabore un élargissement sans gap seulement dans le cas où deux points d'ancrage sont proches l'un de l'autre. De manière conventionnelle, la distance séparant les deux points d'ancrage doit être inférieure à 40 acides aminés. Un alignement sans gap qui inclue ces 2 points d'ancrage est alors calculé. Si le score coorespondant à cet alignement est supérieure à une valeur seuil S1.

Etape 4: extension avec gap. au cours de cette phase, les déveoppeurs ont cherché de nouveau à élargir l'alignement en tenant compte des erreurs de gap. Cette procédure est effectuée par des méthodes de programmation dynamique et se repose sur les réslultats fournis lors de l'étape précédente. Ainsi, à partir des extrémités de l'alignement sans gap, les chercheurs ont vérifié la possibilité d'inclure ou d'éliminer certains caractères. Si le score de ce nouvel alignement est supérieure à un seuil S2, alors l'étape 5 peut être effectuée.

Etape 5: visualistion des résultats. Durant cette dernière étape, BLASTP fournie progressivement une visualisation des alignements. Cettevisualisation élabore une méthode de traceback sur la configuration de données pour bien évaluer les emplacements des appariements entre synmboles.

La rapidité de BLASTP en fait son principal avantage par rapport aux techniques de programmation dynamique. Cette rapidité fournie par BLASTP est significative particulièrement lorsque la taille W du point d'ancrage est importante. En effet, plus W augmente, moins on aura à trouver un mot de cette taille et donc moins les étapes 3,4 et 5 seront exécutés. Ainsi, pour la comparaison de protéines la taille est conventionellement de 3 caractères.

Néanmoins, lorsque BLASTP est amener à traiter d'important groupe de requête, l'algorithme doit parcourir à chaque reprise la banque pour chaque séquence requête.

2.1.4 L'algorithme iBLASTP

La banque et les requêtes sont tous les deux indexés par l'algorithme de iBLASTP. La doublle indexation à ce que il n'y est plus lieu de parcourir la banque. En effet, à partir d'une structure de donnée approprié, l'ensemble des mots identiques peuvent être pointés à droite et à gauche des deux banques.

Par exemple, si un mot M est présent $n1$ fois dans la banque n1 et $n2$ fois dans la banque n2, cela implique qu'il y a $n1 \times n2$ points d'ancrage. Donc, il y a dans ce cas $n1 \times n2$ calculs d'alignements sans gap à calculer. L'algorithme de iBLASTP s'effectue aussi en 5 étapes:

Algorithme de iBLASTP

```
1: indexation des 2 banques sur la base                    - étape 1
       de mots de W caractères
2: pour chaque mot M différent
3:  construire une liste de n1 sous-séquence banque 1      - étape 2
4:   construire une liste de n2 sous-séquences banque 2
5: pour chaque sous-séquence du bloc 1
6:    pour chaque sous-séquence du bloc 2
7:       extension sans gap                                - étape 3
8:       Si score >= S1
9:          début d'extension avec gap                     - étape 4-1
10:       Si score >= S2
11:          extension complète avec gap                  - étape 4-2
12:       Si score >= S3
13:             afficher l'alignement                      - étape 5
```

Etape 1: indexation. Le principe de point d'ancrage est gardé. Cependant, contrairement à l'algorithme précédent, l'indexation est effectuée sur les deux banques. L'objectif de l'indexation est de stocker les positions des mots similaires dans une liste.

Etape 2: recherche de points d'ancrage. La recherche s'opère assez rapidement car il est question de consulter les listes établies précédemment. Une sous-séquence réfère à une instance du mot entouré de son voisinage immédiat.

Etape 3: extension sans gap. Le calcul d'extension est réalisé entre toutes les sous-séquences, c'est-à-dire $n1 \times n2$ fois.

Etape 4: cette phase est divisée en deux sous étapes. D'abord, l'espace de recherche de la programmation dynamique est réduit.

Etape 5: visualisation des résultats.

2.2 Parallélisation sur GPU

La parallélisation sur GPU de la recherche de similaraités de séquences a été réalisé sur une carte graphique NVIDIA GeForce 8800 GTX. Dans cette carte, on trouve un GPU ayant 8 multiprocesseurs SIMD. Chacun d'eux possède 16 processueurs. La programmation CUDA a été exploitée. La structure de la programmation prend enn compte une grille composée d'un ensemble de blocs qui exécutent en même temps maximum 256 threads.

2.2.1 Implémentation de iBASTP sur GPU

L'étape 3 de l'algorithme iBLASTP, extension sans gap, génère un nombre important de calculs d'élargissement sans gap $n1 \times n2$ pour un mot donné. Pour optimiser l'algorithme de iBLASTP, l'équipe de développement a eu l'idée de stoker les résultats produits par l'étape 3 et de réaliser l'étape 4-1 quand un nombre suffisant d'extension san gap a été trouvé.

Algorithme de iBLASTP-GPU

```
1 : indexation des 2 banques sur la base de mots de W caractères  { étape 1
2 : pour chaque mot M différent
3 :     construire une liste de n1 sous-séquences banque 1        { étape 2
4 :     construire une liste de n2 sous-séquences banque 2
5 :     effectuer n1 x n2 extensions sans gap sur GPU             { étape 3
6 :     stocker les extensions avec un score >= S1 dans R
7 :     si la liste R contient au moins K éléments
8 :         effectuer K début d'extensions avec gap sur GPU     { étape 4-1
9 :         si score >= S2
10:             extension complète avec gap                     { étape 4-2
11:         si score >= S3
12:             afficher l'alignement                             { étape 5
13:      Supprimer K éléments de R
```

D'après l'équipe qui a mené les tests, les résultats de l'étape 3 sont mémorisés dans la liste R. Les étapes suivantes d'exécutent uniquement lorsque la liste R a un nombre sufisant de résultats pour exploiter correctement le GPU.

2.2.2 Parallélisation de l'extension sans gap

Dans l'algorithme, il y a $n1 \times n2$ extensions sans gap qui doit être réalisé. Pour dimininuer les coûts d'échange entre l'hôte et la carte graphique, l'équipe de chercheurs a construit deux blocs de sous-séqunce: le bloc1[N,n1], avec N la longueur des sous-séquences et le bloc2[n2,N]. Par la suite, les deux blocs sont envoyés à la carte graphique pour réaliser le calcul des extensions sans gap. Ainsi, un bloc C[n1,n2] va stocker les scores résultants. D'après l'équipe qui a mené les tests, la valeur de chaque cellule [i,j] du bloc C est le score d'extension sans gap entre la sous-séquence j du bloc 1 (référant à la ligne i) et la sous-séquence i du bloc 2 (corrspondant à la colonne j).

2.2.3 Parallélisation de l'extension avec gap

Chaque K alignements avec gap va impliquer la parallélisation de 2048 tâches sur le GPU. Pour éviter tout conflit mémoire, l'équipe de développement a fait en sorte qu'en début de calcul, chaque thread charge sa paire de sous-séquence dans sa mémoire locale.

3 Conception séquentielle

3.1 Vue d'ensemble

Problème donné On part dans notre projet d'un fichier "type wikipedia" qui représente un corpus de documents.

```
[[Document 1]]
Texte, texte
[[File:filename.ext|arg]]
Texte, texte
[[Document 2]]
Plus de texte
```

But On veut déterminer les champs lexicaux de ce corpus de document.

Par ceci on veut dire déterminer des groupes de mots qui ont une utilisation (à défaut d'un sens, qu'on ne peut pas déterminer automatiquement) similaire. De même on veut pouvoir montrer comment des documents sont liés ensemble.

On réduit le problème en sous-problèmes — tout d'abord on prépare les documents, puis on crée une liste des mots avec occurences d'un document, qu'on intègre à un dictionnaire existant. Enfin on génère des matrices de distances à partir desquelles on peut trouver des clusters de mots.

Conditions On veut faire aussi peu de suppositions sur les documents et les mots que possible.

En effet on s'autorise certaines règles basiques pour reconnaître un mot et des méthodes statistiques pour faire ressortir les mots importants par leur usage mais jamais par des listes prédéfinies. Si besoin, et dans le meilleur des cas, on peut construire des listes à partir des analyses, comme par apprentissage.

Notre problème doit tenir dans la mémoire, car ce n'est pas dans le spectre du projet de gérer des accès au fichiers en parallèle ou des grandes bases de données.

3.2 Découpage des fichiers

On appelle découpage d'un fichier la création d'une liste des occurences de tous les mots qui y sont présents. L'appel correspondant à ceci est la fonction `decoupe_fichier` dans `src/lexico.c`.

On fait ce découpage en une passe unique par document où on remplit une zone de travail qu'on vide dans le tableau d'occurence ou dans un nouveau mot selon les caractères lus et une recherche dans le tableau.

On peut comme un exemple montrer la fonction `est_une_lettre_valable` qui vérifie les caractères un par un:

```
inline int est_une_lettre_valable(char c) {
    return ((c >= 'A' && c <= 'Z')   // EST UNE LETTRE MAJUSCULE
         || (c >= 'a' && c <= 'z')   // EST UNE LETTRE MINUSCULE
         ||  c == '-' || c == '\''); // EST UN HYPHEN / QUOTE
}
```

3.3 Préparation des fichiers

On crée un autre programme, `splitwiki`, qui transforme le fichier d'entrée type wikipedia en de nombreux fichiers, ce qui nous garantit 1 document = 1 fichier. On se base sur plusieurs règles syntaxiques simples donc l'exécution est très rapide.

On utilise le programme iconv pour translitérer du texte utf8 en ASCII pour garantir autant que possible et en particulier au niveau des mots 1 octet = 1 caractère.

Ces deux étapes qui doivent préceder le découpage résultent de la simplicité avec laquelle on a fait le découpage.

3.4 Ajout au dictionnaire

À chaque fois qu'un fichier est découpé en sa liste de mots on ajoute celle-ci au dictionnaire global, qui est un tableau de tableau dynamiquement alloués et réalloués. Ceci correspond à des parcours de complexité souvent n^2 où l'on doit comparer des chaînes de caractères ensemble. On réduit un peu le nombre de comparaisons en précalculant des checksums.

On va ensuite préparer les matrices de mots/documents stockées pleines 1D.

3.5 Boucle itérative de calcul des distances

En se basant l'une sur l'autre, on compare chaque mot avec chaque autre mot et chaque document avec chaque autre document pour calculer un score de distances entre les mots/documents. On devrait avoir un score borné qui représente une notion tangible mais là le calcul nous a été donné sans explication et qui ne convergait pas dans sa version originelle. On n'a pas réussi à aller plus loin que ceci.

4 Maison de la Simulation

La maison de la simulation, créée en 2011 et inauguré en janvier 2014, est l'établissement dans lequel il nous a été possible de projeter notre projet sur une machine parallèle. Ce laboratoire de recherche et de service, juxtaposé au CEA de Saclay, réunit chercheurs, ingénieurs et doctorants.

Cet amalgamme pluridisciplinaire représente l'essence-même qu'est la chaîne du HPC. C'est donc dans cet enceinte que nous avons évolué durant trois jours complets pour porter et tester notre code sur une machine multicoeur.

Cette machine est un cluster que possède l'institut du développement et des ressources en informatique scientifique (IDRIS), elle se situe sur le campus de l'université de Paris-Sud à Orsay.

Concernant ses spécifications, il s'agit d'un supercalculateur ayant entre autre 92 noeuds de calculs.

Chaque noeud est composé de 2 processeurs possédant 32 Go de mémoire vive. Les CPU de technologie Intel sont des modèles E5-2670 dont la michroarchitecture Sandy Bridge est cadencées à 2.60GHz intégrant 8 coeurs chacun. Cette machine contient aussi 4 autres noeuds couplés à des processeurs graphiques (CLGPU).

Pour pouvoir compiler et exécuter notre code nous devons au préalable nous connecter à l'une des frontales sur le site de l'IDRIS grâce à la commande `poincare`. Il est à noter que ces frontales ont la même architecture que le reste des noeuds de calcul, cela permet d'éviter le cross-compiling. Dans un premier temps il fallait copier tous nos fichiers sources en utilisant une commande basé sur le protocole scp (Secure CoPy), elle nous a donc permis d'assurer un transfert sécurisé vers la machine distante. Une fois compilé, notre code est exécuté et traité sous la forme d'un "job" ajouté dans la queue, ceci est possible en passant par un script appelé `LoadLeveler`. La gestion et l'ordonnancement de l'ensemble des exécutions sont fait par les frontales.

En plus de cette complexification de la compilation et de l'exécution d'un programme, les affichages ne se font pas directement sur notre machine, ainsi toutes les écritures sur la sortie standard sont enregistrées dans un fichier `.log` et les erreurs dans un fichiers `.err`. Quand nous avons écrit dans un fichier nous utilisons le protocole scp pour récupérer ce fichier sur notre machine.

Test de scalabilité

Nous avons testé un code de simulation (avec résolution par Lattice-Boltzman) que nous avons optimisé dans une autre matière. Nous avons ainsi pu faire un exemple de scale-up fort et faible en prenant parti d'un nombre de coeurs bien plus conséquent qu'on le peut d'habitude, avec 256 processus physiquement distribués.

Avec la scalabilité forte, on voit qu'avec notre découpage et notre schéma de communications on a un temps d'exécution optimal pour $np = 16$ avec un speed-up d'environ 10 (Jusqu'à 16 on a les temps d'exécution qui semblent proches de l'inversement linéaire).

Avec la scalabilité faible il semble que nos communications subissent très mal le scale-up de la taille totale du problème avec des temps très très lointains du constant.

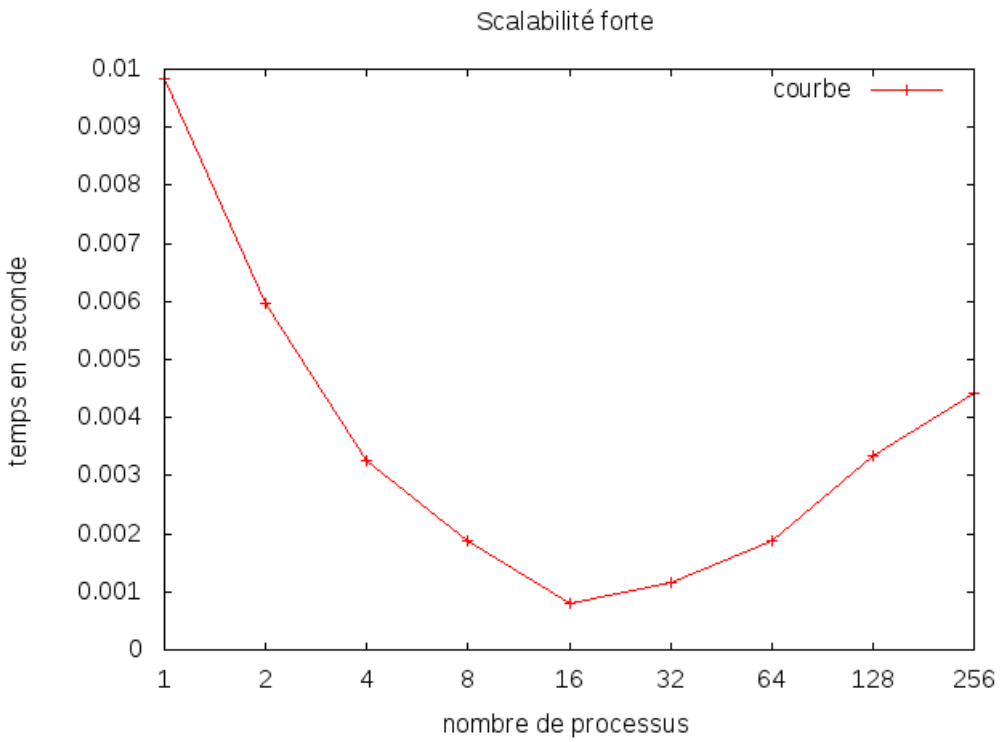

Figure 1: Scalabilité forte de Lattice Boltzman

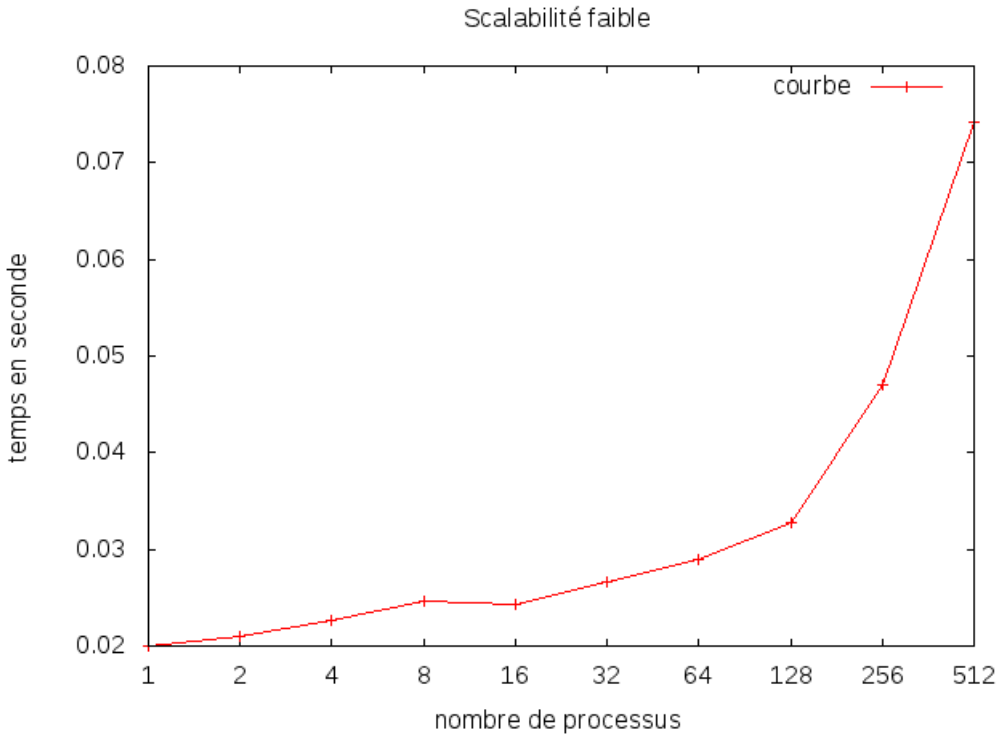

Figure 2: Scalabilité faible de Lattice Boltzman

5 Parallélisation

Notre code est intrinsèquement divisé en deux étape, la première étant l'initialisation du dictionnaire, la seconde est le calcul matriciel qui permet de converger vers un résultat.

L'initialisation est la partie qui prend le moins de temps à l'exécution, phénomène d'autant plus remarquable si la taille des documents et leur nombre augmente. Nous avons de plus remarqué qu'il était compliqué de répartir cette initialisation sur plusieurs noeuds MPI sans engendrer une graphe de communication très chargé. En effet les identifiants associés à un document ou à un mot sont créés si ce mot/documents est relevé pour la première fois dans l'initialisation. Par exemple on aura alors le cas où deux noeuds découvrent à des instants différents dans leur processus de création de dictionnaire un document. Ce document aura donc un identifiant i sur le premier noeud et un identifiant j dans le second or nous devons avoir un identifiant unique pour un élément du dictionnaire.

Nous nous sommes donc concentrés sur la parallélisation du calcul matriciel. Une fois le dictionnaire et son indexation inversée établis nous savons que les matrices de fréquence qui les représentent ne seront plus modifiées au cours de l'algorithme. Ainsi nous avons décidé de distribuer à chacun des sites ces deux matrices (*Docs* et *Words*). Notre algorithme consiste à chaque étape à comparer les documents/mots deux à deux. Afin que la charge de travail entre processus MPI soit équilibrée chaque processus devra effectuer

$$\frac{N_d \times (N_d + 1)}{P \times 2} + \frac{N_w \times (N_w + 1)}{P \times 2}$$

comparaisons à chaque étape (avec N_d le nombre total de documents, N_w le nombre total de mots et P le nombre de processus).

Figure 3: Opérations inter-documents

Pour être plus précis:

le processus 1 compare le document 1 avec les documents 1,2,3 et 4
le processus 2 compare le document 2 avec les documents 2,3 et 4
le processus 3 compare le document 2 avec les documents 3 et 4
le processus 4 compare le document 2 avec les documents 4.

Nous obtenons donc le graphe suivant:

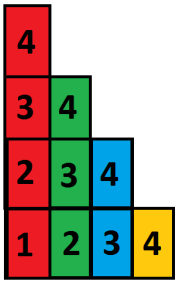

Figure 4: Charge déséquilibré

On remarque que dans notre exemple ci-dessus, si nous souhaitons le répartir sur quatres processus MPI de façon naïf, la charge de travail ne sera pas équilibré (rapport de 4 entre le processus 1 et le dernier). Il convient de répartir plus intéligement commes dans l'exemple suivant où il y a 10 documents, cela représente 55 combinaisons possible de comparaison. Ainsi nos 5 processus devront traiter exactement 11 combinaisons.

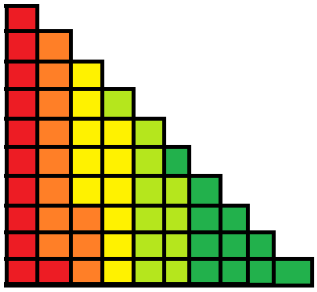

Figure 5: Charge équilibré

Une étape est constitué des deux appels de fonctions suivant :

```
Matrix_Docs  = dist_polia(Docs, Matrix_Words)
Matrix_Words = dist_polia(Words, Matrix_Docs)
```

Ici la fonction calcul une nouvealle matrice de distance à partir du matrice de fréquence et la matrice de distance complémentaire. Les processus savent donc sur quelle plage de donnée ils effectuent leurs calculs. Une étape consiste à recalculer les matrices de distances `Matrix_Docs` et `Matrix_Words`. Ainsi entre deux appels de la fonction `dist_polia` chaque processus doit recevoir les nouvelles distances que les processus restant viennent de calculer.

6 Conclusion

La première conclusion qui peut venir à l'esprit est que l'on n'a pas réussi à construire de champ lexical — on n'a même pas eu d'information sur comment analyser les matrices de distance construites.

Ça a pourtant été une très bonne occasion de mettre en place un cahier des charges très peu complet avec une liberté de conception, et de transformer un problème d'analyse de fichier en un problème d'analyse numérique qu'on peut ramener à des situations précédemment résolues.

On a été déçu d'avoir été guidé uniquement à implémenter une méthode de construction de distances sans explication, sans conception et sans indication quant à la manière d'analyser les résultats, alors que c'était le coeur de l'implémentation. Il est bien plus difficile de réorganiser une implémentation pour des performances / du parallélisme lorsqu'on doit deviner quelle méthode elle représente.

Il était instructif de découvrir ce projet avec une personne extérieure à la formation et de se rendre compte des besoins d'une entreprise.

C'était aussi l'occasion de découvrir un vrai espace de travail dans lequel prendre sa place pour tenter de réaliser de bonnes performances (nous remercions l'organisation du Master de nous avoir donné accès à ce lieu). Nous avions peu de tests à effectuer en vraie grandeur donc nous avons étudié la scalabilité d'un programme de simulation optimisé en TP. C'est dommage car c'était une de nos seules occasions de lancer un job qui pouvait vraiment passer à l'échelle.