SXS HPC : BE 1 – Architecture à mémoire partagée

Auteur : Antoine Brunet <antoine.brunet@onera.fr>
Public : SXS
Date : 2023-2024

Présentation générale

Ce BE consiste à paralléliser un algorithme de calcul sur une architecture à mémoire partagée, et à analyser la performance de cette parallélisation. Le BE se fait en binômes, et sera évalué sur les livrables suivants :

- Un rapport au format `pdf` présentant les réponses aux questions du BE.
- L'ensemble du code source produit.

Ces fichiers seront regroupés dans une archive, qui sera déposée sur LMS avant le 5 novembre 23h59 CET.

Le BE est composé de deux parties, contenant chacune des questions d'implémentation logicielle et d'analyse des résultats. Il est recommandé d'avancer au maximum l'implémentation pendant les trois heures de BE encadrées.

Partitionnement en *k-moyennes*

Le problème du partitionnement en *k-moyennes* (*k-means*) consiste à déterminer, pour un ensemble de points donné, un partitionnement de ces points minimisant la distance de chaque point à la moyenne de sa partition (ou *cluster*).

Formellement, étant donné un ensemble de n points $(p_1, \dots, p_n)$, on cherche un partitionnement de ces points en k ensembles $S = (S_1, \dots, S_k)$, en minimisant la distance entre chaque point $p_j \in S_i$ et le barycentre μ_i de la partition S_i à laquelle il appartient :

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{p_j \in S_i} \|p_j - \mu_j\|^2 \tag{1}$$

Le nombre de partition à k classes étant fini, et les distances positives, ce problème admet bien au moins une solution dans tous les cas. Cependant, on peut montrer que déterminer le partitionnement optimal est un problème NP-difficile dans le cas général.

Dans ce BE, on s'intéresse à un algorithme heuristique classique permettant de résoudre ce problème de manière approchée, appelé algorithme des *k-moyennes*, ou méthode des *k-moyennes*.

1 Méthode des *k-moyennes*

L'algorithme des *k-moyennes* fonctionne comme suit :

1. Initialisation : Choisir k points $(m_1^{(0)}, \dots, m_k^{(0)})$ qui représentent la position initiale des moyennes des partition
2. Répéter jusqu'à convergence :
   (a) Affecter chaque point à la partition dont la moyenne est la plus proche :

$$S_i^{(n)} = \left\{ p_j \;\middle|\; \|p_j - m_i^{(n-1)}\| \leq \|p_j - m_{i^*}^{(n-1)}\| \; \forall i^* \neq i \right\} \tag{2}$$

   (b) Re-calculer la moyenne de chaque partition :

$$m_i^{(n)} = \frac{1}{|S_i^{(n)}|} \sum_{p_j \in S_i^{(n)}} p_j \tag{3}$$

L'algorithme est illustré dans la figure 1.

Une implantation séquentielle en C de cet algorithme est donnée dans le fichier `src/kmeans.c`. Les différentes fonctions sont documentées dans le fichier header `include/kmeans.h`. On identifiera les trois fonctions principales suivantes :

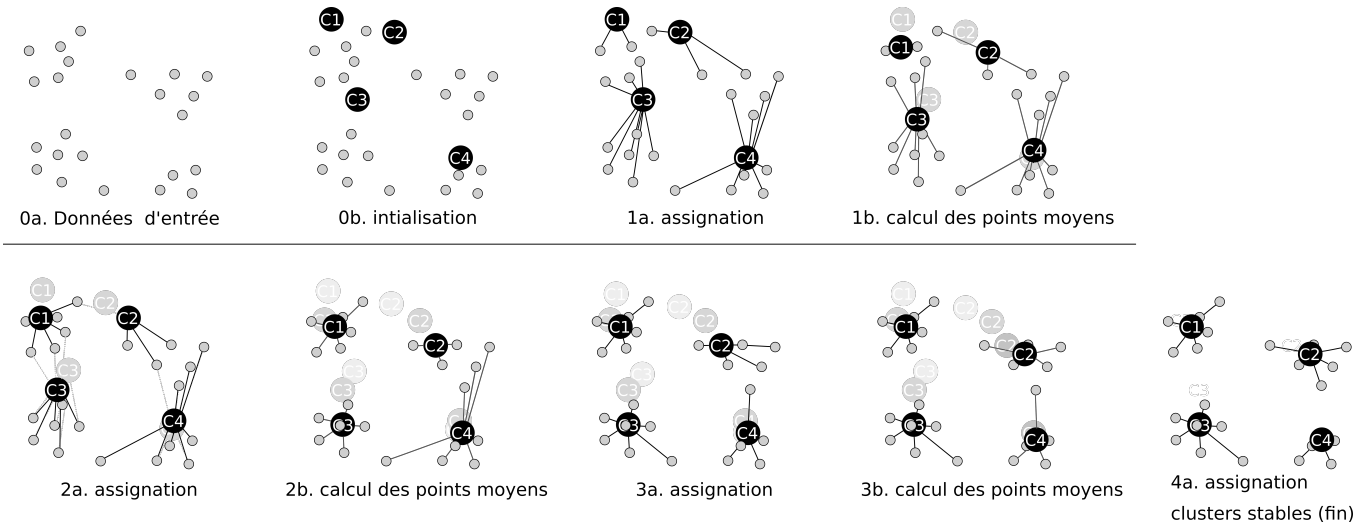

FIGURE 1 – Méthode des k-moyennes (crédit : Mquantin, fr.wikipedia.org)

FIGURE 2 – Illustration de la compression d'image par k-moyennes. À gauche, image originale, à droite l'image avec 4 couleurs.

- `kmeans` : Fonction principale du calcul de l'algorithme.
- `kmeans_assign_clusters` : Réalise l'opération (2.a) de l'algorithme, qui affecte à chaque point la partition dont la moyenne est la plus proche.
- `kmeans_compute_means` : Réalise l'opération (2.b) de l'algorithme, qui Calcule le barycentre de chaque partition.

Deux programmes d'exemples sont également donnés :

- `kmeans` (dont le codes est dans le fichier `src/main.c`) : Ce programme génère un nuages de points dans le plan 2D, et applique la méthode des *k-moyennes*.
- `compress` : Ce programme permet de compresser une image au format `png` à l'aide d'un partitionnement des couleurs des différents pixels de l'image. La couleur de chaque pixel est remplacée par la couleur de la moyenne de la partition correspondante. Le résultat de ce processus est illustré en figure 2.

Question 1.1 :

Compiler les programmes à l'aide de la commande `make`. Lire le codes des deux programmes, et les lancer. Dans la suite du BE, sauf mention du contraire, on appliquera la compression sur l'image `images/toulouse.png` avec 8 partitions de couleurs.

Question 1.2 :

Instrumenter le code de la fonction `kmeans` dans le fichier `src/kmeans.c` pour mesurer le temps cumulé passé dans chaque sous-fonction d'intérêt. Dans les cas des deux programmes fournis, quelles parties sont les plus coûteuses dans le calcul du partitionnement ? Pouvait-on s'y attendre ?

Question 1.3 :
Ces fonctions sont-elles parallélisables facilement ? Argumentez vos choix.

2 Parallélisation de l'affectation des points

Dans cette partie, on cherchera à paralléliser la fonction `kmeans_assign_clusters` à l'aide d'OpenMP.

Question 2.1 :
Proposer et implémenter une parallélisation efficace de cette fonction. On s'assurera que le comportement des programmes fournis n'est pas impacté par la parallélisation du code (en dehors de la vitesse de calcul). Décrire les difficultés rencontrées et les solutions mises en œuvre pour les résoudre. On spécifiera éventuellement les clauses `schedule` utilisées avec les directives `omp for`.

Question 2.2 :
Pour chaque programme, évaluer le facteur d'accélération pour un nombre de thread variable. On ne prendra en compte que le temps d'exécution cumulé des appels à la fonction `kmeans_assign_clusters`. Tracer et discuter les courbes d'accélération obtenues.

3 Parallélisation du calcul des barycentres

Dans cette partie, on cherchera à paralléliser la fonction `kmeans_compute_means` à l'aide d'OpenMP.

Question 3.1 :
Proposer et implémenter une parallélisation efficace de cette fonction. On s'assurera que le comportement des programmes fournis n'est pas impacté par la parallélisation du code (en dehors de la vitesse de calcul). Décrire les difficultés rencontrées et les solutions mises en œuvre pour les résoudre. On spécifiera éventuellement les clauses `schedule` utilisées avec les directives `omp for`.

Question 3.2 :
Pour chaque programme, évaluer le facteur d'accélération pour un nombre de thread variable. Tracer et discuter les courbes d'accélération obtenues.

License CC BY-NC-SA 3.0