Programmation parallèle haute performance

TP 4 : Calcul d'enveloppe convexe

Présenté par:

Boulmaali Linda Imene

Ivan Miszczuk

2023/2024

Introduction

Dans le cadre du module Programmation parallèle haute performance, notre mission consiste à aborder le calcul de l'enveloppe convexe de points générés aléatoirement. L'essence de cette tâche réside dans l'apprentissage et la maîtrise de l'outil PVM, dédié au parallélisme, afin d'implémenter divers algorithmes parallèles.

Problème

Le problème de l'enveloppe convexe consiste à trouver le plus petit polygone convexe qui englobe un ensemble de points dans le plan. Dans notre cas, nous nous intéressons à l'enveloppe convexe haute qui prend les points supérieure de l'enveloppe.

Les contraintes avec les méthodes séquentielles pour ce type de problème résident dans le fait que les algorithmes traditionnels peuvent être coûteux en termes de temps de calcul, en particulier lorsque la taille de l'ensemble de points est grande.

Le parallélisme peut réduire les coûts de ce type de problème en permettant de diviser le problème en sous-problèmes indépendants qui peuvent être résolus simultanément. Dans le cas de l'algorithme diviser et conquérir mentionné dans l'énoncé, la partition de l'ensemble de points en deux sous-ensembles peut être effectuée de manière parallèle. Ensuite, les enveloppes convexes des deux sous-ensembles peuvent être calculées en parallèle, ce qui permet de réduire le temps global nécessaire pour obtenir la solution.

Parallélisation de l’algorithme à partir de la solution séquentielle

L'algorithme ConvexeParallele est conçu pour calculer l'enveloppe convexe d'un ensemble de points de manière parallèle. (Nous avons vu l'implémentation de cet algorithme lors des TP1 et TP2 traitant du tri fusion.)

Algorithme ConvexeParallele(S)

```
// Étape 1 : Génération ou réception des points
si (pas de parent) alors
generer des points aleatoires.
sinon
recevoir des points du parent
fin si;

// Étape 2 : Calcul de l'enveloppe convexe
si |S| <= 4 alors
calculer l'enveloppe haute UH directement
sinon
// Étape 3 : Division et parallélisme
partitionner S en deux ensembles
S1 = { p1 , . . . , pn/2 } et
S2 = { pn/2+1 , . . . , pn }

// Lancer deux processus fils
lancer deux fils

// Étape 4 : Communication entre les processus
envoyer S1 au fils 1
envoyer S2 au fils 2

recevoir new_s1 de fils 1
recevoir new_s2 de fils 2

// Étape 5 : Fusion des résultats
calculer la tangente commune à new_s1 et new_s2
en déduire l'enveloppe convexe UH

fin si;

// Étape 6 : Affichage ou envoi des résultats au parent
si (pas de parent) alors
afficher l'ensemble des points restants
sinon
renvoyer S au parent
fin si;
```

L'algorithme exploite le parallélisme en divisant l'ensemble de points en deux sous-ensembles traités simultanément par des processus fils. Ces processus fils communiquent ensuite leurs résultats pour fusionner les enveloppes convexes partielles en une enveloppe convexe globale.

Les routines de transfert de liste de points

Dans cette section, nous avons formulé plusieurs hypothèses concernant le choix de la structure la plus adaptée pour les routines facilitant le transfert de listes de points entre les différents processus. Parmi ces hypothèses :

1. Envoi de la structure de liste chaînée de points.
2. Envoi d'un tableau de points.
3. Envoi de deux tableaux représentant les coordonnées x et y respectivement.
4. Envoi d'un tableau contenant les valeurs de x et y dans un même vecteur.

Analyse des hypothèses :

1. **Envoi de la structure de liste chaînée de points :**
    - La bibliothèque PVM ne permet pas directement d'envoyer des structures de données complexes telles que des listes chaînées.
2. **Envoi d'un tableau de points :**
    - Facile à gérer et peut être implémenté avec les fonctions `pvm_pkbyte` pour l'envoi et `pvm_upkbyte` pour la réception.
3. **Envoi de deux tableaux représentant les coordonnées x et y respectivement :**
    - Facile à implémenter en utilisant `pvm_pkint` pour l'envoi et `pvm_upkint` pour la réception. Cela nécessite la synchronisation entre les deux tableaux.
4. **Envoi d'un tableau contenant les valeurs de x et y dans un même vecteur :**
    - Simplifie la gestion des données en utilisant `pvm_pkint` pour l'envoi et `pvm_upkint` pour la réception, et réduit la nécessité de synchroniser deux tableaux. Cependant, cette méthode peut être moins lisible et nécessite un codage/décodage spécifique pour extraire les coordonnées x et y.

L'hypothèse à tester :

Nous optons pour la deuxième hypothèse. `pvm_pkbyte` et `pvm_upkbyte` permettent de transférer un bloc de données brut sans préoccupation de la structure interne, facilitant ainsi l'envoi et la réception de données.

Nous implémentons une fonction `list_to_array` qui sérialise la liste chaînée de points en un tableau de points, et une fonction `array_to_list` pour la désérialisation.

Ensuite, deux fonctions `send_points_array` et `receive_points_array` sont implémentées pour envoyer et recevoir respectivement un tableau de points en utilisant `pvm_pkbyte` et `pvm_upkbyte`.

Fonctionnement de `pvm_pkbyte` et `pvm_upkbyte` :

- `int info = pvm_pkbyte(char *xp, int nitem, int stride)` : Cette fonction permet de packer (empaqueter) des données brutes pour les envoyer à un autre processus.
- `int info = pvm_upkbyte(char *xp, int nitem, int stride)` : Cette fonction permet de dépaqueter des données brutes reçues d'un autre processus.
    - `nitem` : Le nombre total d'éléments à dépaqueter/empaqueter (ce n'est pas le nombre d'octets).
    - `stride` : Le pas à utiliser lors de l'empaquetage/dépaquetage des éléments. Par exemple, si `stride = 2` dans `pvm_upkcplx`, alors chaque autre nombre complexe sera dépaqueté.
    - `xp` : Pointeur vers le début d'un bloc d'octets. Peut être de n'importe quel type de données, mais doit correspondre au type de données d'empaquetage correspondant.

Mise en oeuvre d'une version parallèle maître-esclave

Comme nous l'avons vu dans le TP3, nous allons appliquer l'algorithme maître-esclave parallèle sur le problème de l'enveloppe convexe. Nous allons suivre le même cheminement en créant une structure "problème" qui permettra au maître de transmettre des problèmes aux esclaves à résoudre, en utilisant une pile.

```
struct st_pb
{
    //Type 1 : point_UH, 2 : point_merge_UH, 3 : Terminer le programme
    int type;
    point * data1;  // pointeur 1ere liste
    point * data2;  // pointeur 2eme liste

};
```

Data1 est utilisé pour stocker les points traités par les esclaves qui ont reçu le 1er et 2ème type de problèmes, Data2 est est utilisé seulement quand les esclaves reçoivent le second type de problème.

Étant donné que les points sont triés, nous pouvons supposer que si le premier et le dernier point de la liste appartiennent au résultat, et que tous les problèmes ont été traités, nous pourrons sortir de la boucle du programme maître et ainsi libérer les esclaves.

Pour savoir si tous les problèmes sont traités, il suffit de décrémenter le nombre "i" , qui initialement est égale au nombre d'esclaves, après avoir dépilé et d'incrémenter dès qu'on empile un nouveau problème.

Algorithme slave_convexe():

```
        Tantque le fils reçoit un problème à traiter faire :

            Si (problème == pb_calcul) alors
               Calculer la tangente commune;
            Sinon
               Fusionner deux listes de points; // Nous avons utiliser les fonctions merge_list
et upper_hull
            Fin Si;

               Envoyer le problème résolu au parent;

        Fin Boucle;
```

Algorithme maitre_convexe():

```
        // Génération des points
        Générer des points aléatoires;
        Initialiser la pile de problèmes;
        Création de p fils; // p est donné

        // Envoi d'un problème (de calcul) à chaque esclave
        Pour i de 0 à nb_esclave faire
           Dépiler un problème de la pile;
           Envoyer le problème à chaque esclave;
        Fin Pour;

        Tantque 1 faire

           Recevoir un problème résolu du fils;
           Changer le type de problème à traiter en problème de fusion;
           Décrémenté i ;

          Si le problème résolu contient la 1ère valeur et la dernière valeur de la liste alors
             Arrêter la boucle;
           Fin Si;

           Empiler le problème résolu;
           Dépiler un problème de la pile;

           Si le type du problème est pb_calcul alors
              Envoyer le problème au fils;
           Sinon
              pb2 = dépiler un problème de la pile;
           Si pb2 est null alors
                 Empiler le problème;
                 Incrément i;
           Sinon si le type du problème pb2 est pb_calcul alors
```

```
            Envoyer le problème à l'esclave ;
        Sinon
            Remplir data2 de pb à partir de data1 de pb2;
            Envoyer pb à l'esclave;
            libérer pb2;
        Fin Si
    Fin Si

    mettre fin aux processus esclaves;

Fin Boucle
```

Implementation

1. Pour l'algorithme de parallélisme simple, nous avons utilisé `send_points_liste` et `receive_points_liste` qui permettent d'envoyer et de recevoir des listes de points. l''implémentation de l'algorithme se trouve dans le fichier `upperParaSimple.c`
2. Pour l'algorithme Maître-esclave, nous implémentons les fonctions `receive_pb` et `send_pb`.
    - La fonction receive_pb reçoit un problème (pb_t) depuis un processus identifié par tid en utilisant PVM. Elle extrait le type du problème, les tailles des deux listes de points (data1 et data2), puis les données des listes. Elle récupère l'ID du processus émetteur (sender).
    - La fonction send_pb envoie une structure de problème (pb_t) à un processus identifié par tid en utilisant PVM. Elle envoie le type du problème, les tailles des deux listes, puis les données des listes.
    - L'implémentation de l'algorithme maître, la structure du problème(pb_t) et l'initialisation de la pile se trouvent dans le fichier master.c.
    - l'algorithme esclave se trouve dans slave.c, avec les fonctions merge_list et upper_hull, utilisés pour la partie fusion.

Vous trouverez l'implémentation des fonctions de sérialisation/désérialisation ainsi que des fonctions de transfert de données dans le fichier `point.c`.

Résultat et évaluation

Nous avons testé notre programme avec 4 esclaves.
Nous avons rajouté une condition pour empêcher le lancement du programme si le nombre de points est insuffisant pour envoyer un problème à chaque esclaves.

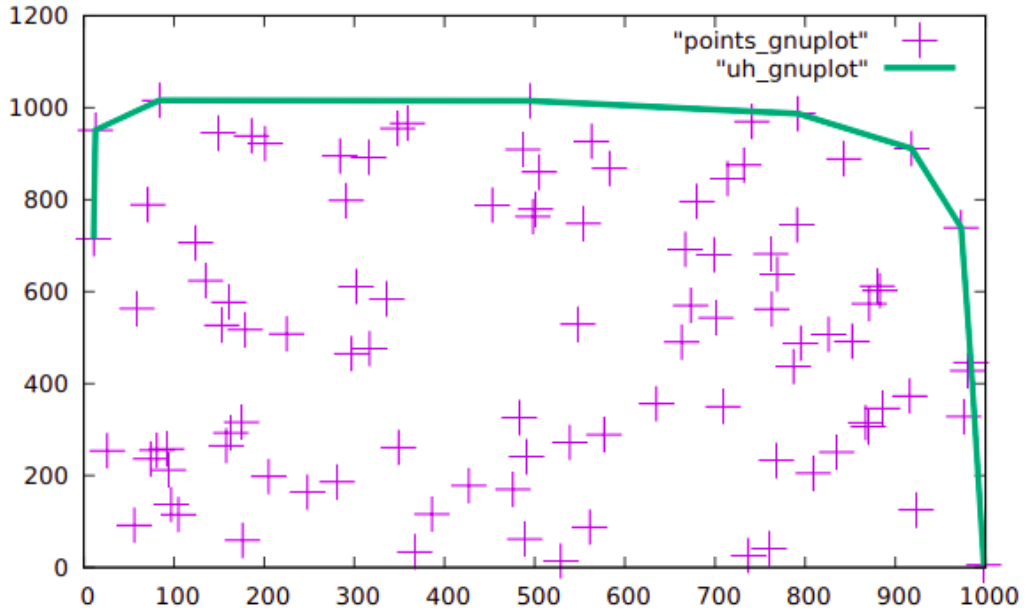

Figure 1: Courbe générée avec 100 points

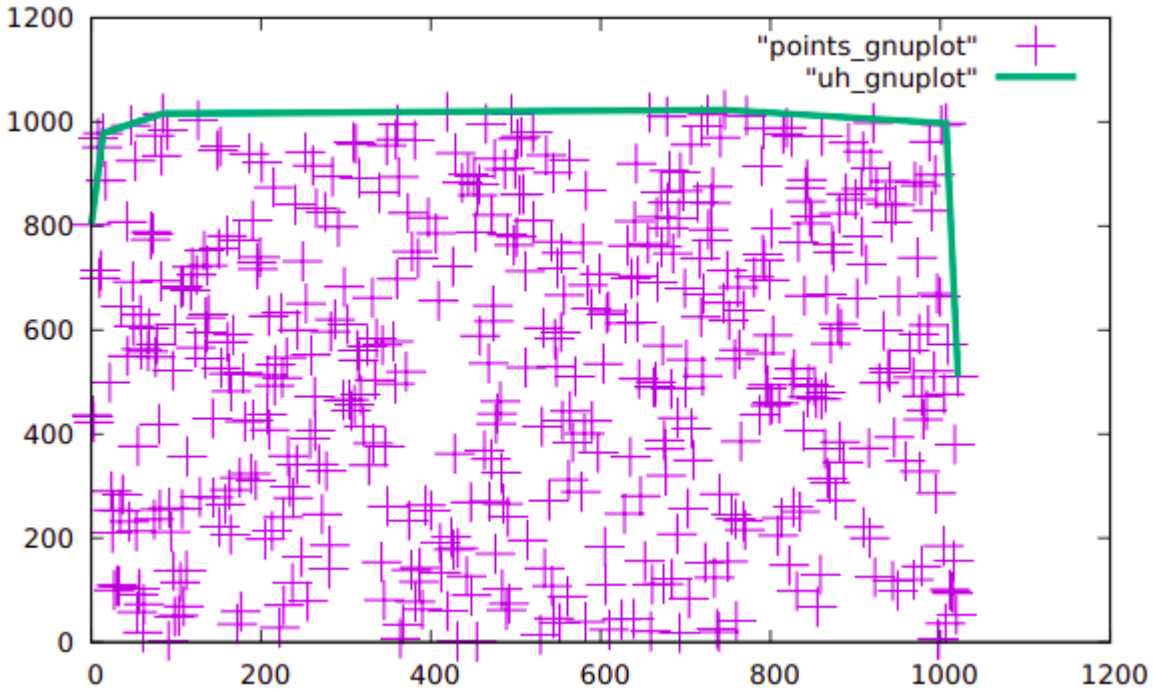

Figure 2: Courbe générée avec 500 points

Nous avons aussi évalué le temps d'exécution des deux solutions :

Solution parallèle simple	Solution parallèle maître esclave
16.94 secondes	2.25 secondes

Conclusion

Dans ce projet, nous avons abordé le problème de l'enveloppe convexe en mettant en œuvre des solutions exploitant le parallélisme pour améliorer les performances de la solution séquentielle. Nous avons réussi à mettre en place une solution parallèle simple en nous appuyant sur la version séquentielle du programme existant. Pour ce faire, nous avons utilisé une structure de tableau de points que nous avons transmise aux processus fils. Par la suite, nous avons développé une deuxième version en utilisant l'algorithme maître-esclave, où nous avons créé une structure de problème permettant au processus maître d'envoyer un problème à un esclave et de recevoir les problèmes résolus en retour. Enfin, nous avons constaté que la solution parallèle maître-esclave est beaucoup plus rapide que la solution parallèle simple.

Annexes :

/- point.c

/- point.h

/- upperOriginal.c

/- upperParaSimple.c

/- slave.c

/- master.c

/- Makefile