Université de Bretagne Occidentale
Faculté des Sciences et Technologie

Programmation parallèle haute performance
Calcul d’enveloppe convexe

LATRACH Bilal
ELAATFI Mohamed

2023-2024

Table des matières

1. Introduction :

Ce rapport rend compte de notre travail effectué lors du TP 4 sur la programmation parallèle haute performance, centré sur le calcul de l'enveloppe convexe. L'enveloppe convexe, définie comme le plus petit polygone contenant tous les points d'un ensemble donné dans le plan, est l'objectif principal de ce TP. Notre tâche consistait à élaborer un algorithme parallèle pour calculer l'enveloppe convexe haute d'un ensemble de points en appliquant une stratégie de diviser pour régner.

2. Problématique :

Notre objectif est de concevoir un algorithme parallèle visant à construire l'enveloppe convexe la plus petite possible à partir d'un ensemble de points dans un plan donné. Cette enveloppe doit garantir l'inclusion de tous les points de l'ensemble, et l'ensemble lui-même sera considéré comme convexe si tous les segments de droite définis par ses points sont entièrement contenus dans le polygone. Dans la suite de notre projet, nous présenterons les deux algorithmes parallèles que nous avons développés pour relever le défi de construire cette enveloppe convexe minimale à partir de l'ensemble de points donné sur le plan.

3. Parallélisation simple de l'algorithme:

Cet algorithme met en œuvre une stratégie de "diviser pour régner" en fonction de la taille de l'ensemble de points *S*. Le nombre de processus fils impliqués dans le calcul de l'enveloppe convexe est proportionnel à la taille de *S*.

1. Le processus principal, ne possédant pas de processus parent, crée l'ensemble de points initial S.
    - Il vérifie la taille de l'ensemble en utilisant la méthode ***point_nb***():
        - Si le nombre de points dans cet ensemble est inférieur ou égal à 4, il utilise la fonction ***point_UH***() pour calculer l'enveloppe convexe, puis affiche l'enveloppe finale. Ensuite, le programme se termine.
        - Sinon, si la taille de l'ensemble de points est supérieure à 4, l'algorithme divise l'ensemble en deux sous-ensembles égaux, attribuant un processus fils à chaque sous-ensemble. Chaque fils traite ensuite son sous-ensemble de points de manière indépendante, en répétant les mêmes étapes de vérification de la taille et de division jusqu'à ce que des ensembles de taille suffisamment petite soient obtenus pour utiliser la fonction ***point_UH***() et calculer l'enveloppe convexe. Une fois les ensembles d'enveloppe convexe calculés par les processus fils, ils sont fusionnés à l'aide de la méthode ***point_merge_UH***() pour obtenir l'enveloppe convexe finale, qui contient tous les points de l'ensemble initial, et cette enveloppe est affichée. Ensuite, le programme se termine.

2. Les processus fils reçoivent chacun un ensemble de points et utilisent la méthode ***point_nb***() pour vérifier la taille de cet ensemble.
    - Si le nombre de points dans l'ensemble est inférieur ou égal à 4, ils utilisent la fonction ***point_UH***() pour calculer l'enveloppe convexe et renvoient le résultat à leur processus parent.
    - Sinon, ils divisent l'ensemble de points en deux, créent deux processus fils et envoient à chacun la moitié de l'ensemble initial pour qu'ils continuent le traitement de manière récursive.
    - Les fils attendent de recevoir les ensembles d'enveloppe convexe calculés par leurs propres fils, fusionnent ces ensembles à l'aide de la méthode ***point_merge_UH***() et renvoient le résultat obtenu à leur processus parent.
    - Ce processus se répète jusqu'à ce que le résultat remonte au premier processus père, permettant ainsi d'obtenir l'enveloppe convexe finale.

4. Solution:

1. initialisation/Réception :

```
int nbPoints;
int tids[2];
point *pts, *pts2;

int parent = pvm_parent();
if (parent == PvmNoParent) { // Si le père ( le processus principale )

    // initialisation de l'ensemble de point utilisant la valeur passé en paramètre
    pts = point_random(atoi(argv[1]));
    point_print_gnuplot(pts, 0); //affichage des points
    nbPoints = point_nb(pts); // le nombre de points
}
else { // si les fils
    // ils reçoient les information suivant :
    pvm_recv(parent, MSG_DATA); // un message de père
    pvm_upkint(&nbPoints, 1, 1); //  le nombre de points
    int table_x [nbPoints];
    int table_y [nbPoints];
    pvm_upkint(table_x, nbPoints, 1); // les cordonnées x de toutes les points
    pvm_upkint(table_y, nbPoints, 1); // les cordonnées y de toutes les points
        //creation de l'ensemble du points utilisant les deux tableaux
    pts = creer_S(nbPoints,table_x,table_y);
   }
```

Ce bloc de code permet au processus principal de générer aléatoirement un ensemble de points dans le plan s'il n'a pas de parent, ou de recevoir les données sur les points de son parent s'il est un processus fils.

1. Vérification du Parent :
    - La fonction pvm_parent() est utilisée pour déterminer si le processus a un parent ou s'il s'agit du processus principal.
    - Si parent == PvmNoParent, cela signifie qu'il n'y a pas de parent, donc c'est le processus principal qui doit générer aléatoirement les points.

- Sinon, le processus a un parent et il doit attendre de recevoir les données sur les points.

2. Processus Principal :
   - Si le processus n'a pas de parent, cela signifie qu'il est le processus principal.
   - Il utilise la valeur passée en paramètre (argv[1]) pour déterminer le nombre de points à générer.
   - Il utilise ensuite la fonction point_random pour générer aléatoirement les points dans le plan.
   - Ces points sont affichés via la fonction point_print_gnuplot et le nombre de points est calculé avec point_nb.
3. Processus fils :
   - Si le processus a un parent, cela signifie qu'il est un processus enfant et qu'il doit recevoir les données sur les points de son parent.
   - Il utilise la fonction pvm_recv pour recevoir un message de son parent.
   - Ensuite, il utilise pvm_upkint pour déballer les données, y compris le nombre de points, les coordonnées x et les coordonnées y.
   - Une fois les données reçues, il utilise ces informations pour créer l'ensemble de points à l'aide de la fonction creer_S.

Ce bloc de code assure une communication efficace entre le processus principal et ses processus enfants, permettant ainsi de générer et de traiter les points de manière appropriée en fonction du contexte dans lequel le processus se trouve.

2. Divisé l'ensemble de points en deux sous-ensembles et la Création 2 fils

```
if (nbPoints > 4) {
    // Création de 2 fils
    pvm_spawn(BPWD "/upper_par", (char**)0, 0, "", 2, tids);
    // couper l'ensemble à deux sous ensembles
    pts2 = point_part(pts);
    int nbPoints1=point_nb(pts); //le nombre de points de la prémiere ensemble
    int nbPoints2=point_nb(pts2); //le nombre de points de la deuxiéme ensemble

    int table_x1 [nbPoints1], table_x2 [nbPoints2];
    int table_y1 [nbPoints1], table_y2 [nbPoints2];
    copier_cordonnee(pts , table_x1, table_y1);
    copier_cordonnee(pts2 , table_x2, table_y2);
```

Ce bloc de code est exécuté si le nombre de points est supérieur à 4. Il s'agit de la phase de parallélisation de l'algorithme, où le processus principal divise l'ensemble de points en deux sous-ensembles et crée deux processus enfants pour calculer les enveloppes convexes de ces sous-ensembles de manière parallèle.

1. Création de Processus Fils :
2. Division de l'Ensemble de Points :
   - La fonction point_part est utilisée pour diviser l'ensemble de points en deux sous-ensembles.

- La première moitié des points reste dans pts, tandis que la deuxième moitié est stockée dans pts2.

3. Copie des Coordonnées dans des Tableaux :
    - Les coordonnées des points dans les deux sous-ensembles sont copiées dans des tableaux distincts pour les abscisses et les ordonnées à l'aide de la fonction copier_cordonnee.

3. L'envoie des données sur les sous-ensembles

```
// Envoyer les information vers le premier fils
pvm_initsend(PvmDataDefault);
pvm_pkint(&nbPoints1, 1, 1); // envoyer le nombre de points
pvm_pkint(table_x1,nbPoints1, 1); // envoyer les x de sous ensemble
pvm_pkint(table_y1,nbPoints1, 1); // envoyer les y de sous ensemble
pvm_send(tids[0], MSG_DATA);  // envoyer vers le premier fils
// Envoyer vers le fils 2
pvm_initsend(PvmDataDefault);
pvm_pkint(&nbPoints2, 1, 1); // envoyer le nombre de points
pvm_pkint(table_x2,nbPoints2, 1); // envoyer les x de sous ensemble
pvm_pkint(table_y2,nbPoints2, 1); // envoyer les y de sous ensemble
pvm_send(tids[1], MSG_DATA); // envoyer vers le deuxième fils
```

Ce bloc de code envoie les informations sur les sous-ensembles de points aux processus enfants créés précédemment. Ces informations comprennent le nombre de points dans chaque sous-ensemble ainsi que les coordonnées x et y de chaque point.

1. Envoi des Informations au Premier Fils (tids[0]) :
    - La fonction pvm_initsend est utilisée pour initialiser l'envoi de données.
    - En utilisant pvm_pkint, le nombre de points (nbPoints1), ainsi que les coordonnées x (table_x1) et y (table_y1) du premier sous-ensemble sont empaquetés.
    - Enfin, pvm_send est utilisé pour envoyer ces données au premier fils.
2. Envoi des Informations au Deuxième Fils (tids[1]) :
    - De manière similaire, les informations sur le deuxième sous-ensemble de points sont empaquetées et envoyées au deuxième fils.

4. Recevoir les enveloppes convexes calculées

```
// reception des deux envelopes
// reception du Fils 1
pvm_recv(-1, MSG_SORT);
pvm_upkint(&nbPoints, 1, 1); // recevoir le nombre de points de sous ensemble
int tab_x1 [nbPoints];
int tab_y1 [nbPoints];
pvm_upkint(tab_x1, nbPoints, 1); // recevoir les x de sous ensemble
pvm_upkint(tab_y1, nbPoints, 1); // recevoir les y de sous ensemble
pts = creer_S(nbPoints,tab_x1,tab_y1); // construire l'ensemble de points a partir les X et les Y

// reception de la part de Fils 2
pvm_recv(-1, MSG_SORT);
pvm_upkint(&nbPoints, 1, 1); // recevoir la taille de sous ensemble
int tab_x2 [nbPoints];
int tab_y2 [nbPoints];
pvm_upkint(tab_x2, nbPoints, 1); // recevoir les x de sous ensemble
pvm_upkint(tab_y2, nbPoints, 1);// recevoir les y de sous ensemble
pts2 = creer_S(nbPoints,tab_x2,tab_y2);// construire l'ensemble de points a partir les X et les Y
// fusionner les deux sous ensebmle
point_merge_UH(pts,pts2);
```

Recevoir les enveloppes convexes calculées par les processus enfants et les fusionne pour obtenir l'enveloppe convexe globale.

1. Réception des Enveloppes Convexes :
    - Le processus principal utilise pvm_recv pour recevoir les messages des processus enfants. La valeur -1 indique qu'il peut recevoir de n'importe quel expéditeur.
    - Pour chaque processus enfant, il commence par recevoir le nombre de points de son enveloppe convexe à l'aide de pvm_upkint.
2. Construction de l'Ensemble de Points à Partir des Coordonnées Reçues :
    - Une fois les coordonnées reçues, le processus principal utilise la fonction creer_S pour construire l'ensemble de points correspondant à l'enveloppe convexe de chaque fils.
3. Fusion des Deux Ensembles de Points :
    - Après avoir construit les ensembles de points pour chaque fils, le processus principal utilise la fonction point_merge_UH pour fusionner ces ensembles et obtenir l'enveloppe convexe globale.

```
else{
    pts = point_UH(pts);
}
```

Si le nombre de points est inférieur ou égal à 4, le processus (qui peut être le processus principal ou un processus fils, en fonction du contexte dans lequel cette condition est évaluée) ne procède pas à la création de processus enfants pour diviser le travail. Au lieu de cela, il calcule directement l'enveloppe convexe des points présents dans l'ensemble en utilisant la fonction point_UH().

5. Affichage du Resultat

Si le processus en cours d'exécution est le processus principal, il utilise la fonction **point_print_gnuplot**() pour afficher les points contenus dans **pts** à l'aide de **Gnuplot**. Cela indique la fin du traitement parallèle et le passage à la visualisation des résultats.

```
if (parent == PvmNoParent) {
    point_print_gnuplot(pts, 1, "version_parallele");
}
```

Si le processus courant n'est pas le processus principal, le code entre dans la section **else** où il prépare à envoyer l'enveloppe au processus parent :

1. **Initialisation de l'envoi :** initialise l'envoi d'un message, en utilisant le mode par défaut pour la mise en forme des données.

2. **Envoi des données :** Il initialise deux tableaux d'entiers afin de conserver les coordonnées de l'ensemble des points de l'enveloppe **pts**. Ensuite, il transmet la dimension de l'enveloppe et les contenus de ces tableaux au processus parent.

```
else { // renvoi les enveloppe au parents
    pvm_initsend(PvmDataDefault);
    int taille= point_nb(pts);
    int table_x [taille];
    int table_y [taille];
    copier_cordonnee(pts , table_x, table_y); //copie_abs: permettre au tabPointsX1 de conserver les x

    pvm_pkint(&taille, 1, 1);
    pvm_pkint(table_x,taille, 1);
    pvm_pkint(table_y,taille, 1);
    pvm_send(parent, MSG_SORT);
```

6. Fonction de comparaison

```
/*
   static int compareX(i,j)

   comparaison des abscisses de 2 points
   pour la fonction qsort
*/

static int compareX(point **i, point **j)
{ return (*i)->x - (*j)->x; }
```

La fonction de comparaison conçue pour trier des points selon leurs abscisses. Cette fonction prend deux pointeurs vers des pointeurs de structure Point en entrée et renvoie un entier. Elle compare les valeurs des abscisses de ces deux points et renvoie un nombre négatif si le premier point a une

abscisse inférieure, un nombre positif si elle est supérieure, et zéro s'ils sont égaux. Cette fonction est essentielle pour garantir un ordre spécifique lors du tri d'ensembles de points selon leurs coordonnées sur l'axe des abscisses.

7. Fonction pour créer l'ensemble

```
point *creer_S(int nbPoints, int table_x[], int table_y[])
{
    point **pts;
    pts = (point **)malloc(nbPoints * sizeof(point*));
    for(int i = 0; i < nbPoints; i++){
        pts[i] = point_alloc();
        pts[i]->x = table_x[i];
        pts[i]->y = table_y[i];

    }
    qsort(pts, nbPoints, sizeof(point *), (__compar_fn_t) compareX);
    // Pour lier les points entre eux avec l'attribut next
    for(int i= 0; i < nbPoints - 1; i++){
        pts[i]->next = pts[i+1];
    }
    //retourner la premier point de l'ensemble
    return pts[0];
    //return (point *) *pts;
}
```

La fonction crée un ensemble de points dans le plan en utilisant deux tableaux de coordonnées passés en paramètres. Pour chaque paire de coordonnées fournies, un nouveau point est créé et inséré dans l'ensemble.
Les points sont liés entre eux pour former une liste chaînée, où chaque point pointe vers le suivant dans l'ordre de tri des abscisses. La fonction retourne l'adresse du premier point de l'ensemble ainsi construit.

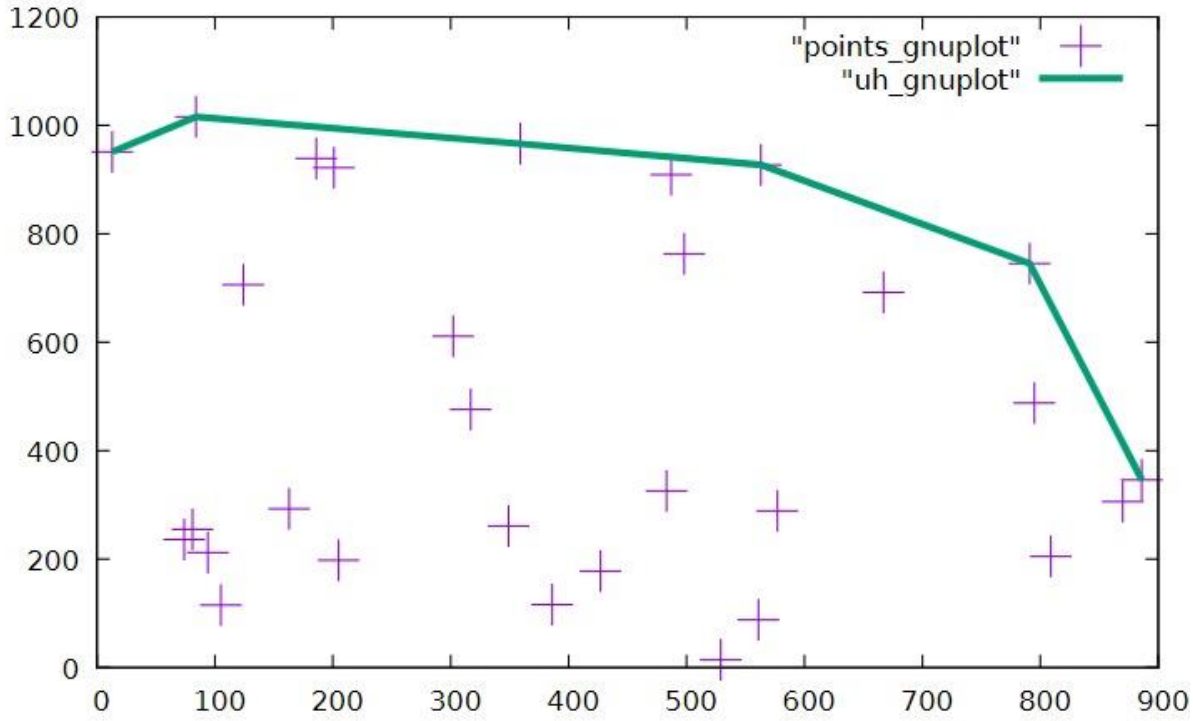
8. Fonction pour copié les coordonnée

```
void copier_cordonnee(point *pts, int table_x[], int table_y[])
{
    point *p;
    int i=0;

    for (p=pts; p!=NULL; p=p->next) {
        table_x[i] = p->x;
        table_y[i] = p->y;
        i++;
    }

}
```

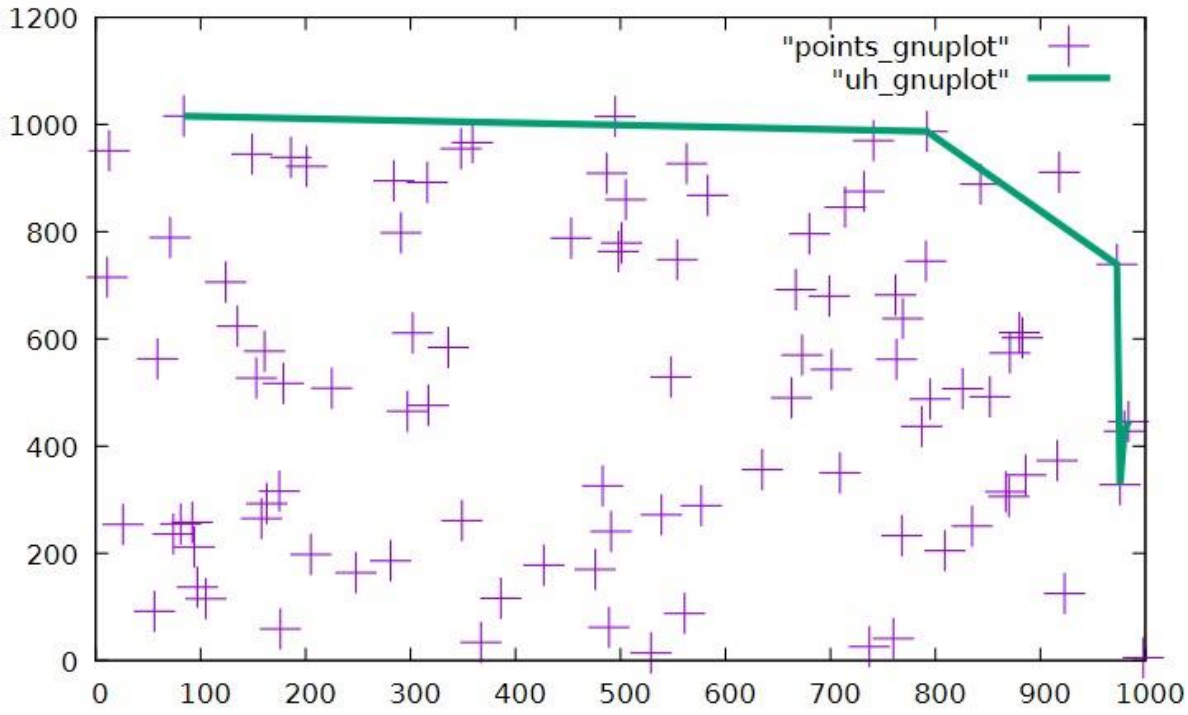
La fonction réalise la copie des coordonnées des points d'un ensemble dans deux tableaux, copiant les coordonnées x et y de chaque point dans les tableaux respectifs. Cette fonction est utile pour extraire les coordonnées des points d'un ensemble afin de les utiliser dans d'autres calculs ou opérations.

9. Résultat FINAL

Résultat du calcul d'enveloppe convexe pour 30 point choisi par hasard :



Résultat du calcul d'enveloppe convexe pour 100 point choisi par hasard :



5. Version parallèle maitre-esclave :

Cette version parallèle se compose de deux programmes : un programme maître et un programme esclave.

Partie Maître :

1. Génération d'une liste de points.
2. Initialisation de la pile de tâches.
3. Démarrage de P esclaves.
4. Attribution d'une tâche à chaque esclave.
5. Boucle continue jusqu'à ce que la taille reçue soit égale à la taille globale initiale.

Partie Esclave :

L'esclave traite les tâches en boucle jusqu'à ce qu'elles soient épuisées. Si la tâche est un calcul d'enveloppe, la fonction "calcul_pointUH" est appelée. En cas de problème de fusion, la fonction "merge_data" est appelée pour gérer la fusion. Une fois le problème résolu, l'esclave retourne les résultats au maître.

1. Solution :

a. Maitre :

```
    int i;
    int tids[P]; // Identifiants des processus esclaves
    point *pts; // Données à traiter
    pb_t *pb; // Problème actuellement traité
    int sender[1];

    pts = point_random(DATA); // Génération aléatoire des données
    init_queue(pts); // Initialisation de la pile avec les données générées

    // Création de P processus esclaves
    pvm_spawn(EPATH "/upperEsclave", (char**)0, 0, "", P, tids);

    // Distribution des problèmes aux esclaves
    for (i=0; tete>0 && i<P; i++) send_pb(tids[i], depile());

    while (1) {
        pb_t *pb2;

        // Réception d'une solution (type fusion)
        pb = receive_pb(-1, sender);
        empile(pb);

        // Vérification de la fin du traitement
        if (pb->nbPoints1 == DATA) break;

        pb = depile(); // Récupération d'un problème à traiter
        // Traitement en fonction du type de problème
    if (pb->type == PB_UH)
        // Envoi du problème à l'esclave pour calcul d'enveloppe
        send_pb(sender[0], pb);
    else { // Si le problème est de type fusion
        pb2 = depile(); // Récupération d'un deuxième problème pour fusion
        if (pb2==NULL) // Si la pile est vide
            empile(pb); // Remplacement du problème dans la pile
        else {
            if (pb2->type == PB_UH) { // Si le deuxième problème est de type calcul d'enveloppe
                empile(pb); // Remplacement du problème actuel dans la pile
                send_pb(sender[0], pb2); // Envoi du deuxième problème à l'esclave
            }
            else { // Si le deuxième problème est de type fusion
                // Fusion des deux problèmes en un seul
                pb->nbPoints2 = pb2->nbPoints1;
                pb->pts2 = pb2->pts1;
                send_pb(sender[0], pb); // Envoi du problème fusionné à l'esclave
                pb_free(pb2); // Libération de la mémoire du deuxième problème
            }
        }
    }
}

pvm_initsend(PvmDataDefault);
pvm_mcast(tids, P, MSG_END); // Notification des esclaves de la fin du traitement
point_print_gnuplot(pb->pts1, 0, "version_parallele"); // Affichage des points restants
point_print_gnuplot(pb->pts1, 1, "version_parallele");
pb_free(pb); // Libération de la mémoire du dernier problème
```

Ce code représente un programme maître conçu pour un environnement de programmation parallèle utilisant PVM . Son objectif est de coordonner l'exécution parallèle d'un algorithme de calcul d'enveloppe convexe haute (UH).
Voici un aperçu simplifié de son fonctionnement :

1. Initialisation des données : Le programme génère aléatoirement un ensemble de points dans le plan grâce à la fonction point_random(). Ces points sont ensuite organisés dans une structure de pile de problèmes à l'aide de la fonction init_queue().
2. Création des esclaves : Le maître déploie plusieurs esclaves à l'aide de la fonction pvm_spawn.
3. Répartition des tâches : Le maître envoie des problèmes à résoudre à chaque esclave en les retirant de la pile, puis en les envoyant aux esclaves avec la fonction send_pb().
4. Traitement des solutions : Le maître reçoit les solutions des esclaves, les empile, puis continue à traiter les problèmes jusqu'à ce qu'il reçoive un problème de même taille que le problème initial.
5. Fin de l'exécution : Une fois la boucle de traitement terminée, le maître envoie un message de fin à tous les esclaves, affiche l'ensemble des points restants après l'exécution de l'algorithme, puis libère la mémoire allouée avant de quitter l'environnement PVM.

En résumé, ce programme orchestre efficacement l'exécution parallèle de l'algorithme UH pour calculer l'enveloppe convexe haute d'un ensemble de points dans le plan.

b. esclave :

```c
int main()
{
    extern pb_t *receive_pb();
    int parent, sender[1]; // Identifiant du processus parent et envoyeur
    pb_t *pb; // Problème courant à traiter

    parent = pvm_parent(); // Récupération de l'identifiant du processus parent

    // Boucle pour traiter les problèmes reçus jusqu'à ce qu'il n'y en ait plus
    while ((pb = receive_pb(parent, sender)) != NULL) {
        if (pb->type == PB_UH) { // Si le problème est de type calcul d'enveloppe
            calcul_pointUH(pb); // Calcul du point d'enveloppe
        } else { // Sinon, si le problème est de type fusion
            merge_data(pb); // Fusion des données
        }
        // Envoi de la solution au processus parent
        send_pb(parent, pb);
    }

    pvm_exit(); // Terminaison propre du processus esclave
    exit(0);
}
```

Ce programme commence par obtenir l'identifiant du processus parent à l'aide de la fonction pvm_parent(). Ensuite, il entre dans une boucle while où il reçoit des problèmes à traiter à l'aide de la fonction receive_pb(). Ces problèmes sont stockés dans une variable de type pb_t. En fonction du type de problème reçu, soit un calcul d'enveloppe convexe, soit une fusion d'enveloppes, le processus esclave utilise les fonctions correspondantes pour effectuer le traitement approprié. Une fois le traitement terminé, la solution est renvoyée au processus parent à l'aide de la fonction send_pb(). Le processus esclave continue à attendre et à traiter les problèmes jusqu'à ce qu'il n'en reçoive plus.

```c
void calcul_pointUH(pb_t *pb)
{
    pb->pts1 = point_UH(pb->pts1);
    pb->type = PB_FUSION;
}

// Fonction pour fusionner les données
void merge_data(pb_t *pb)
{
    pb->nbPoints1 = pb->nbPoints1 + pb->nbPoints2;
    pb->pts1 = point_merge_UH(pb->pts1, pb->pts2);
    point_free(pb->pts2);
    pb->pts2 = NULL;
    pb->nbPoints2 = 0;
}
```

La fonction calcul_pointUH() prend en paramètre un pointeur vers une structure et effectue le calcul de l'enveloppe convexe supérieure des points contenus. Elle met à jour le pointeur pts1 avec les points de l'enveloppe calculée, et modifie le type de problème dans type pour signaler qu'il s'agit désormais d'un problème de fusion.

D'autre part, la fonction merge_data() a pour rôle de fusionner les données de deux problèmes de fusion. Elle assemble les points des deux ensembles pour former un unique ensemble de points. Ensuite, elle actualise le nombre de points dans pb->nbPoints1, libère l'espace mémoire occupé par le deuxième ensemble de points, et réinitialise les champs pb->pts2 et pb->nbPoints2 pour signaler que le deuxième ensemble de points a été fusionné avec succès.

2. La manipulation et le traitement des données :

Les fonctions **send_pb** et **receive_pb** permettent respectivement d'envoyer et de recevoir des structures de problème entre les processus. Ces fonctions prennent en charge l'empaquetage et le dépaquetage des données nécessaires pour représenter les ensembles de points.

```c
// Envoie une structure de problème à un processus spécifique
void send_pb(int tid, pb_t *pb)
{
    pvm_initsend(PvmDataDefault); // Initialise l'envoi de données
    pvm_pkint(&(pb->nbPoints1), 1, 1); // Empaquette la taille du premier ensemble de points
    pvm_pkint(&(pb->nbPoints2), 1, 1); // Empaquette la taille du deuxième ensemble de points
    pvm_pkint(&(pb->type), 1, 1); // Empaquette le type de problème
    // Préparation des données à envoyer
    int table_x[pb->nbPoints1], table_y[pb->nbPoints1];
    copier_cordonnee(pb->pts1, table_x, table_y); // Copie les coordonnées du premier ensemble de points
    pvm_pkint(table_x, pb->nbPoints1, 1); // Empaquette les coordonnées x du premier ensemble de points
    pvm_pkint(table_y, pb->nbPoints1, 1); // Empaquette les coordonnées y du premier ensemble de points
    if (0 != pb->nbPoints2){
        int table_x2[pb->nbPoints2], table_y2[pb->nbPoints2];
        copier_cordonnee(pb->pts2, table_x2, table_y2); // Copie les coordonnées du deuxième ensemble de points
        pvm_pkint(table_x2, pb->nbPoints2, 1); // Empaquette les coordonnées x du deuxième ensemble de points
        pvm_pkint(table_y2, pb->nbPoints2, 1); // Empaquette les coordonnées y du deuxième ensemble de points
    }
    // Envoi du message
    pvm_send(tid, MSG_PB);
}
```

```c
// Reçoit une structure de problème d'un processus spécifique
pb_t *receive_pb(int tid, int *sender)
{
    int tag, taille, bufid;
    bufid = pvm_recv(tid, -1); // Reçoit un message d'un processus spécifique
    pvm_bufinfo(bufid, &taille, &tag, sender); // Obtient des informations sur le message reçu
    if (tag != MSG_PB) return NULL; // Vérifie si le message est du type attendu
    pb_t *pb = pb_alloc(); // Alloue de la mémoire pour une structure de problème
    pvm_upkint(&(pb->nbPoints1), 1, 1); // Dépaquette la taille du premier ensemble de points
    pvm_upkint(&(pb->nbPoints2), 1, 1); // Dépaquette la taille du deuxième ensemble de points
    pvm_upkint(&(pb->type), 1, 1); // Dépaquette le type de problème
    pb->pts1 = point_alloc(); // Alloue de la mémoire pour le premier ensemble de points
    int table_x[pb->nbPoints1], table_y[pb->nbPoints1];
    pvm_upkint(table_x, pb->nbPoints1, 1); // Dépaquette les coordonnées x du premier ensemble de points
    pvm_upkint(table_y, pb->nbPoints1, 1); // Dépaquette les coordonnées y du premier ensemble de points
    pb->pts1 = creer_S(pb->nbPoints1, table_x, table_y); // Crée le premier ensemble de points
    if (0 != pb->nbPoints2) {
        pb->pts2 = point_alloc(); // Alloue de la mémoire pour le deuxième ensemble de points
        int table_x2[pb->nbPoints2], table_y2[pb->nbPoints2];
        pvm_upkint(table_x2, pb->nbPoints2, 1); // Dépaquette les coordonnées x du deuxième ensemble de points
        pvm_upkint(table_y2, pb->nbPoints2, 1); // Dépaquette les coordonnées y du deuxième ensemble de points
        pb->pts2 = creer_S(pb->nbPoints2, table_x2, table_y2); // Crée le deuxième ensemble de points
    }
    return pb;
}
```

Les fonctions **pb_alloc** et **pb_free** sont utilisées pour allouer et libérer la mémoire associée à une structure de problème.

```c
// Alloue de la mémoire pour une structure de problème
pb_t *pb_alloc()
{
    pb_t *pb;
    pb = (pb_t *)malloc(sizeof(pb_t)); // Alloue de la mémoire pour une structure de problème
    pb->nbPoints1 = 0; // Initialise la taille du premier ensemble de points
    pb->nbPoints2 = 0; // Initialise la taille du deuxième ensemble de points
    pb->type = 0; // Initialise le type de problème
    pb->pts1 = pb->pts2 = NULL; // Initialise les pointeurs vers les ensembles de points
    return pb;
}

// Libère la mémoire allouée pour une structure de problème
void pb_free(pb_t *pb)
{
    if (pb->pts1) point_free(pb->pts1); // Libère la mémoire allouée pour le premier ensemble de points
    if (pb->pts2) point_free(pb->pts2); // Libère la mémoire allouée pour le deuxième ensemble de points
    free(pb); // Libère la mémoire allouée pour la structure de problème
}
```

3. Résultat FINAL :

Le programme est bien implémenté et bien compilé mais il n'affiche pas le résultat attendu malheureusement

```
bilal@DESKTOP-UPINB55:~/PVM/UH/Maitre_Esclave$ ./upperMaitre 30
upperMaitre: malloc.c:2617: sysmalloc: Assertion `(old_top == initial_top (av) && old_size
== 0) || ((unsigned long) (old_size) >= MINSIZE && prev_inuse (old_top) && ((unsigned long)
 old_end & (pagesize - 1)) == 0)' failed.
Aborted
bilal@DESKTOP-UPINB55:~/PVM/UH/Maitre_Esclave$
```

6. Conclusion :

Ce projet nous a permis de mettre en pratique nos connaissances en programmation parallèle haute performance. Nous avons appris à paralléliser un algorithme récursif en utilisant une approche maître-esclave, ainsi qu'à optimiser le transfert de données entre les processus parallèles.

Actuellement, le projet a atteint un stade où l'algorithme de parallélisation est pleinement fonctionnel et sans erreur. Cependant, malgré la logique correcte du code, l'implémentation de l'algorithme Maître/Esclave a rencontré un problème inattendu. Ce contretemps nécessitera une révision approfondie pour identifier et résoudre l'erreur, afin de garantir le bon fonctionnement de cette partie cruciale du système.