

Université de Bretagne Occidentale
Faculté des Sciences et Technologie
Département d'Informatique

Année 2019-2020
Deuxième Semestre
Master première année

Programmation parallèle haute performance

Calcul d'enveloppe convexe

Réalisé par :
Khaled CHENOUF

Sommaire

I. Problématique	2
II. Parallélisation de l'algorithme séquentiel	2
1. Descriptif de l'algorithme parallèle	2
2. Descriptif de la structure de données	4
3. Explication sur le code de parallélisation	5
a) Phase numéro 1	5
b) Phase numéro 2	5
c) Phase numéro 3	8
d) Méthodes ajoutées	9
4. Résultat obtenu de cet algorithme	10
III. Version parallèle maitre-esclave	11
1. Descriptif de l'algorithme parallèle	11
a) Partie Maitre	11
b) Partie Esclave	11
2. Descriptif de la structure de données	12
3. Explication sur le code de parallélisation	13
a) Partie Maitre	13
b) Partie Esclave	15
c) Fonction d'envoi et de réception de problème	16
d) Résultat obtenu de cet algorithme	18
IV. Conclusion:	19
V. Annexe :	20

I. Problématique

La problématique de ce projet est de mettre en place un algorithme parallèle qui construit à partir d'un ensemble de points du plan une enveloppe convexe qui représente le plus petit polygone convexe contenant tous les points de l'ensemble, ce dernier est dit convexe si pour tout couple appartenant à l'ensemble de point, le segment de droite défini par le couple est entièrement contenu dans le polygone.

Dans ce qui suit, nous allons vous expliquer les deux algorithmes parallèles proposés pour résoudre cette problématique .

II. Parallélisation de l'algorithme séquentiel

1. Descriptif de l'algorithme parallèle

Cet algorithme est de type diviser et conquérir la taille du problème , tel que le nombre de processus fils qui vont contribuer au calcul de l'enveloppe convexe va dépendre de la taille de l'ensemble de points S comme suit :

- 1- Le processus initial (qui n'a pas de père) crée l'ensemble initial de point S .
- 2- Il teste la taille de l'ensemble avec la méthode **point_nb()** :
 - Si le nombre de point constituant cet ensemble est : Inférieur ou égal à 4, il fait appel à la fonction **point_UH()** qui calcul l'enveloppe convexe, puis il affiche l'enveloppe finale.
Fin de programme.
 - **Sinon** il divise l'ensemble de point sur deux , il crée deux fils et pour chaque fils il envoie $\frac{1}{2}$ de l'ensemble initial.
Il attend de recevoir deux ensembles d'enveloppe convexe calculés par ses fils , il fusionne entre ses deux enveloppes avec la méthode **point_merge_UH()**, puis il affiche l'enveloppe finale.

Fin de programme.

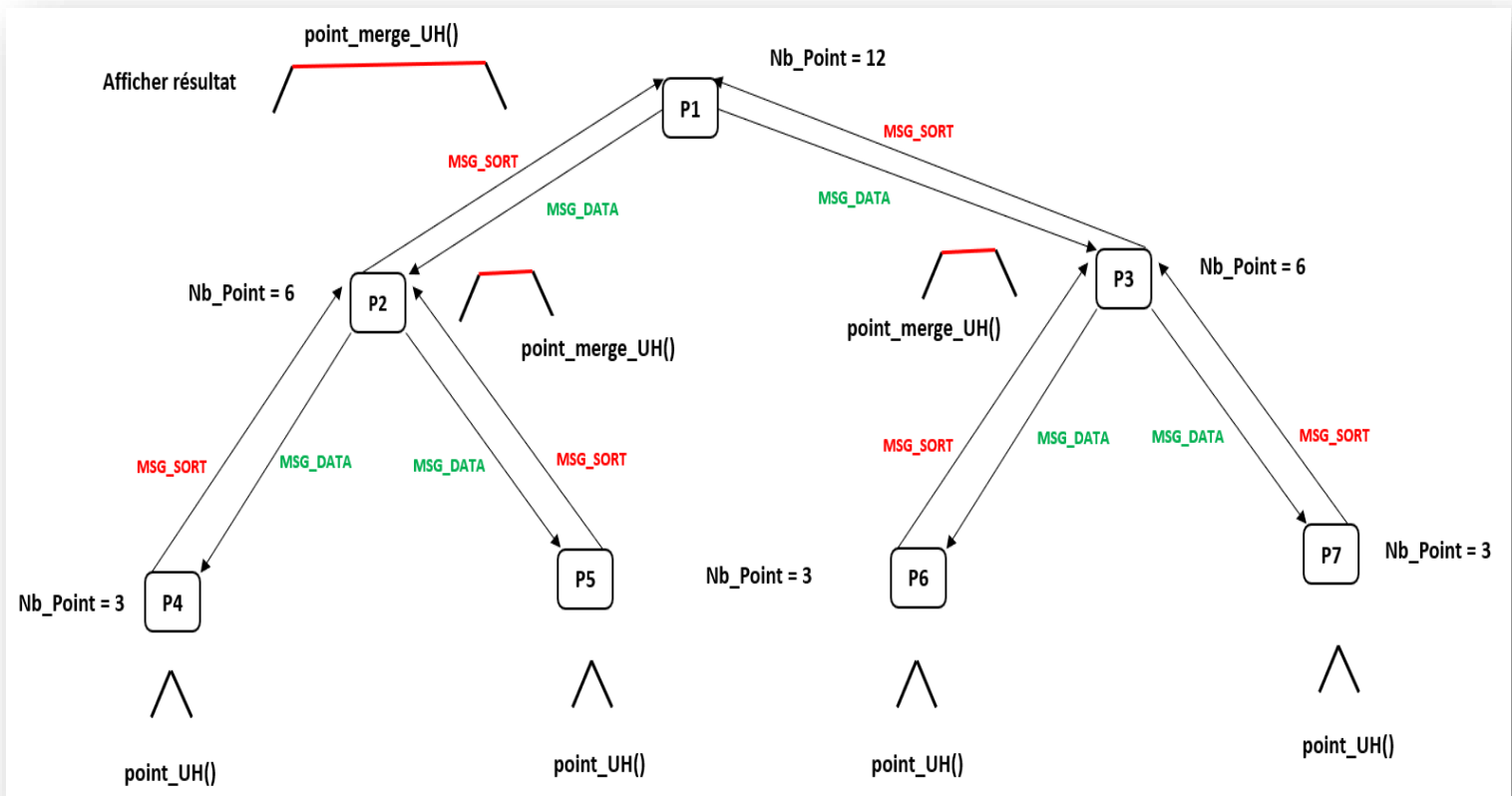
3- Les processus fils reçoivent chacun un ensemble de point, Ils testent la taille de l'ensemble reçu avec la méthode **point_nb()** :

- Si le nombre de point constituant cet ensemble est : Inferieur ou égale à 4, ils font appel à la fonction **point_UH()** qui calculent l'enveloppe convexe, et renvoie le résultat à leurs processus père.
- Sinon divisent l'ensemble de point sur deux , ils créent deux fils et pour chaque fils ils envoient $\frac{1}{2}$ de l'ensemble initial.

Ils attendent de recevoir deux ensembles d'enveloppe convexe, ils fusionnent entre ses deux enveloppes avec la méthode **point_merge_UH()**, et renvoient le résultat à leurs processus pères.

Refaire l'étape 3 jusqu'à retourner au premier processus père.

La figure ci-dessous explique le déroulement de l'algorithme :



2. Descriptif de la structure de données

Par rapport à la structure de données, nous avons une liste chaînée de couple **(x , y)** qui représente la position de l'élément de l'ensemble S dans le plan, pour pouvoir transférer les ensembles de point entre les processus père et les processus fils en utilisant l'outil **pvm** , nous avons défini par rapport à:

- **Envoie de l'ensemble** :mettre en place deux méthodes qui nous permettent à partir d'un ensemble de points de créer deux tableaux d'entiers, un pour stocker les **x** et l'autre pour stocker les **y** qui constituent l'ensemble de points.
- **Réception de l'ensemble** : mettre en place une méthode qui nous permet à partir de deux tableaux d'entiers qui représente les **x** et les **y** de créer un ensemble de point.

Au lieu de transférer un ensemble de points, nous transférons deux tableaux de type entier et qui auront la même taille que cet ensemble de points.

3. Explication sur le code de parallélisation

a) Phase numéro 1

Cette phase est composée de deux parties **if/else**, la première est pour initialiser un ensemble de points qui sera exécutée par le processus qui n'a pas de père (processus initial), la deuxième partie est pour la réception des données de type **MSG_DATA**, elle permet au processus de recevoir des données de la part de leurs parents.

```
parent = pvm_parent();

if (parent == PvmNoParent) { //initialisation

    pts = point_random(atoi(argv[1])); // initialisation d'un ensemble de point
    //point_print_gnuplot(pts, 0);
    point_print(pts, 0); //affichage de l'ensemble des points
    nbPoints = point_nb(pts); // nombre d'elements de l'ensemble

}
else { // reception des donnees

    pvm_recv(parent, MSG_DATA);
    pvm_upkint(&nbPoints, 1, 1); // Recevoir la taille de l'ensemble
    int tabX [nbPoints];
    int tabY [nbPoints];
    pvm_upkint(tabX, nbPoints, 1); // recevoir les x de sous ensemble
    pvm_upkint(tabY, nbPoints, 1); // recevoir les y de sous ensemble
    pts = Copier_Tab(nbPoints,tabX,tabY); // construire l'ensemble de points a partir les X et les Y

}
```

b) Phase numéro 2

Cette partie de condition **if** est exécutée par les processus qui ont un ensemble de point supérieur à 4 éléments où le processus va :

1. Créer deux fils avec la fonction **pvm_spawn()**, puis il divise l'ensemble de point sur 2 avec la fonction **point_part()**.
2. Créer 4 tableaux d'entier qui auront la même taille que les sous-ensembles, pour stocker les couple **(x, y)** constituant chaque sous-ensemble de type points.

3. Faire appel à la fonction **copier_X()** et **copier_Y()** qui s'occupent de copier les données des couple **(x, y)** de chaque sous-ensemble dans les tableaux créés avant.
4. Créer deux variable **taille1** et **taille2** de type entier afin de stocker les tailles de sous-ensembles. Il fait appel à la fonction **point_nb()** pour retourner la taille de sous-ensemble.

```
if (nbPoints > 4) {
    // créer 2 fils
    pvm_spawn(BPWD "/upper", (char**)0, 0, "", 2, tids);
    // partitionner l'ensemble
    pts2 = point_part(pts);
    // declaration de tableau de stockage
    int tabPointsX1 [point_nb(pts)] ;
    int tabPointsY2 [point_nb(pts)] ;
    int tabPointsX3 [point_nb(pts2)] ;
    int tabPointsY4 [point_nb(pts2)] ;
    // stocker les x et y dans des tableaux d'entiers,
    // ensemble gauche de points
    Copier_X(pts , tabPointsX1);
    Copier_Y(pts , tabPointsY2);
    // ensemble droite de points
    Copier_X(pts2 , tabPointsX3);
    Copier_Y(pts2 , tabPointsY4);

    int taille1=point_nb(pts);
    int taille2=point_nb(pts2);
}
```

5. Envoie pour chacun de ses deux fils la taille de sous ensemble, ainsi que les deux tableaux d'entier qui stockent les valeurs de x et y de ce dernier.

```
// envoi de l'ensemble de points

// Fils 1
pvm_initsend(PvmDataDefault);
pvm_pkint(&taille1, 1, 1); // envoyer la taille de sous ensemble
pvm_pkint(tabPointsX1, taille1, 1); // envoyer les x de sous ensemble
pvm_pkint(tabPointsY2, taille1, 1); // envoyer les y de sous ensemble
pvm_send(tids[0], MSG_DATA);

// Fils 2
pvm_initsend(PvmDataDefault);
pvm_pkint(&taille2, 1, 1); // envoyer la taille de sous ensemble
pvm_pkint(tabPointsX3, taille2, 1); // envoyer les x de sous ensemble
pvm_pkint(tabPointsY4, taille2, 1); // envoyer les y de sous ensemble
pvm_send(tids[1], MSG_DATA);
```

6. Attendre la réception de deux enveloppes de la part de ses deux fils, pour cela il reçoit la taille de l'ensemble et le maître dans **nbPoints**, il crée deux tableaux d'entier de taille **nbPoints**, il reçoit les données des x et y dans les deux tableaux créés au par avant puis il fait appel à la fonction **copier_Tab()**, qui s'occupe à partir de deux tableaux d'entiers de créer un ensemble de point.

```
// reception de deux enveloppes

// Fils 1
pvm_recv(tids[0], MSG_SORT);
pvm_upkint(&nbPoints, 1, 1); // recevoir la taille de sous ensemble
int tabX1 [nbPoints];
int tabY1 [nbPoints];
pvm_upkint(tabX1, nbPoints, 1); // recevoir les x de sous ensemble
pvm_upkint(tabY1, nbPoints, 1); // recevoir les y de sous ensemble
pts = Copier_Tab(nbPoints,tabX1,tabY1); // construire l'ensemble de points a partir les X et les Y

// Fils 2
pvm_recv(tids[1], MSG_SORT);
pvm_upkint(&nbPoints, 1, 1); // recevoir la taille de sous ensemble
int tabX2 [nbPoints];
int tabY2 [nbPoints];
pvm_upkint(tabX2, nbPoints, 1); // recevoir les x de sous ensemble
pvm_upkint(tabY2, nbPoints, 1); // recevoir les y de sous ensemble
pts2 = Copier_Tab(nbPoints,tabX2,tabY2); // construire l'ensemble de points a partir les X et les Y
```

7. Le processus père qui a reçu deux ensembles d'enveloppe de la part de ses fils doit appeler la fonction **point_merge_UH()** pour les fusionner.

```
// fusion les deux resultats
point_merge_UH(pts,pts2);

}
```


Cette partie de **else** est exécutée par les processus qui ont reçu un ensemble de points de taille inférieur ou égal à 4 éléments, ils font appel à la fonction **point_UH()** pour calculer l'enveloppe convexe.

```
else
    // Calculer l'enveloppe haute directement
    pts = point_UH(pts);
```

c) Phase numéro 3

Cette phase est constituée de deux parties de condition **if/else**, définie pour renvoyer les résultats d'enveloppe aux parents, du coup :

- 1- **Si** le processus est le père qui a créé l'ensemble initiale, il affiche directement le résultat en faisant appel à la fonction **point_print()** ou **point_print_gnuplot()** ;
- 2- **Sinon** il crée deux tableaux d'entiers pour stocker les couple (**x** , **y**) constituant l'ensemble de l'enveloppe **pts**, ensuite il envoie la taille de l'enveloppe ainsi que ses deux tableaux d'entier à son père.

```
if (parent == PvmNoParent) { // Affichage Resultat
    //point_print_gnuplot(pts, 1);
    point_print(pts, 1);
}
else { // renvoi les enveloppe au parents
    pvm_initsend(PvmDataDefault);
    int tabPointsX1 [point_nb(pts)] ;
    int tabPointsY2 [point_nb(pts)] ;
    int taille3=point_nb(pts);
    Copier_X(pts , tabPointsX1);
    Copier_Y(pts , tabPointsY2);
    pvm_pkint(&taille3, 1, 1); // envoyer la taille de l'ensemble qui construit l'enveloppe haute
    pvm_pkint(tabPointsX1,taille3, 1); // envoyer les X de l'ensemble qui construit l'enveloppe haute
    pvm_pkint(tabPointsY2,taille3, 1); // envoyer les Y de l'ensemble qui construit l'enveloppe haute
    pvm_send(parent, MSG_SORT);
}
```

d) Méthodes ajoutées

Les deux méthodes **copier_X()** et **copier_Y()** prennent en paramètre , un ensemble de type points et un tableau d'entiers. Elles s'occupent de copier les couple X et Y dans des tableaux d'entiers.

```
void Copier_X(pts,tab)
point *pts;
int tab[point_nb(pts)];
{
    point *pt1;
    int i=0;

    for (pt1=pts; pt1!=NULL; pt1=pt1->next) {
        tab[i]=pt1->x;
        i++;
    }
}
```

```
void Copier_Y(pts,tab)
point *pts;
int tab[point_nb(pts)];
{
    point *pt1;
    int i=0;

    for (pt1=pts; pt1!=NULL; pt1=pt1->next) {
        tab[i]=pt1->y;
        i++;
    }
}
```

La méthode **copier_Tab()** prend en paramètre le nombre de couple (**x , y**), et deux tableaux de type entiers, le premier pour stocker les valeurs de X et le deuxième pour stocker les valeurs de Y. Elle s'occupe de crée un ensemble de type point à partir de deux tableaux d'entiers.

```

point *Copier_Tab(nbPts,tabX,tabY)
{
    int nbPts;
    int tabX[nbPts];
    int tabY[nbPts];
    {
        int i,j=0;
        point **pts;

        pts = (point **)malloc(nbPts*sizeof(point *));
        for(i=0; i < nbPts; i++) {
            pts[i] = point_alloc();
            pts[i]->x = tabX[j];
            pts[i]->y = tabY[j];
            j++;
        }

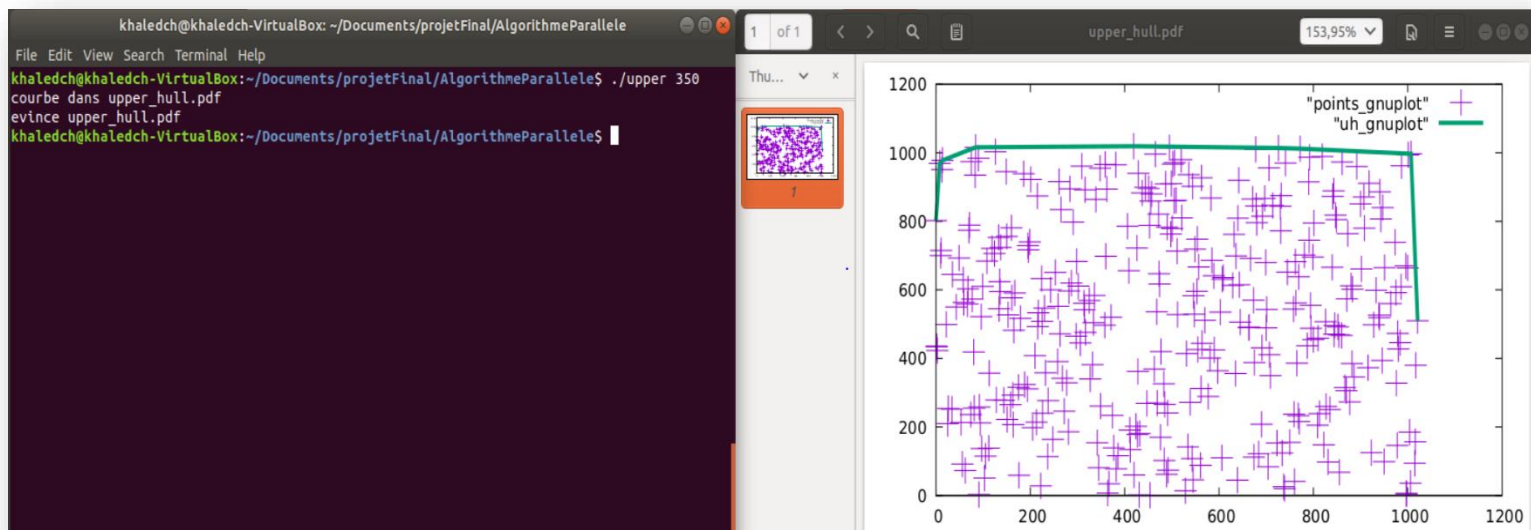
        qsort(pts, nbPts, sizeof(point *), compareX);
        for (i=0; i<nbPts-1; i++)
            pts[i]->next = pts[i+1];

        return (point *)*pts;
    }
}

```

4. Résultat obtenu de cet algorithme

L'exécution de 'upper' avec un argument qui représente le nombre de points dans l'ensemble crée un fichier upper_hull.pdf, qui contient exactement le résultat attendu avec n'importe quel nombre de points comme le montre la figure ci-dessous avec un nombre de points égal 350.



III. Version parallèle maitre-esclave

1. Descriptif de l'algorithme parallèle

Cette version parallèle est constituée de deux programmes. Un programme maitre et un programme esclave.

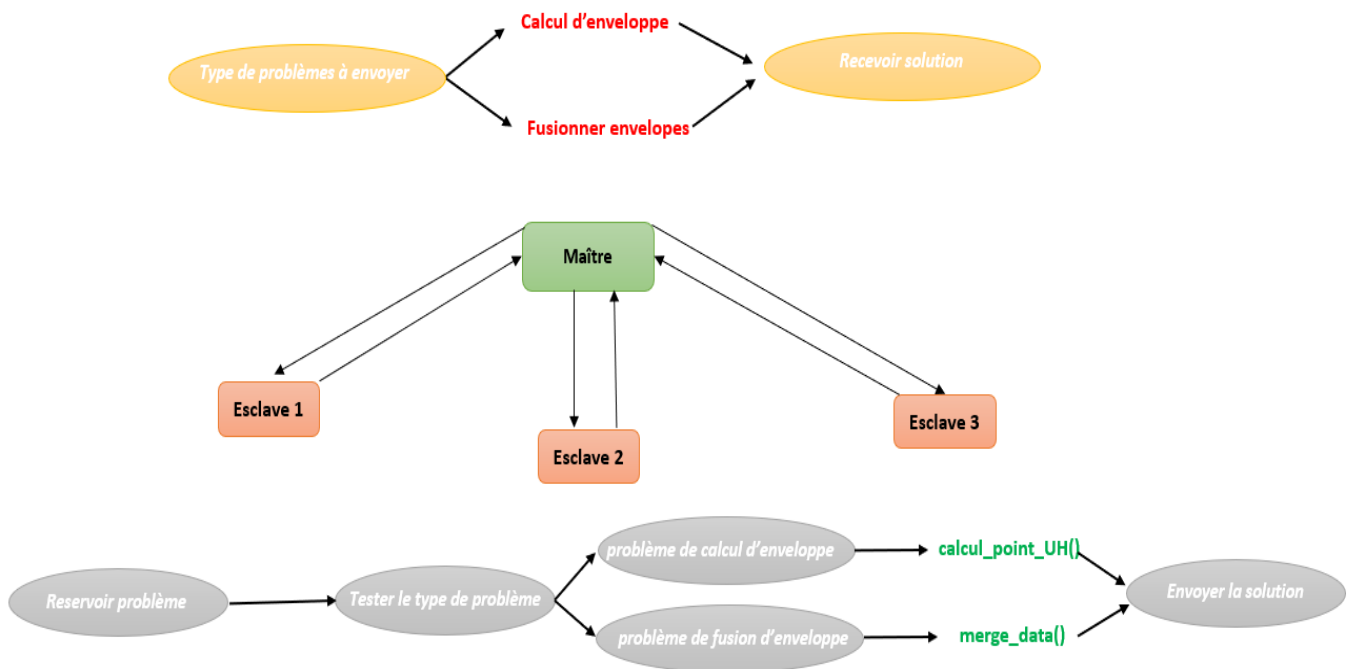
a) Partie Maitre

1. Création d'une liste de point contient un nombre d'éléments égal à **DATA**.
2. Initialisation de la pile des problèmes (plus de détails dans la partie suivante).
3. Lancement de P esclaves.
4. Envoyer un problème à chaque esclave.
5. Dans une boucle infinie :
 - a. Réception du problème traité envoyé par un esclave, l'empiler, vérifier sa taille, son type et choisir la suite de traitement en fonction de ces paramètres.
 - b. Dépiler un autre problème puis tester son type. S'il est un problème de :
 - Calcul d'enveloppe on l'envoie directement à l'esclave.
 - Fusion, dépiler un autre problème de même type et l'envoyer à l'esclave pour la fusion.
6. Reprendre depuis l'étape 5.

b) Partie Esclave

1. Dans une boucle, l'esclave continue à recevoir des problèmes, tant qu'il y en a.
 - Si le problème est de type calcul d'enveloppe, appeler la fonction **calcul_pointUH**.
 - Si le problème est de type fusion, appeler la fonction **merge_data**, qui s'occupe de la fusion.
2. Renvoyer le problème traité au maitre.

La figure ci-dessous illustre le déroulement de l'algorithme :



2. Descriptif de la structure de données

Pour le bon déroulement de l'algorithme nous avons choisi une pile de problèmes (un tableau en C avec un indice qui change).

Chaque case de cette de pile est une partie du problème de taille ($PB = DATA/N$), tel que :

- DATA : représente le nombre de points initial (Fixé au début de l'algorithme).
- N : il est égal à 4, il represente le nombre de point qui peut être traité en une fois par la fonction **calcul_pointUH()** de l'esclave.

Chaque case de problème contient une structure en C de type **pb_t**, comme la montre la figure ci-dessous :

```
/* structure de probleme */
struct st_pb {
    int taille1, taille2;
    int type;
    point *pts1;
    point *pts2;
};

typedef struct st_pb pb_t;
```

Cette structure est composée de plusieurs éléments tel que :

- **taille1 / taille2** : représente respectivement la taille de l'ensemble pts1 et pts2.
- **pts1** : représente une liste chaine des points de l'enveloppe.
- **type** : représente le type de problème à traité, y'en a deux types, problème de calcul d'enveloppe et problème de fusion de deux enveloppes.
- **pts2** : représente une liste chaine des points de l'enveloppe, elle est non null si le problème est de type fusion.

3. Explication sur le code de parallélisation

a) *Partie Maitre*

Le maitre initialise l'ensemble de point avec la fonction **point_random()**, il met l'ensemble créé dans une pile en faisant l'appel à la fonction **init_queue()**.

Il lance les esclaves en utilisant l'outil **pvm_spawn**, ensuite il envoie à tous les esclaves des problèmes en dépilant le problème de la pile avec la fonction **depile()** qui retourne un problème depuis la pile et en envoyant ce dernier avec la fonction **send_pb()**.

Dans une boucle while :

Il reçoit une solution avec la fonction **receive_pb()** qu'il va l'empiler, ensuite il dépile un autre problème qu'il teste son type, si ce dernier est de type calcul d'enveloppe il l'envoie

aux esclaves pour le résoudre sinon s'il est de type fusion il doit dépiler un autre problème de même type pour la fusion des deux enveloppes.

La boucle while s'arrête quand il reçoit un problème de même taille que le problème initial DATA.

```
pts = point_random(DATA);          /* initialisation aleatoire */
init_queue(pts);                   /* initialisation de la pile */

/* lancement des P esclaves */
pvm_spawn(EPATH "/uhs", (char**)0, 0, "", P, tids);

/* envoi d'un probleme (UH) a chaque esclave */
for (i=0; Q_nb>0 && i<P; i++) send_pb(tids[i], depile());

while (1) {
    pb_t *pb2;

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

    /* dernier probleme ? */
    if (pb->taille1 == DATA)
        break;

    pb = depile();
    if (pb->type == PB_UH)
        send_pb(*sender, pb);
    else { // PB_FUS
        pb2 = depile(); /* 2eme pb pour fusion ... */
        if (!pb2) {
            empile(pb); // rien a faire
        }
        else {
            if (pb2->type == PB_FUS) { /* on fusionne pb et pb2 */
                pb->taille2 = pb2->taille1;
                pb->pts2 = pb2->pts1;
                send_pb(*sender, pb); /* envoi du probleme a l'esclave */
                pb_free(pb2);
            }
            else { // PB_UH
                empile(pb);
                send_pb(*sender, pb2); /* envoi du probleme a l'esclave */
            }
        }
    }
}
}
```

b) Partie Esclave

L'esclave reçoit des problèmes avec la fonction **receive_pb()**, il stock le problème dans une variable de type **pb_t** qui représente le problème courant. En fonction de type de problème reçu il fait appel à la fonction de calcul d'enveloppe ou de fusions des enveloppes après il renvoie la solution au maître avec la fonction **send_pb()**.

```
extern pb_t *receive_pb();
int parent, sender[1]; /* pere et envoyeur (non utilise) */
pb_t *pb;               /* probleme courant */

parent = pvm_parent();

/* tant que l'on recoit un probleme a traiter ... */
while ((pb = receive_pb(parent, sender)) != NULL) {
    if (pb->type == PB_UH) /* on traite suivant le cas */
        calcul_pointUH(pb);
    else
        merge_data(pb);
    /* et on revoie la solution */
    send_pb(parent, pb);
}

pvm_exit();
exit(0);
```

La fonction ci-dessous représente la fonction de calcul d'enveloppe, elle prend en paramètre un problème de type **pb_t**, elle calcul l'enveloppe en faisant appel à la fonction **point_UH()**, ensuite elle change le type de problème à un problème de fusion.

```
/* calcul enveloppe de 4 point */

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


La fonction ci-dessous représente la fonction de fusion de deux enveloppes, elle prend en paramètre un problème de type **pb_t**, elle incrémente la taille de problème ,ensuite elle merge entre deux enveloppes en faisant appel à la fonction **point_merge_UH()**, ensuite elle libère le deuxième ensemble **pts2** qui servait à stocker le deuxième ensemble de points constituant l'enveloppe.

```
/*
 * fusion
 */

void merge_data(pb)
    pb_t *pb;
{
    pb->taille1 = pb->taille1 + pb->taille2;
    pb->pts1 = point_merge_UH(pb->pts1,pb->pts2);
    point_free(pb->pts2);
    pb->pts2 = NULL;
    pb->taille2 = 0;
}
```

c) Fonction d'envoi et de réception de problème

Les deux fonctions suivantes **send_pb**, **receive_pb** s'occupent de l'envoi et de la réception des problèmes c'est-à-dire la communication entre le maitre et l'esclave, et qui sont spécialement adaptées pour la structure de type **pb_t** :

***send_pb** : elle s'occupe de :

- Envoi d'un problème a un processus tid.

- Le problème est désalloué localement.

```
void send_pb(int tid, pb_t *pb)
{
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&(pb->taille1), 1, 1);
    pvm_pkint(&(pb->taille2), 1, 1);
    pvm_pkint(&(pb->type), 1, 1);
    pvm_pkint(&(pb->pts1->x), pb->taille1, 1);
    pvm_pkint(&(pb->pts1->y), pb->taille1, 1);
    if (pb->taille2 > 0){
        pvm_pkint(&(pb->pts2->x), pb->taille2, 1);
        pvm_pkint(&(pb->pts2->y), pb->taille2, 1);
    }
    pvm_send(tid, MSG_PB);

    pb_free(pb);
}
```

***receive_pb** : elle s'occupe de :

- Réception d'un problème venant d'un processus tid.
- Allocation locale pour le problème.
- Met à jour le tid de l'envoyeur dans sender (utile dans le cas où la réception venait

d'un processus indifférent (tid == -1)).

- retourne NULL si le message n'est pas de type MSG_PB

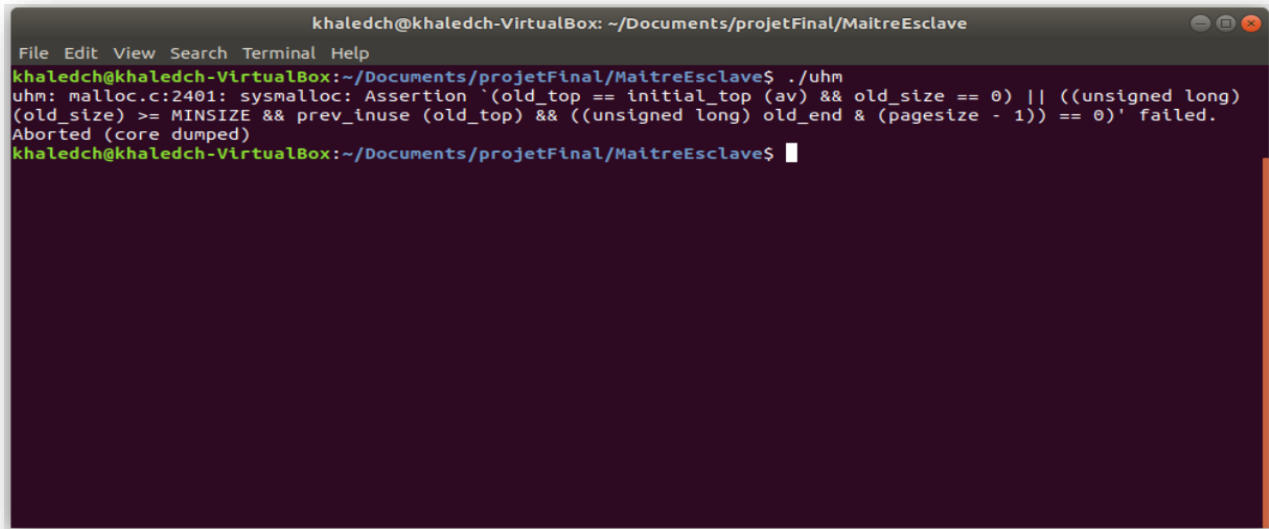
```
pb_t *receive_pb(int tid, int *sender)
{
    int tag, taille, bufid;

    bufid = pvm_rcv(tid, -1);
    pvm_bufinfo(bufid, &taille, &tag, sender);
    if (tag != MSG_PB) return NULL;
    pb_t *pb = pb_alloc();
    pvm_upkint(&(pb->taille1), 1, 1);
    pvm_upkint(&(pb->taille2), 1, 1);
    pvm_upkint(&(pb->type), 1, 1);
    pb->pts1 = point_alloc();
    pvm_upkint(&(pb->pts1->x), pb->taille1, 1);
    pvm_upkint(&(pb->pts1->y), pb->taille1, 1);
    if (pb->taille2 > 0) {
        pb->pts2 = point_alloc();
        pvm_upkint(&(pb->pts2->x), pb->taille2, 1);
        pvm_upkint(&(pb->pts2->y), pb->taille2, 1);
    }
    return pb;
}
```

d) Résultat obtenu de cet algorithme

Le programme compile avec réussite sans erreur, mais au moment de l'exécution on n'aboutit pas au résultat attendu, malgré qu'on ait implémenté correctement l'algorithme de Maitre/Esclave.

Le résultat obtenu après l'exécution :



```
khaledch@khaledch-VirtualBox: ~/Documents/projetFinal/MaitreEsclave
File Edit View Search Terminal Help
khaledch@khaledch-VirtualBox:~/Documents/projetFinal/MaitreEsclave$ ./uhm
uhm: malloc.c:2401: 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 (core dumped)
khaledch@khaledch-VirtualBox:~/Documents/projetFinal/MaitreEsclave$
```

IV. Conclusion:

La réalisation de ce projet nous a permis d'appliquer les connaissances acquises lors des séances de TP , Ainsi de se familiariser et bien manipuler les algorithmes parallèles en utilisant l'outil PVM.

En effet, ce travail étant un essai, n'est donc pas un modèle unique et parfait, c'est pourquoi nous restons ouverts à toutes les suggestions et remarques pour améliorer davantage cette initiative.

Par rapport à l'état d'avancement de projet :

- **L'algorithme de parallélisation** :le résultat final est abouti, aucune erreur de compilation ou d'exécution.
- **L'algorithme de Maitre/Esclave** : le programme se compile sans erreurs, mais au moment d'exécution il s'est implanté dans une erreur malgré que le code implémenté est logiquement juste.

V. Annexe :

Cette archive contient un rapport PDF et un dossier **sourceChenouf** à l'intérieur de ce dossier on trouve deux autres dossiers :

- **AlgorithmeParalleleSimple** qui continent :
 - Makefile
 - Upper.c : le programme principal.
 - Point.h, point.c.
- **MaitreEsclave** qui contient :
 - Makefile
 - Uhm.c : le programme du maitre.
 - Uhs.c : le programme de l'esclave.
 - Point.h, point.c.

Remarques :

- Le fichier résultat (à exécuter après le make) est nommé 'upper' qui s'exécute avec paramètres (le nombre de points) pour **AlgorithmeParalleleSimple**.
- Le fichier à exécuter après le make est nommé 'uhm' qui s'exécute sans paramètres pour **MaitreEsclave**.