Matthieu TRINQUART Info2, TD IN ,TP1

Hugo FAMECHON

Projet Système :

Calcul de surface d'un objet 3D maillé

SOMMAIRE

Introduction

1)Présentation du problème

a) But de l'algorithme

L'objectif de notre sujet consiste à calculer l'aire d'un objet maillé le plus rapidement possible.

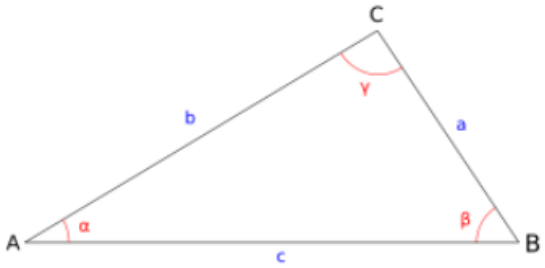
Pour ce faire il faut que nous utilisions la programmation multi-thread pour paralléliser l'algorithme séquentiel et ainsi répartir le travail entre plusieurs cœurs du processeur.

L'objet maillé est composé de plusieurs faces triangulaires.
Pour calculer l'aire totale de l'objet il nous suffit donc de faire la somme de l'aire de toute les faces.

Pour calculer l'aire d'une face nous allons utiliser la formule de Héron qui est la suivante :

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

$$\text{avec} \quad p = \frac{a+b+c}{2}.$$

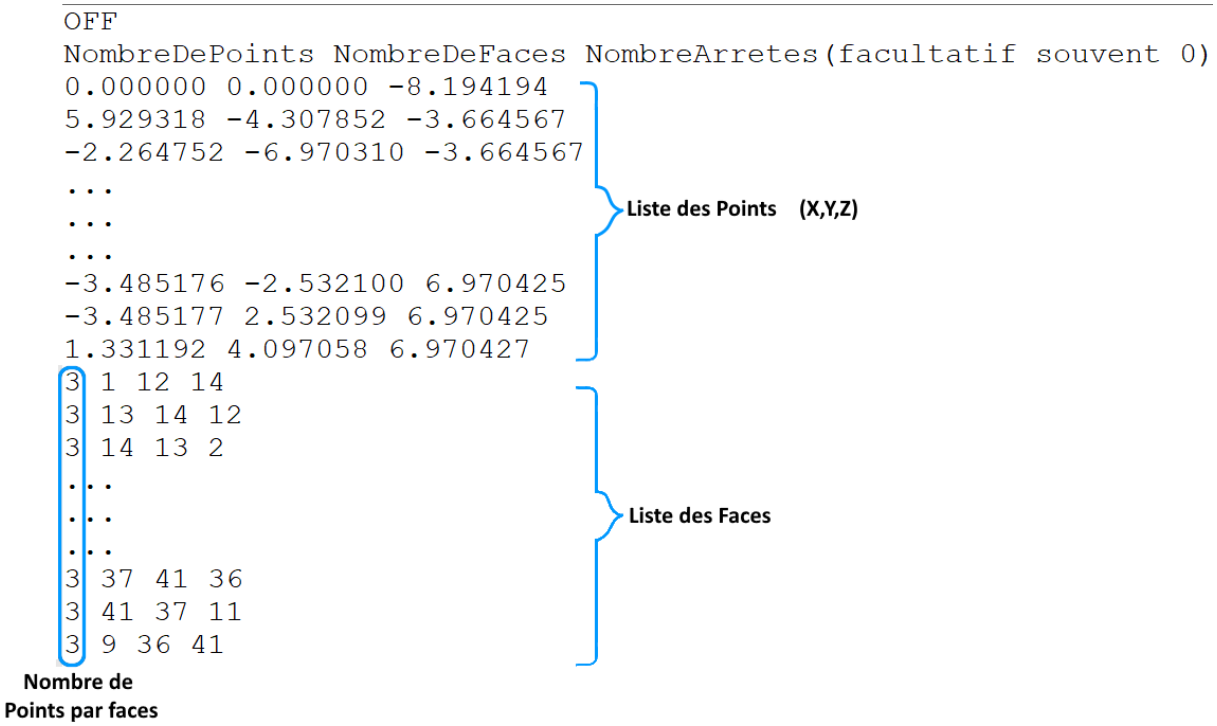


Il nous faut donc les coordonnées de chaque point pour calculer la distance a, b et c, et ainsi avoir le périmètre pour appliquer la formule de Héron.

b)Composition d'un fichier OFF

Un fichier OFF est composé de l'en tête OFF puis du nombre de Points, faces et arrêtes.
Puis de la liste des Points avec leur coordonnées X,Y,Z suivis de la liste des faces avec le nombre de points pour la face (pour notre cas toujours 3) ainsi que l'index des points utilisé

Dans l'exemple suivant la première face est composé de 3 points et utilise les points numéro 1,12 et 14 .



2)Description de la machine de teste

Nom du processeur	Intel I7-5500U
Nombre de cœur	2
Nombre de Threads	4
Fréquence processeur	3,00GHz
Cache de Niveau 1 (L1)	128KB
Cache de Niveau 2 (L2)	512KB
Cache de Niveau 3 (L3)	4MB
Mémoire RAM	7,9GB
RAM utilisée avant calcul	897MB
RAM utilisée pendant le calcul (à son maximum avec le fichier lucy.off)	2,38GB

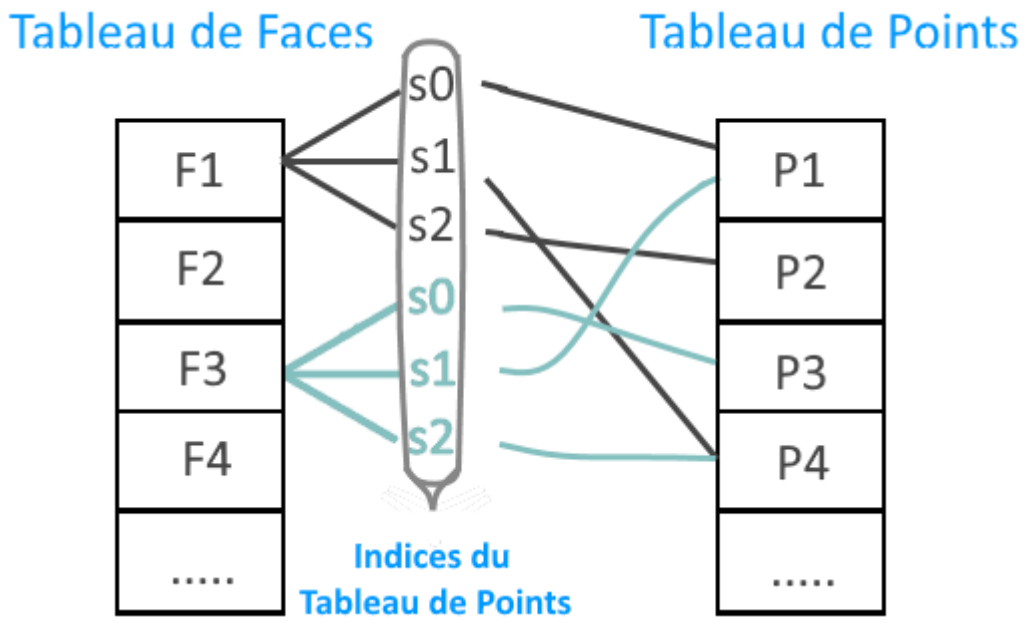
Description de l’algorithme

1)Algorithme séquentiel

Chargement du fichier OFF

Pour le chargement du fichier OFF, on Stocke dans un std::deque (file a 2 entrée) les faces et dans un autre les points.

Tableau de points et tableau de faces



Après avoir rempli les tableaux il faut calculer l’air de chaque triangle en utilisant la formule de Héron. Pour ca il faut d’abord calculer la distance entre chaque point du triangle on utilise la fonction calculDistance(point3 A , point3 B) qui utilise une formule pour calculer la distance entre 2 point dans un révérenciel en 3 dimension

```
double Objet::calculDistance(point3 A,point3 B)     //la methode utilisé pour calculer la distance entre 2 points
{
    return (sqrt((B.x - A.x ) * (B.x - A.x ) + ( B.y - A.y ) * ( B.y - A.y ) + ( B.z - A.z ) * ( B.z - A.z )));
}
```

Puis ensuite on utilise la formule de heron pour calculer l'air du triangle

```
void Objet::calculAir() //la methode utilisée pour calculer l'air
{
    double a,b,c,p;     //  p est le perimetre du triangle
                        //  a est la distance entre A et B
                        //  b est la distance entre A et C
                        //  c est la distance entre B et C



    for(unsigned int i = 0 ; i <= nbfaces - 1 ; ++i)        //on parcours le tableau de faces
    {
        a = calculDistance(lpoints[lifaces[i].s0],lpoints[lifaces[i].s1]);      //  calcule de la distance entre A et B
        b = calculDistance(lpoints[lifaces[i].s0],lpoints[lifaces[i].s2]);      //  calcule de la distance entre A et C
        c = calculDistance(lpoints[lifaces[i].s1],lpoints[lifaces[i].s2]);      //  calcule de la distance entre B et C


        p = ( a + b + c ) / 2 ;     //calcul du permietre du triangle
        Air += sqrt(p * (p - a) * (p - b) * (p - c)) ;  //utilisation de la Formule de Héron
    }
}
```

2) Algorithme avec thread

Le fonctionnement du chargement des fichiers OFF et du tableau de points et de face est le même qu'en séquentiel. La seule chose qui va changer est le calcul de l'aire.

Découpage du Tableau

Pour accélérer le temps de calcul de l'aire, on a découpé le tableau de faces par le nombre de thread que l'on fait :

```
const unsigned int nbthread = atoi(argv[1]);
```

On a ensuite fait une structure contenant la position au début et à la fin du tableau de faces ainsi que l'aire que chaque thread doit calculer :

```
struct intervalAire
{
    unsigned int debut,fin;
    double air;
    //Objet * Obj;
};
```

De cette façon le tableau de faces sera découpé et tous les threads ne calculerons que les aires des faces comprises dans l'intervalle Début/Fin.

Pour créer suffisamment de threads, on fait un tableau de threads de taille nbthread.

```
pthread_t * tbthread = nullptr ;

tbthread = new pthread_t[nbthread] ;
```

Ensuite on fait un tableau de Structure intervalAire de taille nbthread, et on instancie chaque case par un nouveau pointeur sur la structure.

```
intervalAire ** tbstruct ;

tbstruct = new intervalAire * [nbthread] ;
for (unsigned int i=0; i< nbthread ; ++i )
  tbstruct[i]= new intervalAire;
```

Puis on rempli la structure de chaque case du tableau dans un for.

```
tbstruct[i]->debut = i *  (monObjet->nbfaces / nbthread);
tbstruct[i]->fin = i * (monObjet->nbfaces / nbthread)  + (monObjet->nbfaces / nbthread);
tbstruct[i]->air = 0  ;
```

Ainsi la structure contiendra bien l'index du début et de fin que chaque thread devra traiter.

Gestion du reste

Malheureusement si nbfaces ÷ nbthread n'est pas entier, on va avoir un problème lors du découpage du tableau.
En effet, certaines faces ne vont pas être calculées.

Exemple :
Si on a 9968 faces mais qu'on veut faire 5 threads ;
Alors il y aura 3 faces qui ne seront pas prises en compte lors du découpage.(9968%5=3)
Pour remédier à ce problème il faudrait que 3 des 5 threads traitent 1 face de plus.
On a donc décalé Début et Fin lorsqu'on commençait à remplir les 3 derniers threads.

```
int o =  1; //compteur pour decaler de +1 a chaque fois
for(unsigned int i = 0 ; i < nbthread ; ++i)
{
    if(reste > 0)
    {
        if((int)i < (int)nbthread - reste)
        {
            tbstruct[i]->debut = i * (monObjet->nbfaces / nbthread);
            tbstruct[i]->fin = i * (monObjet->nbfaces / nbthread) + (monObjet->nbfaces / nbthread);
            tbstruct[i]->air = 0;
        }else if((int)i == (int)nbthread - reste)
        {
            tbstruct[i]->debut = i * (monObjet->nbfaces / nbthread);
            tbstruct[i]->fin = i * (monObjet->nbfaces / nbthread) + (monObjet->nbfaces / nbthread) + 1;
            tbstruct[i]->air = 0;
        }
        else if((int)i > (int)nbthread - reste)
        {
            tbstruct[i]->debut = i *  (monObjet->nbfaces / nbthread) + o  ;
            tbstruct[i]->fin = i * (monObjet->nbfaces / nbthread) + (monObjet->nbfaces / nbthread) + o + 1 ;
            tbstruct[i]->air = 0 ;

            ++o;    //incrémentation du compteur
        }
    }else if(reste == 0)
    {
        tbstruct[i]->debut = i * (monObjet->nbfaces / nbthread);
        tbstruct[i]->fin = i * (monObjet->nbfaces / nbthread) + (monObjet->nbfaces / nbthread);
        tbstruct[i]->air = 0;
    }
        pthread_create(&tbthread[i], NULL, calculAir, (void *)tbstruct[i]); //Creation des threads
}
```

De cette façon, même s'il reste des faces, certains threads géreront 1 face de plus.

Calcul de l'Aire :
Pour le parallèle, rien ne change sur le calcul d'aire. Les seules différences sont :
-calculAir() et calculDistance() n'appartiennent plus à la classe Objet.
-On ne retourne plus rien, vu que l'aire est un pointeur et qu'on modifie son adresse.
-Objet a été déclaré en variable globale pour pouvoir y accéder dans cacluclAir() sans avoir à le faire passer dans la structure.

```
Objet * monObjet ;      //declaration de monObjet en global pour pouvoir s'en servir dans calcuAir dans avoir a le faire passer dans une structure

double calculDistance(point3 A,point3 B)        //la methode utilisé pour calculer la distance entre 2 points
{
    return (sqrt((B.x - A.x ) * (B.x - A.x ) + ( B.y - A.y ) * ( B.y - A.y ) + ( B.z - A.z ) * ( B.z - A.z )));
}
```

```
void * calculAir(void * _arg)
{


    intervalAire * ptr = (struct intervalAire *)  _arg ;
    double a=0, b=0, c=0, p=0;  //  p est le perimetre du triangle
                                //  a est la distance entre A et B
                                //  b est la distance entre A et C
                                //  c est la distance entre B et C

    for(unsigned int i = ptr->debut ; i< ptr->fin ; ++i)
    {
        a = calculDistance(monObjet->lpoints[monObjet->lifaces[i].s0],monObjet->lpoints[monObjet->lifaces[i].s1]);      //  calcule de la distance entre A et B
        b = calculDistance(monObjet->lpoints[monObjet->lifaces[i].s0],monObjet->lpoints[monObjet->lifaces[i].s2]);      //  calcule de la distance entre A et C
        c = calculDistance(monObjet->lpoints[monObjet->lifaces[i].s1],monObjet->lpoints[monObjet->lifaces[i].s2]);      //  calcule de la distance entre B et C

        p = ( a + b + c ) / 2 ;                             //calcul du permietre du triangle
        ptr->air += sqrt(p * (p - a) * (p - b) * (p - c)) ;     //utilisation de la Formule de Héron
    }


    return NULL ;
}
```

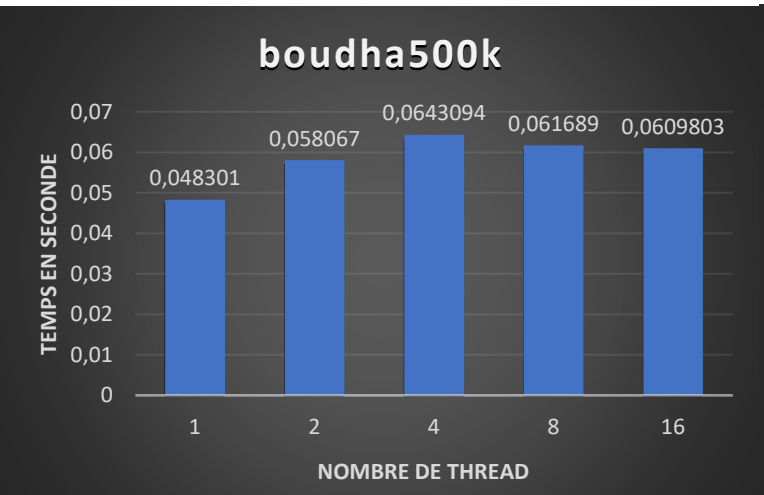
Puis on fait un pthread_join pour attendre la fin de l'exécution des threads.

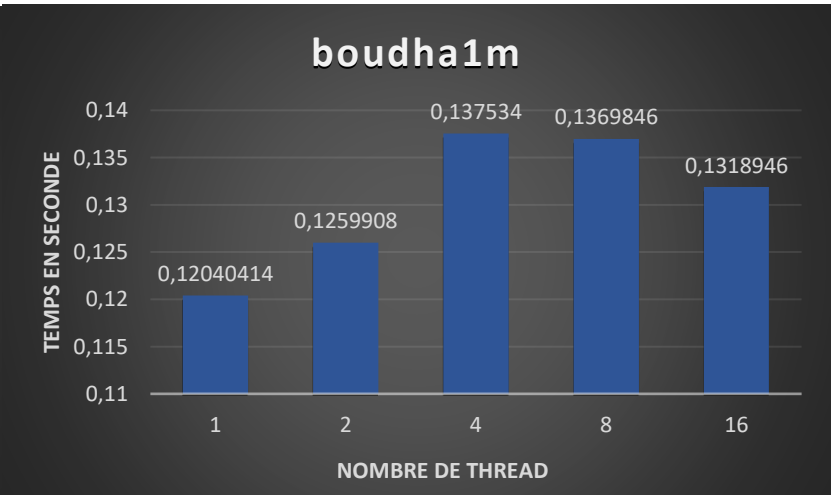
Et on fait la somme des aires du tableau de structure.

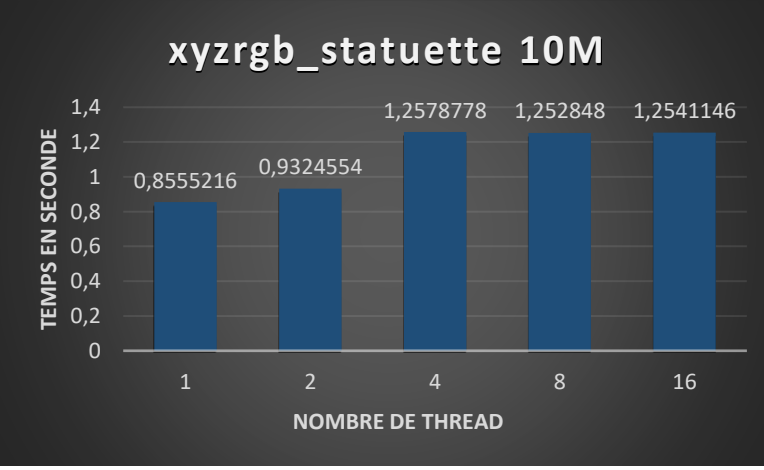
```
for(unsigned int i = 0 ; i <nbthread ; ++i)
{
    pthread_join(tbthread[i], NULL);
}

double air = 0 ;
for(unsigned int i = 0; i<nbthread ; i++)
{
//  std::cout << tbstruct[i]->air << endl;
    air = air + tbstruct[i]->air ;
}
```

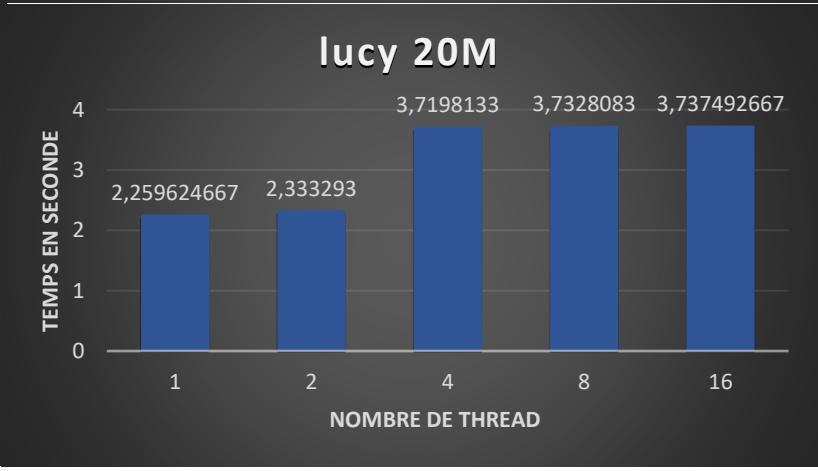
III-Analyse des résultats

1)Analyse du temps



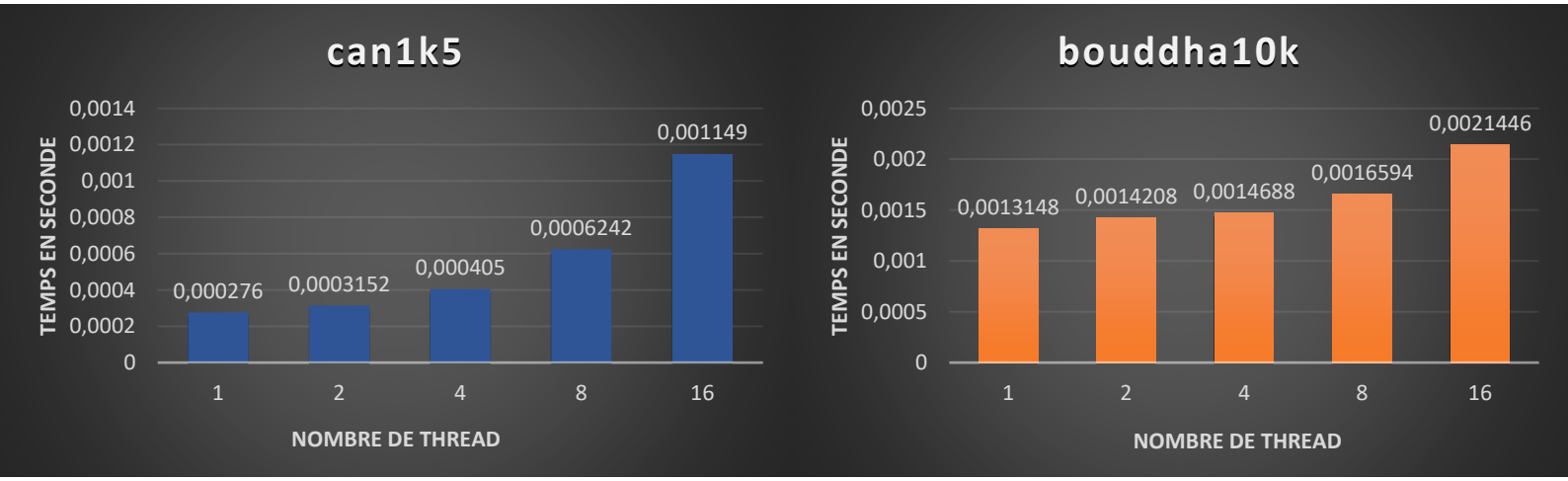






(Tous les temps ne compte pas le temps de lecture et de mise en RAM des fichiers. Il ne prend en compte que le temps de calcul de chaque face, de la création du tableau thread et la répartition des calcules entre les Threads.)

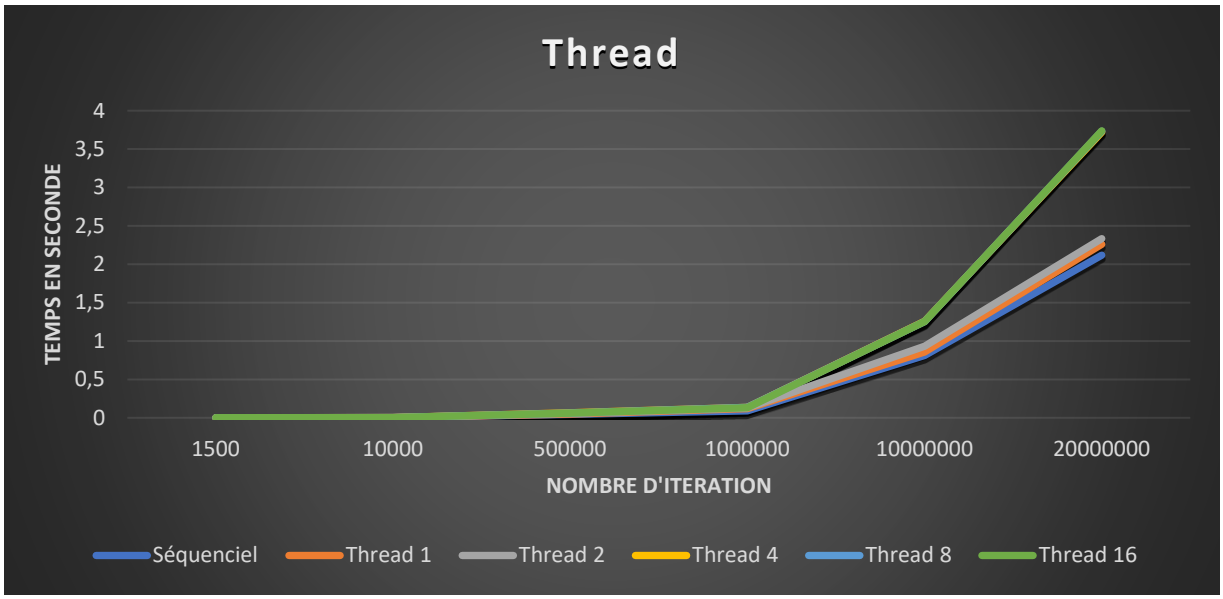
Les graphiques nous montre qu'il y a d'abord une augmentation du temps à chaque création de thread puis à partir de 4 threads une stabilisation du temps. Cette augmentation de 1 à 4 threads est due au fait que chaque Threads sont en concurrence pour l'accès à la RAM car chaque threads lisent leur face mais il arrive que certaines face aient besoins des mêmes points ce qui crée de la concurrence. Cette concurrence créer un ralentissement général du programme.On aurait put copier le tableau de point pour que chaque thread est accès a sont propre tableau de point mais le temps de copie ralentirai le temps global du programme de plus pour les gros fichier(exemple lucy.off) cela satuerai la RAM et provoquerai un swap. Ensuite la stabilisation du temps pour 8 et 16 threads est dû au fait que la machine utilisé ne peut avoir que 4 thread en même temps(cf Description de la machine).Donc la compétition ce limité à 4 thread en même temps .



On peut voir que pour les fichiers qui ne demande pas beaucoup d'itération (1500 itérations pour can1K5 et 10000 pour boudha10k) qu'il n'y a pas de stabilisation au-delà de 4 threads. Cela peut être dut a 2 raison:

-Le temps de creation du tableau de thread a plus d'importance comparer au temps global d'execution vu qu'on passe moins de temps a calculer et plus de temps a créer le tableau

-De maniere genrale le temps que l'on passe faire les thread est plus important pour les petits fichiers comparer au temps d'execution des threads seuls.



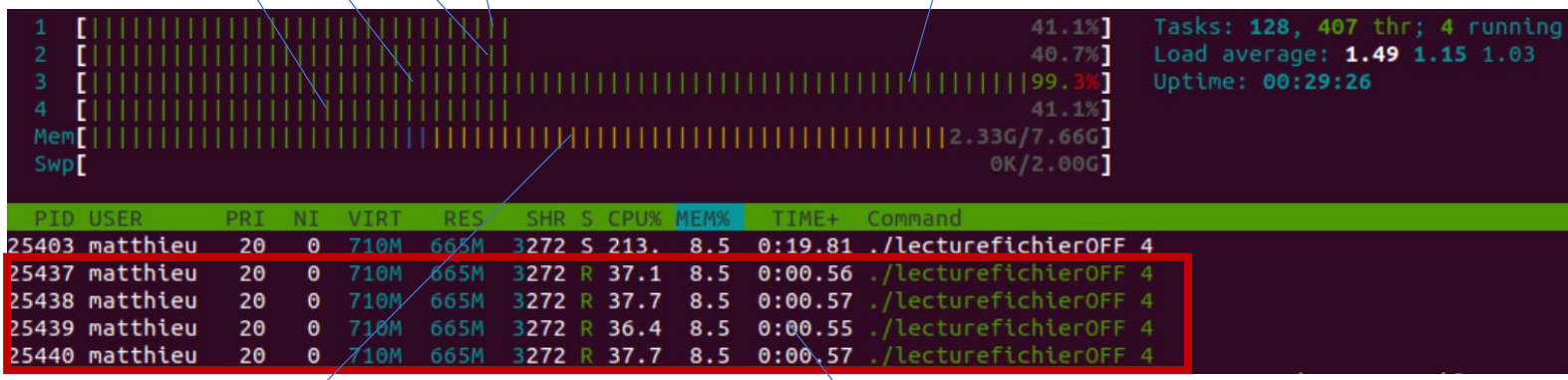
On remarque donc que plus il y a de thread plus le programme met de temps a s’executer.De plus on remarque que le temps d’execution du programme a l’air de croitre de manière légèrement expodantiel. Les courbes pour 4,8,16 thread le temps varie pas beaucoup.(remarque les courbes 4 , 8 et 16 sont confondu)

2)Analyse des threads

Durant le calcul des faces

Tout les Thread du processeur sont bien utilisé en même temps. Donc la répartition entre chaque thread est bien utilisée.

On pense que le 3 ème thread est utilisé a presque 100% parce que le htop n'a pas eu le temps de s'actualisé



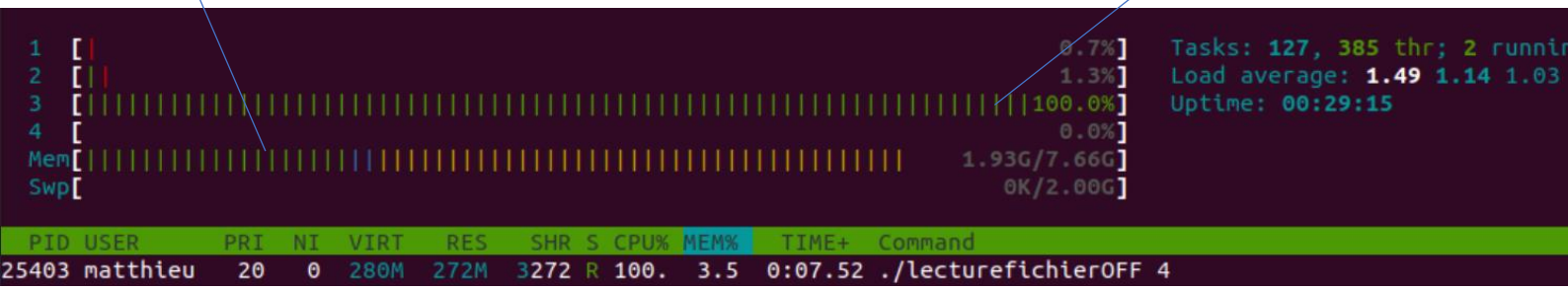
La mémoire RAM est aussi remplit car il a stocker tout le fichier OFF(Lucy.off dans cette exemple)

Les 4 threads créer par le système

Durant le chargement des fichier OFF

Remplissage de la RAM

Un seul thread utilisé durant le chargement des fichiers OFF



Un seul thread est utilisé durant le chargement des fichier OFF car 1 seul thread peux avoir accès a la lecture du disque et a l'écriture dans la mémoire donc une paralysation du chargement ne servira à rien.

Conclusion

Pour conclure, notre méthode de découpage du tableau de faces ne nous permet pas de gagner du temps car :
-La majeure partie du temps d'exécution est consacrée à la lecture.
-Même si on gagne du temps sur les calculs, ce gain est tellement minime qu'on perd plus de temps sur la création des threads et le remplissage du tableau de structure.
-Il y a concurrence lors de la lecture dans le tableau de points, car même si chaque thread traite une face différente, certaines faces utilisent les mêmes points, ce qui met en attente l'exécution de certains threads.

Une idée pour gagner du temps serait :
-De lire sur un disque dur plus rapide (SSD)
-De réussir à « paralléliser » la lecture avec un RAID.

Sinon pour empêcher que 2 threads différents aient besoin d'accéder au même point. On pourrait procéder à un rangement du tableau de faces, de sorte à mettre les faces qui ont besoin des mêmes points ensemble, pour éviter la concurrence à la lecture.
Le problème est que le temps qu'on gagnerait à l'exécution des threads serait perdu lors du rangement du tableau.