

Projet de Systèmes d'exploitation – M3101

- Calcul de la surface d'un objet 3D maillé -

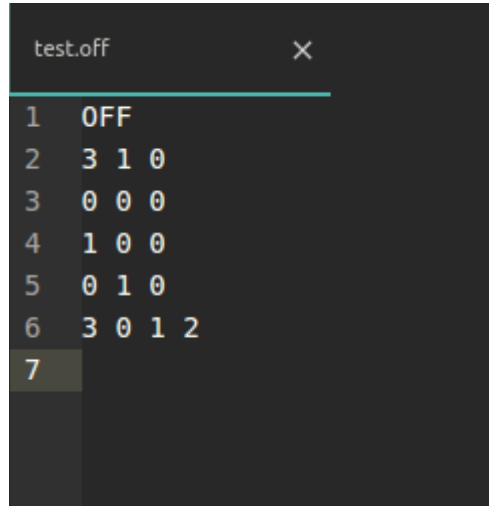
Sommaire :

I – Présentation du problème et contexte	p. 2
II – Méthodes de calcul et sources utilisées	p. 3
III – Fonctionnement du programme séquentiel	p. 4
IV – Explication de la parallélisation	p. 10
V – Exposition des tests	p. 13
VI – Conclusion	p. 17

I - Présentation du problème et contexte :

Il s'agit ici de calculer l'aire totale d'un objet en trois dimensions composé et dessiné exclusivement avec des triangles.

Pour ce faire nous disposons de fichiers d'extension .OFF, en voici un exemple:



```
test.off
1  OFF
2  3 1 0
3  0 0 0
4  1 0 0
5  0 1 0
6  3 0 1 2
7
```

Le fichier test.off est composé de 6 lignes

1ère ligne : « OFF » désigne le fait que l'on soit bien dans un fichier .off.

2ème ligne :

- Le premier nombre, ici '3', nous informe qu'il y a 3 lignes comportant les coordonnées dans l'espace de 3 sommets, ou points dans l'espace.
- Le deuxième nombre, ici '1', nous indique qu'il n'y aura qu'une seule face à calculer (sachant que nous travaillons avec uniquement des triangles, on se doute bien qu'il faille minimum trois points dans l'espace, soit trois sommets).
- Le '0' suivant n'as pas d'importance dans nos travaux.

3ème à 5ème ligne : Nous pouvons lire les coordonnées dans l'espace de points. Soit les coordonnées x, y et z par exemple.

6ème ligne :

- Le premier chiffre, ici '3', confirme le fait que nous travaillons uniquement avec des triangles. Car un triangle est bien composé uniquement de 3 sommets.
- Le deuxième nombre, ici '0', nous donne la ligne à visiter pour trouver les coordonnées dans l'espace du premier sommet de cette face.
- Même chose pour les nombres suivants.

Important : Sachant que dans notre lecture de fichier la première et la deuxième ligne ne sont pas des coordonnées de points (seulement des indications sur la composition du fichier), il est évident que la ligne « 0 » commence en fait à la ligne « 3 » de notre fichier.

Exemple : Si lors de la lecture de la composition d'une face nous voyons le nombre « 2 », cela voudra dire que la ligne à lire pour trouver les coordonnées de ce point se situe réellement à la ligne « 2+3 », soit « 5 » dans le fichier.

Ce problème n'en sera plus un une fois programmé en C++, il suffira de lire toutes les coordonnées (réelles) à partir de la première ligne qui nous intéresse jusqu'au nombre de points donné. Puis de lire les faces (nombres entiers positifs) restantes jusqu'à la fin du fichier.

Si nous reprenons l'exemple, nous atterrirons bien à l'indice 2 du tableau de points (et non 2+3 comme dans le fichier).

Plus aucun décalage n'est alors à prendre en compte.

II - Méthodes de calcul & [sources](#) utilisées :

Le problème est ici de pouvoir calculer la surface d'un objet composé d'une multitude de triangles.

Nous savons que chaque face(triangle) a 3 sommets, chaque sommets a trois coordonnées dans l'espace.

Pour connaître l'aire d'un triangle il nous est possible d'utiliser la formule d'[Héron](#) :

$$A = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{avec} \quad p = \frac{1}{2}(a+b+c)$$

Soit dans un triangle ABC les longueurs $a=AB$, $b=BC$ et $c=CA$.
 p le demi-périmètre du triangle ABC.

Il nous faut donc obtenir avant tout les longueurs des côtés du triangle avant de pouvoir calculer son aire.

Pour cela utilisons le [calcul de distance entre deux points sur le plan cartésien](#) suivant leurs coordonnées :

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}.$$

Nous pouvons ainsi la généraliser pour deux points dans [l'espace](#) :

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}.$$

Une fois ses trois côtés calculés il nous est possible de calculer la surface du triangle.

Il suffit de réitérer l'opération autant de fois qu'il y a de faces et d'ajouter chaque aire respective entre elles. Ainsi nous obtenons la surface totale de l'objet maillé.

Nous avons utilisé un bon nombre de fichier .off provenant des [numérisations de l'université de Stanford](#) pour tester le fonctionnement de notre programme.

III - Fonctionnement du programme séquentiel :

Dans le cadre de notre DUT informatique et pour pratiquer une programmation soignée nous préférons partir sur de la programmation orienté objet. Quand bien même il n'existe pas d'héritage entre nos classes.

Une documentation Doxygen est à retrouver dans l'archive rendu le 8 décembre 2015.

Récapitulatif des Classes utilisées :

- **Mesh** : Cette classe représente l'objet 3D maillé. Les principaux attributs qui la composent sont en rapport avec le fichier .off (nombre de points, nombre de faces) mais aussi avec la surface totale de l'objet. Ses méthodes sont principalement les accesseurs et mutateurs de ses attributs privés.

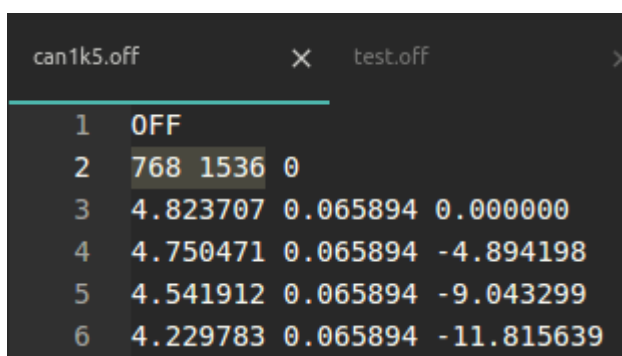
- **Face** : Elle définit le triangle, sa surface, son demi-périmètre, ses segments ainsi que les sommets qui la composent. Nous utiliserons un tableau dynamique de Face dans notre programme. Celui-ci sera associé aux accesseurs et mutateurs de la classe Face. On retrouve notamment dans cette classe le calcul d'Héron pour trouver l'aire d'un triangle en connaissant uniquement la longueur de tous ses côtés.

- **Point** : Répertorie les trois coordonnées dans l'espace de chaque sommet. Un tableau dynamique de Point sera aussi utilisé dans notre programme. Une méthode permettant le calcul de distance entre deux points dans l'espace sera notamment utilisée.

Explication :

Dans un premier temps nous traduisons tout simplement en C++ ce qui a été expliqué dans le I.

C'est-à-dire que nous ouvrons en lecture le fichier .off et récupérons le nombre de points et faces à lire.



```
1  OFF
2  768 1536 0
3  4.823707 0.065894 0.000000
4  4.750471 0.065894 -4.894198
5  4.541912 0.065894 -9.043299
6  4.229783 0.065894 -11.815639
```

Comme nous pouvons le lire dans le fichier « can1k5.off », il y a 768 points et 1536 faces. Soit 768 lignes de réels doubles à lire ainsi que 1536 d'entiers positifs.

Une fois ces deux valeurs récupérées nous créons une instance de Mesh, prenant comme paramètre le nombre de points et de faces présents dans l'objet.

```
// Ouverture du fichier en lecture -----
std::ifstream file(name_file.c_str(), std::ios::in);

if(file) // Test pour savoir si le fichier est bien présent dans le répertoire donné
{
    is_here = true;           // Si le fichier est vérifié alors la valeur est vraie et nous pouvons sortir de la boucle
    file.seekg(4, file.beg);  // On se place au quatrième octet dans le file (en partant du début), ici après le "OFF"

    unsigned int point_count = 0; // Création d'un entier positif représentant le nombre de points présent dans le fichier
    unsigned int face_count = 0;  // Création d'un autre entier positif, celui-ci représentant le nombre de faces

    file >> point_count >> face_count; // Lecture du fichier, les deux premiers entiers positifs sont insérés dans les variables précédemment définies
    Mesh mesh(point_count, face_count); // Création d'une instance de Mesh en insérant dans le constructeur le nombre de points et faces obtenus
}
```

Code C++

Nous avons maintenant besoin de lire toutes les coordonnées de points présentes dans le fichier. Si nous reprenons le fichier can1k5.off cela nous fait $(768 \times 3) = 2304$ coordonnées à stocker avant de pouvoir calculer.

Plutôt que de créer un tableau à trois dimensions, nous préférons partir sur un tableau dynamique de Point, ayant pour taille le nombre de points composant l'objet.

```
tab_point = new Point[mesh.getNumberOf_p()]; // Allocation dynamique du tableau de Point ayant pour taille le nombre de coordonnées
tab_face = new Face[mesh.getNumberOf_f()];   // Allocation dynamique du tableau de Face prenant comme taille le nombre de faces
```

Création de deux nouvelles instances de tableaux dynamiques

Nous pouvons maintenant attribuer aux attributs (privés de la classe Point) point_one, point_two et point_three les valeurs réelles récupérées à chaque ligne.

```
// ----- Lecture des coordonnées de chaque sommet -----
double p_value; // Variable temporaire servant à la lecture des doubles
/*
    Boucle for qui permet de lire la totalité des réels (coordonnées des sommets) composants le fichier .off
    Appel des différents mutateurs de la classe Point à chaque itération de boucle
    pour insérer une valeur dans les trois coordonnées du sommet 'i'
*/
for(unsigned int i=0; i<mesh.getNumberOf_p(); ++i){
    file >> p_value;
    tab_point[i].setP_one(p_value);

    file >> p_value;
    tab_point[i].setP_two(p_value);

    file >> p_value;
    tab_point[i].setP_three(p_value);
}
```

Boucle for utilisée pour la lecture des points

Dans le même principe nous effectuons la même chose pour les faces. Cette fois-ci avec les attributs `summit_one`, `summit_two` et `summit_three` eux aussi privés.

```
772 3 767 752 0
773 3 766 15 14
774 3 766 767 15
775 3 765 14 13
776 3 765 766 14
777 3 764 13 12
778 3 764 765 13
779 3 763 12 11
```

Le «3» présent lors de la lecture au début de chaque ligne sera juste lu, mais non utilisé. Nous rappelons que ce dernier indique le nombre de sommets qui sont présents dans la face. Sachant que nous travaillons exclusivement avec des triangles, nous avons évincé cette « donnée » qui pourra être utile dans un cas dit « général » de traitement de fichier .off.

```
// ----- Lecture des emplacements des coordonnées de chaque sommet pour chaque face
unsigned int s_value; // Variable temporaire pour la lecture des entier positifs
/*
   Boucle for qui lit la totalité des entiers positifs (emplacement - ligne) composants le fichier .off
   Appel des différents mutateurs de classe Face à chaque itération de boucle
   pour insérer le numéro de ligne auquel se trouve les coordonnées du sommet 'i'
*/
for(unsigned int i=0; i<mesh.getNumeroof_f(); ++i){
    file >> s_value; /* Lecture du '3' nous indiquant qu'il y a bien trois sommets
                       (inutile de le traiter car nous ne travaillons qu'avec des faces triangulaires) */

    file >> s_value;
    tab_face[i].setS_one(s_value);

    file >> s_value;
    tab_face[i].setS_two(s_value);

    file >> s_value;
    tab_face[i].setS_three(s_value);
}
```

Boucle for pour la lecture des faces

Nous avons donc nos deux tableaux dynamiques de pointeurs sur Face et Point. Chaque pointeur à un indice 'i' a donc trois attributs privés.

Indice	Attribut 1	Attribut 2	Attribut 3
0	summit_one	summit_two	summit_three
1	summit_one	summit_two	summit_three
2	summit_one	summit_two	summit_three
3	summit_one	summit_two	summit_three
4	summit_one	summit_two	summit_three
5	summit_one	summit_two	summit_three
6	summit_one	summit_two	summit_three
7	summit_one	summit_two	summit_three
8	summit_one	summit_two	summit_three
9	summit_one	summit_two	summit_three

Représentation imagée du tableau de Face et de leurs attributs

Indice	Attribut 1	Attribut 2	Attribut 3
0	point_one	point_two	point_three
1	point_one	point_two	point_three
2	point_one	point_two	point_three
3	point_one	point_two	point_three
4	point_one	point_two	point_three
5	point_one	point_two	point_three
6	point_one	point_two	point_three
7	point_one	point_two	point_three
8	point_one	point_two	point_three
9	point_one	point_two	point_three

Représentation imagée du tableau de Point et de leurs attributs

Notre lecture est donc dès à présent finie. Nous pouvons commencer nos calculs de distances comme exposé dans le **II**.

Nous utilisons une boucle for allant de 0 au nombre de faces-1 qu'il existe dans l'objet maillé. A l'intérieur de celle-ci nous trouvons :

```
for(unsigned int i=0; i<mesh.getNumberOf_f(); ++i){
    tab_face[i].setSeg_one( tab_point->calc_length( tab_face[i].getS_one(), tab_face[i].getS_two() )); // Calcul de la longueur AB
    tab_face[i].setSeg_two( tab_point->calc_length( tab_face[i].getS_two(), tab_face[i].getS_three() ) ); // Calcul de la longueur BC
    tab_face[i].setSeg_three( tab_point->calc_length( tab_face[i].getS_three(), tab_face[i].getS_one() ) ); // Calcul de la longueur CA

    mesh.setFull( mesh.getFull() + tab_face[i].calc_area() ); // Calcul de l'aire totale : addition de l'aire de la face à l'aire totale

    if(i==mesh.getNumberOf_f()-1)
        std::system("./grepmod");
}
```

Décortiquons ce condensé d'appel de méthodes :

tab_face[i].setSeg_one() remplit l'attribut privé de la classe Face « segment_one » qui représente donc la longueur d'un premier côté du triangle. C'est donc par pur déduction que les deux lignes d'en dessous représente la même chose mais pour le deuxième et troisième côtés.

En paramètre de cette méthode nous appelons la méthode « calc_length » de la classe Point. Celle-ci prend en paramètres deux entiers positifs.

En effet, bien qu'elle nous retourne un réel (une longueur) nous travaillons avec deux valeurs entières.

```
//Calcul d'une longueur d'un segment d'une face suivant les coordonnées de deux sommets :
double Point::calc_length(unsigned int A, unsigned int B) {
    return sqrt( pow(( ( this[B].getP_one()) - ( this[A].getP_one() ) ), 2) //< (xB - xA)2
    + pow(( ( this[B].getP_two()) - ( this[A].getP_two() ) ), 2) //< (yB - yA)2
    + pow(( ( this[B].getP_three()) - ( this[A].getP_three() ) ), 2) ); //< (zB - zA)2
}
```

Calcul d'une distance dans le plan 3D suivant les coordonnées de deux points

Prenons un triangle ABC, avec A(x,y,z), B(x,y,z) et C(x,y,z).

Si nous voulons calculer la longueur AB il nous faut connaître les indices du tableau de pointeur sur Point pour pouvoir récupérer les coordonnées qui nous intéressent. Ces indices nous sont donnés par les deux des trois attributs du tableau de pointeur sur Face.

Maintenant commençons nos calculs :

Nous sommes à la première itération de notre boucle for.
i vaut donc 0.

tab_face[0].getS_one() = L'indice dans le tableau de Point où se trouve les coordonnées du sommet A de la toute première face.

tab_face[0].getS_two() = L'indice pour avoir les coordonnées du sommet B.

Traduisons ça via le fichier can1k5.off :

```
768  4.193710 -0.490903 11.815639
769  4.503169 -0.532119 9.043299
770  4.709943 -0.559659 4.894198
771  3 767 0 15
772  3 767 752 0
773  3 766 15 14
774  3 766 767 15
```

Comme nous le voyons la première ligne désignant une face est la 771. Si nous reprenons notre représentation imagée des tableaux en le complétant des éléments de cette ligne nous avons :

Indice	summit_one	summit_two	summit_three
0	767	0	15

Si nous continuons dans l'exemple de notre triangle ABC première face calculée de cet objet 3D nous avons :

Longueur AB :

- summit_one = A → ses coordonnées sont à lire à l'indice 767 du tableau de Point.
- summit_two = B → ses coordonnées sont à lire à l'indice 0 du tableau de Point.

Longueur BC :

- summit_two = B.
- summit_three = C → ses coordonnées sont à lire à l'indice 15.

[...]

Soit pour la longueur AB les coordonnées suivantes (suivant notre exemple) :

```
768 4.193710 -0.490903 11.815639
769 4.503169 -0.532119 9.043299
770 4.709943 -0.559659 4.894198
771 3 767 0 15
772 3 767 752 0
773 3 766 15 14
```

Coordonnées du point A

Indice 767 dans le tableau mais ligne 770 dans le fichier comme expliqué dans la partie I.

Indice	point_one	point_two	point_three
767	4.709943	-0.559659	4.894198

```
1 OFF
2 768 1536 0
3 4.823707 0.065894 0.000000
4 4.750471 0.065894 -4.894198
5 4.541912 0.065894 -9.043299
```

Coordonnées du point B

Indice 0 vaut la ligne 3, même explication que ci-dessus.

Indice	point_one	point_two	point_three
0	4.823707	0.065894	0.000000

Nous pouvons maintenant effectuer notre calcul entre les points A et B pour connaître la distance qui les sépare et ainsi retourner un double.

Une fois les trois longueurs des côtés du triangle obtenu nous pouvons calculer l'aire de ce triangle et l'ajouter à la surface totale de l'objet via la méthode « calc_area » de la classe Face.

```
mesh.setFull( mesh.getFull() + tab_face[i].calc_area() ); // Calcul de l'aire totale : addition de l'aire de la face à l'aire totale
```

```
double Face::calc_area(){
    this->setPerimeter( ( this->getSeg_one() + this->getSeg_two() + this->getSeg_three() )/2 ); // p = 1/2 (a+b+c)
    this->setArea( sqrt( this->getPerimeter() * ( this->getPerimeter() - this->getSeg_one() ) // Aire = sqrt( p * (p-a)*(p-b)*(p-c) )
        * ( this->getPerimeter() - this->getSeg_two() )
        * ( this->getPerimeter() - this->getSeg_three() ) ) );

    return this->getArea();
}
```

Calcul d'Héron (Partie II)

IV – Explications de la parallélisation :

Il nous est demandé de paralléliser notre programme « manuellement », par la création de threads, mais aussi avec l'outil [OpenMP](#).

Une grande partie de notre programme consiste à lire les données du fichier .off pour pouvoir ensuite travailler avec. Paralléliser une lecture n'est pas recommandé ou tout simplement proscrite car impossible.

La partie parallélisée se limite donc à notre boucle for principale contenant le calcul de longueur et celui d'Héron pour trouver la surface des triangles.

Avec Thread :

#include <pthread.h>

Nous disposons d'une multitude de paramètres ce qui est problématique car il est possible de faire passer seulement un paramètre à nos threads lors de leurs créations.

Une structure permet de palier à ce problème :

```
struct ThreadParams{
    unsigned int _max;
    unsigned int _min;
    Face* _tab_face;
    Point* _tab_point;
    Mesh* _mesh;
};
```

Celle-ci reprend les nombreux éléments vu dans le séquentiel. En plus de deux bornes min et max, qui vont définir dans quel intervalle chacun de nos threads va itérer.

Notre fonction thread reprend donc le principe même du séquentiel, mais ne calcul pas dans cette fonction l'aire des triangles. Cela évite d'écrire dans une même variable lors de leur ajout pour trouver l'aire de l'objet.

```
void* calcul(void* args)
{
    ThreadParams *thread_params = (ThreadParams*)args; // Nous castons le void* obtenu en paramètre en un pointeur sur notre structure
    /*
    Cette boucle for reprend le même principe que le séquentiel, à la chose prêt que l'addition des aires
    se fait dans une boucle à part pour éviter le partage de données dans les threads.
    Il est évident que voulant "threder" à la main il nous est nécessaire de faire varier les intervalles dans lesquels vont itérer nos boucles for.
    D'où la présence de thread_params->_min et ->_max .
    */
    for(unsigned int i=thread_params->_min; i<thread_params->_max; ++i){
        thread_params->_tab_face[i].setSeg_one( thread_params->_tab_point->calc_length( thread_params->_tab_face[i].getS_one(), thread_params->_tab_face[i].getS_two() ) );
        thread_params->_tab_face[i].setSeg_two( thread_params->_tab_point->calc_length( thread_params->_tab_face[i].getS_two(), thread_params->_tab_face[i].getS_three() ) );
        thread_params->_tab_face[i].setSeg_three( thread_params->_tab_point->calc_length( thread_params->_tab_face[i].getS_three(), thread_params->_tab_face[i].getS_one() ) );
    }

    pthread_exit(NULL);
}
```

Il est notamment nécessaire lorsque nous arrivons dans la fonction de caster le pointeur sur void « args » en un pointeur sur notre structure précédemment définie.

Pour créer le nombre de thread dont nous avons besoin nous définissons une variable « THREAD_COUNT ». Il suffit alors de la modifier et de recompiler le code pour effectuer les changements.

```
const unsigned short THREAD_COUNT = 4;
```

Une fois la partie du code que l'on va paralléliser mise en place nous passons à la création de ces threads. Pour ce faire nous utilisons un entier représentant le nombre d'intervalles possible suivant le nombre de faces dans le fichier et le nombre de threads voulus.

Nous l'obtenons en soustrayant le nombre de faces dans le fichier par le nombre de faces modulo le nombre de threads, le tout divisé par le nombre de threads.

Dans le fichier can1k5.off nous avons 1356 faces.
Ainsi : $(1356 - 1356\%4) = 1356 - 0 = 1356$. $1356/4 = 339$.
Nous aurons donc 4 threads ayant 339 itérations chacun.
De cette façon nous sommes sûr d'arriver à un nombre entier.

Ensuite nous créons un tableau dynamique de pthread_t. Ceci est voulu car nous utilisons une boucle for pour automatiser la création de nos threads.

Nous avons aussi besoin d'un tableau dynamique sur notre structure ThreadParams. Celui-ci sera utile pour associer les bon paramètres à chaque threads 'i' de notre boucle for.

En suivant notre exemple avec 339 itérations, nous avons le résultat suivant :

```
thread_params[0]._max=(0+1)*339 = 339 // Donc max vaut 339.  
thread_params[0]._min= 0*339 = 0 // min vaut 0.
```

[...]

```
int segments = (face_count - (face_count%THREAD_COUNT))/THREAD_COUNT; // Représente la taille des segments dans lesquels vont boucler chacun de nos threads.  
pthread_t* threads_array = new pthread_t[THREAD_COUNT]; // Création d'un tableau dynamique de pointeur  
ThreadParams* thread_params = new ThreadParams[THREAD_COUNT]; // Création d'un tableau dynamique sur la structure précédemment définie  
for(unsigned int i=0; i<THREAD_COUNT; ++i){ // Cette boucle for remplit chacun des paramètres des threads dont nous allons avoir besoin  
    thread_params[i]._max=(i+1)*segments;  
    thread_params[i]._min=i*segments;  
    thread_params[i]._tab_face=tab_face;  
    thread_params[i]._tab_point=tab_point;  
    thread_params[i]._mesh=&mesh;  
    pthread_create(&threads_array[i], NULL, calcul, &thread_params[i]); // Création d'un thread 'i' ayant des paramètres 'i', ce dernier est alors exécuté  
}
```

Il suffit donc après avoir correctement associé les bonnes valeurs aux paramètres de nos threads de les créer.

On passe donc en paramètre de la fonction pthread_create, le thread 'i' du tableau nous intéressant, la fonction « calcul » et la structure de paramètres qui nous intéresse.

Pour finir nous devons « join » nos threads puis calculer l'aire de chacun des triangles dont nous avons les côtés. Puis l'ajouter à la surface totale de l'objet. Comme dans le séquentiel.

```

for(unsigned int i=0; i<THREAD_COUNT; ++i){ pthread_join(threads_array[i],NULL); }

for(unsigned int i=0; i<mesh.getNumberOf_f(); ++i){
    mesh.setFull(mesh.getFull()+tab_face[i].calc_area());
}

```

Avec OpenMP :

```
#include <omp.h>
```

La grosse différence par rapport au séquentiel se fait au même endroit que le threading. L'ajout des aires calculées à la surface totale se fait dans une autre boucle for pour éviter toutes collisions d'écritures dans une même variable. Utiliser un #pragma omp critical dans la section parallèle revient à la même chose.

Nous avons donc juste à reprendre le code séquentiel et y déposer des balises #pragma :

```

#pragma omp parallel num_threads(NUM_THREADS)
{
    #pragma omp for
    for(unsigned int i=0; i<mesh.getNumberOf_f(); ++i){
        tab_face[i].setSeg_one( tab_point->calc_length( tab_face[i].getS_one(), tab_face[i].getS_two() ));
        tab_face[i].setSeg_two( tab_point->calc_length( tab_face[i].getS_two(), tab_face[i].getS_three() ) );
        tab_face[i].setSeg_three( tab_point->calc_length( tab_face[i].getS_three(), tab_face[i].getS_one() ) );
    }
}

for(unsigned int i=0; i<mesh.getNumberOf_f(); ++i){
    mesh.setFull(mesh.getFull()+tab_face[i].calc_area());
}

```

Une variable globale « NUM_THREADS » définit le nombre de threads que l'on veut appliquer à notre zone de parallélisation.

V – Expositions des tests :

Description de la machine de test :

- Nombre de CPU et cœurs et Quantité de cache L1, L2 et L3.

```
spoken@dafunk:~/Documents/spacestation/Systeme_Obj3D/Sequentiel$ lscpu
Architecture:          x86_64
Mode(s) opératoire(s) des processeurs :32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) par cœur :   2
Cœur(s) par socket :   4
Socket(s):              1
Nœud(s) NUMA :          1
Identifiant constructeur :GenuineIntel
Famille de processeur :6
Modèle :                60
Révision :              3
Vitesse du processeur en MHz :808.496
BogoMIPS:               4988.77
Virtualisation :        VT-x
Cache L1d :             32K
Cache L1i :             32K
Cache L2 :              256K
Cache L3 :              6144K
NUMA node0 CPU(s):     0-7
```

- Caractéristiques CPU (fréquence):

```
Version: Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Voltage: 1.2 V
External Clock: 100 MHz
Max Speed: 3800 MHz
Current Speed: 2500 MHz
Status: Populated, Enabled
Upgrade: Socket rPGA988B
L1 Cache Handle: 0x000A
L2 Cache Handle: 0x0009
L3 Cache Handle: 0x000B
Serial Number: Not Specified
Asset Tag: Fill By OEM
Part Number: Fill By OEM
Core Count: 4
Core Enabled: 4
Thread Count: 8
Characteristics:
    64-bit capable
```

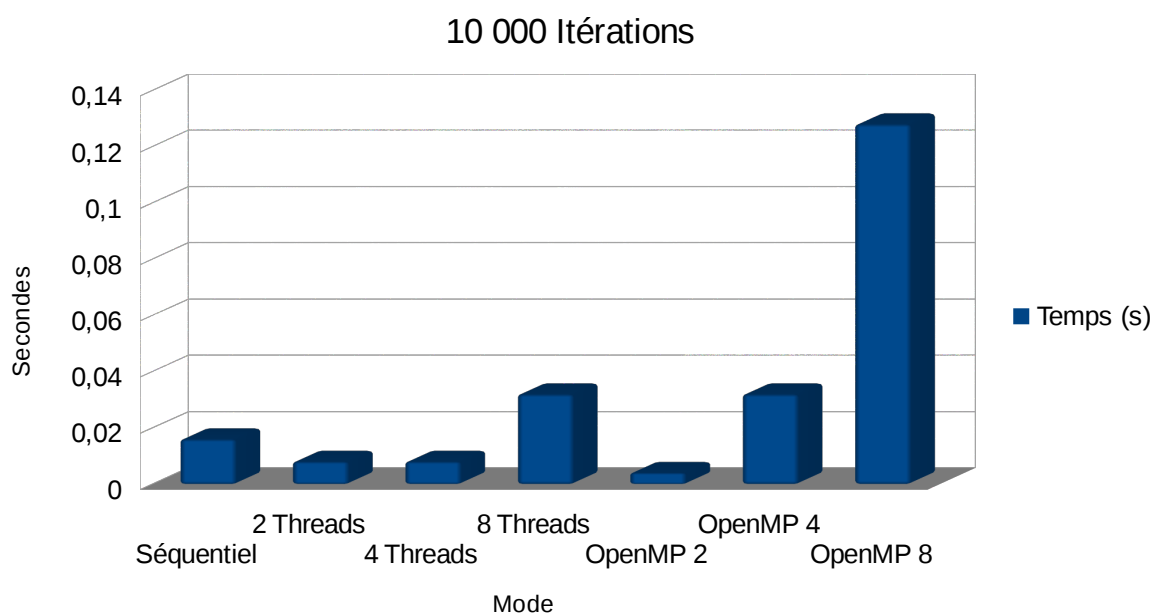
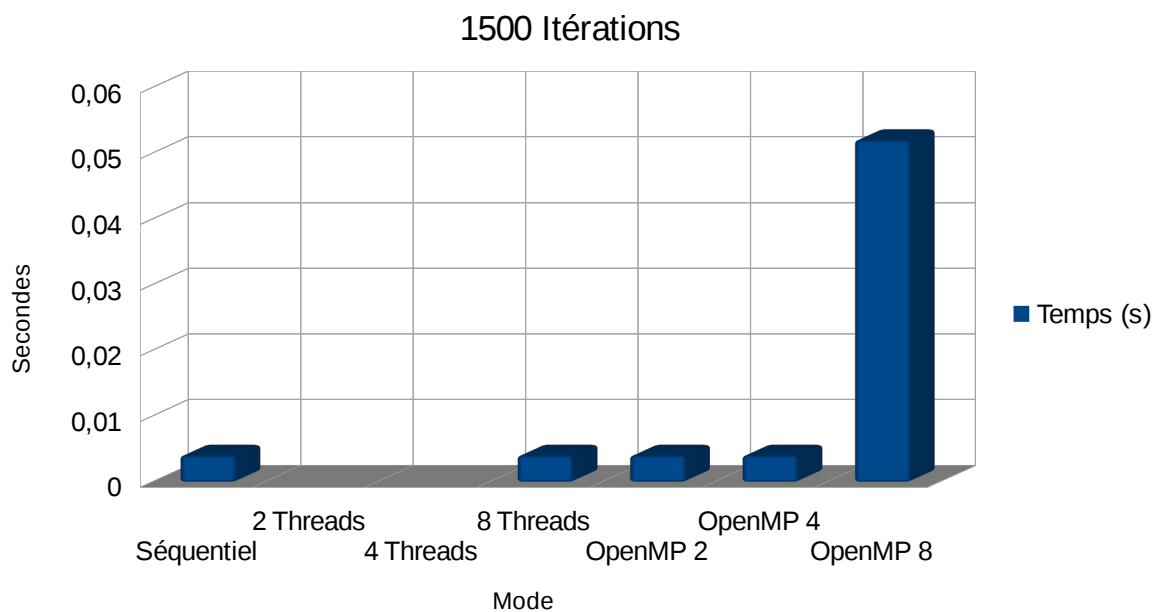
Pour exécuter notre programme nous utilisons un script bash appelé « make ». Ce dernier compile le programme puis l'exécute si l'édition de lien à bien marché. Il nous permet aussi de récupérer la quantité de RAM libre avant l'exécution du programme. Dans nos fichiers C++ une commande Système est appelé lorsque le nombre final de face à calculer est atteint par la boucle for principale. Cette commande appelle un autre script bash celui-ci récupérant une nouvelle fois la mémoire libre au moment exact où les calculs prennent fin.

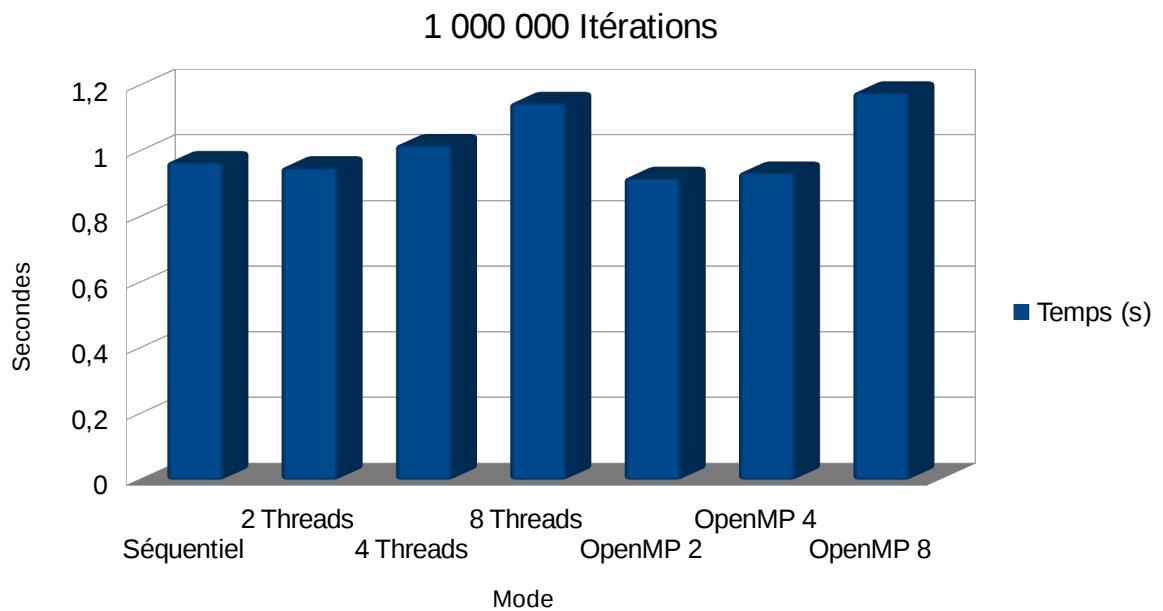
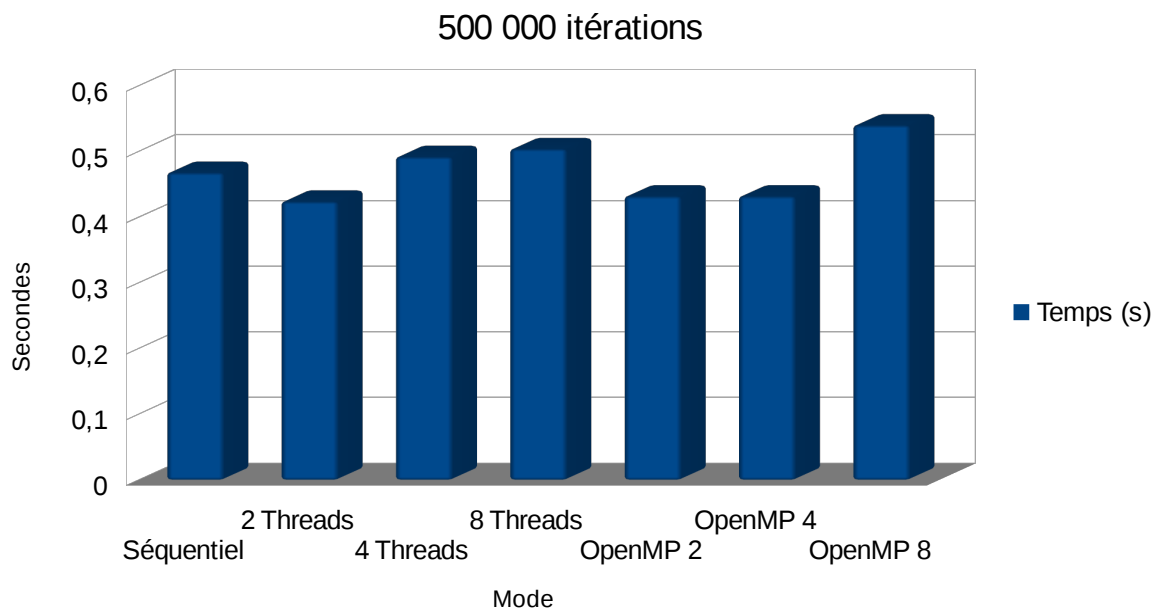
Ces données sont alors stockées dans un fichier appelés « infomem.txt » et affichés dans le terminal lorsque le programme a terminé son exécution.

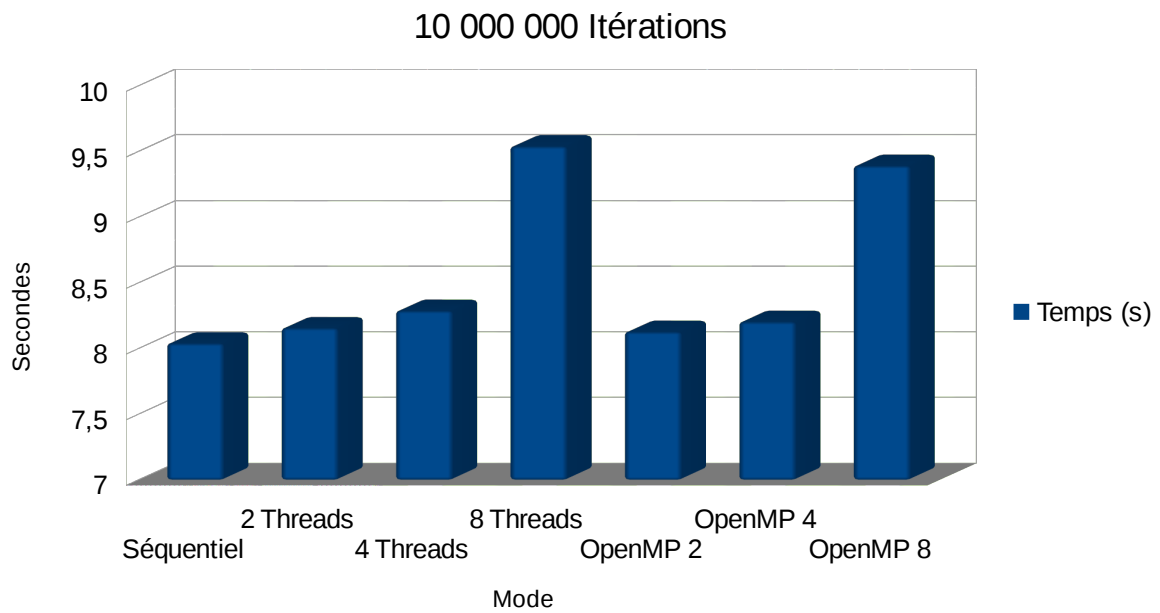
Nous devons effectuer des tests sur un différent nombre d'itérations (et avec différents niveau de threads.

Ainsi nous travaillerons avec :

- can1k5.off = 1500 faces.
- bouddha10k.off = 10 000 faces.
- bouddha500k.off = 500 000 faces.
- bouddha1m.off = 1 000 000 faces.
- statu10m.off = 10 000 000 faces.







Le taux de RAM est visible dans le tableur.ods où tous nos tests sont répertoriés.

1500 itérations		
Mode	Temps (s)	RAM utilisée (kB)
Séquentiel	0,004	2360
2 Threads	0	-808
4 Threads	0	8688
8 Threads	0,004	-596
OpenMP 2	0,004	-304
OpenMP 4	0,004	2472
OpenMP 8	0,052	236

10 000 itérations		
Mode	Temps (s)	RAM utilisée (kB)
Séquentiel	0,016	5440
2 Threads	0,008	-60
4 Threads	0,008	3192
8 Threads	0,032	-272
OpenMP 2	0,004	-596
OpenMP 4	0,032	4232
OpenMP 8	0,128	356

500 000 itérations		
Mode	Temps (s)	RAM utilisée (kB)
Séquentiel	0,468	54840
2 Threads	0,424	31992
4 Threads	0,492	40864
8 Threads	0,504	32592
OpenMP 2	0,432	31844
OpenMP 4	0,432	39052
OpenMP 8	0,54	33172

1 000 000 itérations		
Mode	Temps (s)	RAM utilisée (kB)
Séquentiel	0,968	80656
2 Threads	0,952	89596
4 Threads	1,02	94580
8 Threads	1,148	71528
OpenMP 2	0,92	72920
OpenMP 4	0,936	77212
OpenMP 8	1,18	71856

10 000 000 itérations		
Mode	Temps (s)	RAM utilisée (kB)
Séquentiel	8,04	672144
2 Threads	8,156	666616
4 Threads	8,288	685352
8 Threads	9,54	668248
OpenMP 2	8,128	699368
OpenMP 4	8,204	670104
OpenMP 8	9,392	664872

VI - Conclusion :

Nous pouvons conclure que dans notre choix de conception la parallélisation ne semble pas être un atout majeur. Bien que pour un certain nombre d'itérations OpenMP avec 2 threads et le threading à deux threads apparaissent plus rapide, en moyenne nous avons un séquentiel beaucoup plus stable et fiable au cours du temps et du nombre d'itérations.

La parallélisation à 8 threads semble une très mauvaise idée car augmentant très fortement le temps d'exécution de nos programmes.

Une très grande partie de notre programme consiste à lire dans un fichier. Le pourcentage de « code » « parallélisable » s'apparente donc à 50 % ce qui nous pénalise pour créer une réelle différence avec le séquentiel.