CHIGNARD Julien - MOINARD Corentin

Calcul de normales aux sommets d'un maillage 3D :

I - Présentation :

L'objectif de notre projet est de créer un programme capable de calculer des normales aux sommets d'un maillage 3D. Mais le projet est bien plus complexe qu'il en a l'air, en effet pour pouvoir calculer la normale au sommet d'un maillage 3D, il fallait tout d'abord pour cela calculer chaque normale aux faces de ce sommet et ainsi en faire une moyenne qui permettra de calculer par la suite la normale au sommet.
Le calcul le plus difficile à faire était donc le calcul de la normale à une face.

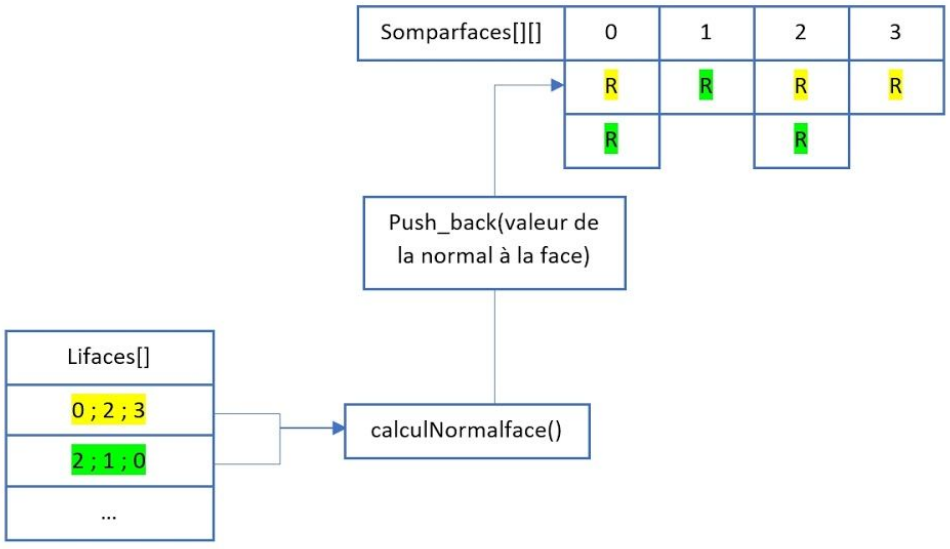
Les sources utilisés pour les calcules :
https://perso.univ-rennes1.fr/pierre.nerzic/IMR2/IMR2%20-%20Synth%C3%A8se%20d'images%20-%20CM3.pdf
https://www.developpez.net/forums/d236688/applications/developpement-2d-3d-jeux/api-graphiques/opengl/calculer-normal-vertex/

II - L'algorithme séquentiel :

On a une première Charge_OFF() (qui a été fait par notre très cher professeur M Raffin) qui nous permet de lire la totalité d'un fichier OFF et de stocker le nombre de sommet dans le fichier ainsi que le nombre de faces. Il permet aussi de stocker l'intégralité des sommets dans un tableau de deque (lpoints) et la totalité des sommets correspondant au face aussi dans un tableau deque (lifaces).

On a aussi la possibilité d'afficher l'ensemble de ses deux tableaux grâce à la fonction info().

Ensuite les fonctions que nous avons réalisé, avec pour commencer la fonction calculNormalface() qui comme indiqué va calculer les normales aux faces. On parcourt le tableau des sommet qui constitue les faces, on calcule grâce au trois sommets (qui constitue une face) la normale et on stocke le résultat dans un tableau deque à deux dimensions (Somparfaces[][]) et on stocke cette valeur à la case correspondant au sommet.



Stockage des normales aux faces (R correspond à un résultat donc une normale à une face) :

On peut alors afficher toutes les normales aux faces correspondant à chaque sommet grâce à la fonction InfoNormF() si on le souhaite.

Après cela on utilise la fonction calculNormalSommet(), cette fonction comme son nom l'indique calcule toute les normales des sommets.
On parcourt chaques cases correspondant à un sommet dans notre tableau 2D et on en fait une moyenne des valeurs que contiennent les cases, on fait ça pour tous les sommets.
On stocke ainsi toutes ses valeurs dans un tableau de deque (NormSommet[]).

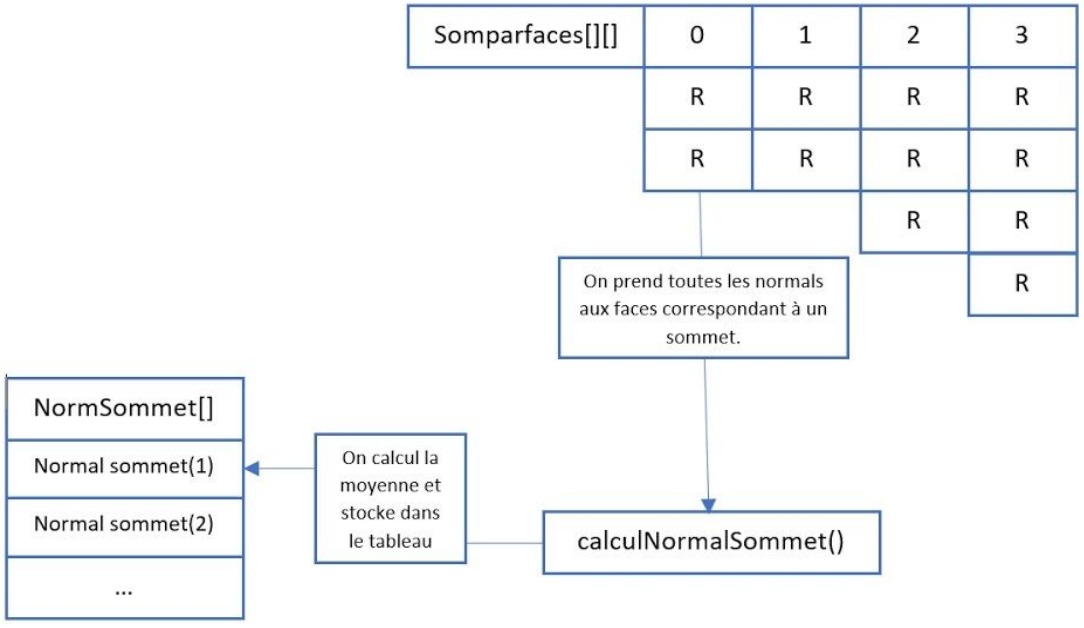
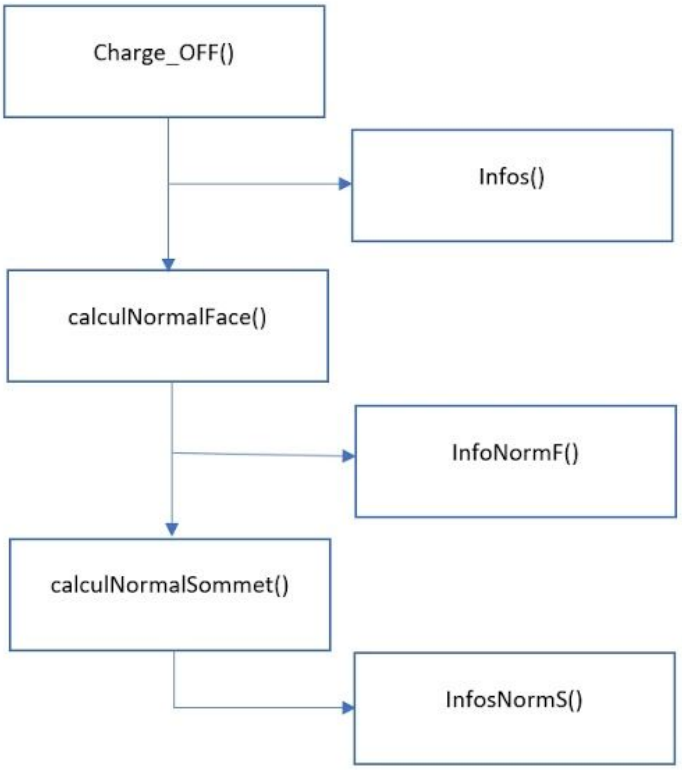


Schéma de la représentation de la fonction de calculNormalSommet :

On affichera toutes les normales aux sommets grâce à la fonction InfoNormS() qui affichera l'entièreté du tableau de normals des sommets.



Représentation de toute les fonctions :

III - La façon de paralléliser cet algorithme :

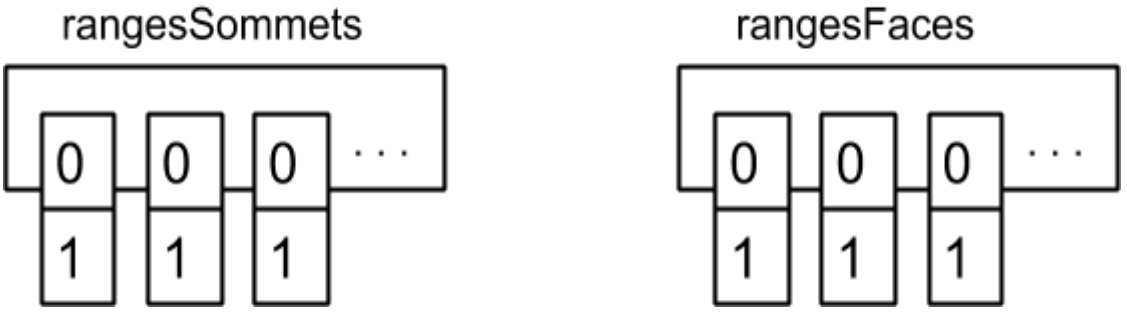
Après la création d'un tableau de threads de la taille du nombre de threads voulu, la parallélisation se passe en 3 étapes :

1. Division des Index
2. Threads Normales Faces -> join
3. Threads Normales Sommets -> join

1 - Division des Index :

Une fois nos tableaux de sommets et de faces constitués nous devons les diviser en autant de threads que l'on souhaite pour pouvoir le répartir

sur différentes parties des tableaux. C’est fait simplement en divisant le nombre de sommets / faces par le nombre de threads souhaités.
Puis pour chaque numéro de thread on multiplie ce résultat de la division par le numéro du thread pour avoir la borne inférieure et on y rajoute ce même résultat de la division -1 pour avoir la borne supérieure.
On stocke chaque borne supérieure et inférieure dans un deque de deque de unsigned long qui contiendront donc par exemple pour un 90 faces pour 3 threads le deque rangesFaces contiendra :
{ { 0 , 29 } , { 30 , 59 } , { 60 , 89 } }



2 - Threads Normales aux Faces :

Création d’un pointeur vers une instance de structure au doux nom de *cettestructuremesoulepourquoionpeutpaspasserplusieursargumentsdansunthread* (toute référence ou message caché ne serait que le fruit du hasard) qui contient

- Un pointeur vers l’Objet à manipuler.
- Un pointeur vers un deque de deque de unsigned long correspondant aux bornes à utiliser.
- Un entier contenant le numéro du thread qui est appelé.

La création des threads est faite dans une boucle for du nombre de threads demandés, la fonction calculNormalface() est appelée et le pointeur sur la structure précédemment créée est entrée en paramètre. Lors de son appel, la fonction va récupérer le numéro de son thread grâce à la structure passée en paramètre et va récupérer les bornes inférieures et supérieures à utiliser dans les boucles à la place de 0 et de la taille du tableau comme c'est fait en séquentiel.
Le calcul est le même qu'en séquentiel après ça, le stockage se fait au même endroit sauf que nous le faisons dans l'objet passé en paramètre.
Le join de tous les threads est ensuite effectué grâce à un for.

3 - Threads Normales aux sommets

Cette étape est presque identique à la précédente sauf qu'on appelle la fonction calculNormalSommet() dans la création du thread et que l'on crée l'instance de la structure avec un pointeur sur le tableau de bornes du tableau de sommets et non celui des faces.
Le joint se fait de la même manière.

IV - Tests :

IV.1 - Machines utilisées (2) :

	CPU	Coeurs	Fréquence	L1 / L2 / L3	RAM
“Fixe” (Windows 7) (1)	i5 2550K	4	3.4GHz	L1 : 2x4x32KiB L2 : 4x256KiB L3 : 6MiB	8Go DDR3
“Portable J” (Kali Linux -> Debian)	i5 2520M	4	2.5GHz	L1 : 2x2x32KiB L2 : 2x256KiB L3 : 3MiB	4Go
“Portable C” (Ubuntu)	i3 350M	2	2.27GHz	L1 : 2x2x32KiB L2 : 2x256KiB L3 : 3MiB	4Go

IV.2 - Pré-test :

Ecriture d’un script Bash (bien sur que le Bash est utile, et heureusement que notre fantastique professeur nous l’a enseigné) pour lancer le programme pour tous les fichier .off du répertoire pour 1,2,3…,10 threads chacun et écrire le temps mis pour chaque dans un fichier texte transposé ensuite en fichier .ods (3).

IV.3 - Conditions & Jeu de test :

Les tests furent effectués dans ces conditions machine :

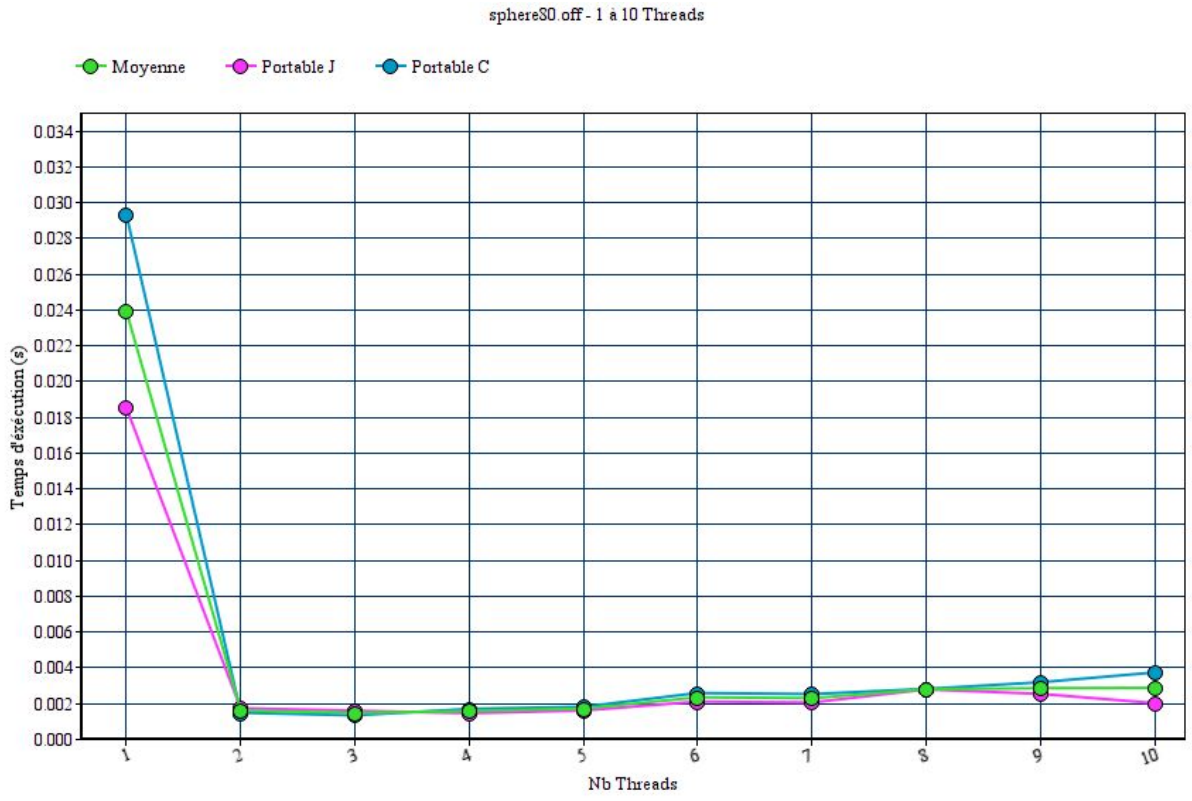
- Toutes les fenêtres autre que le Terminal fermées
- 2 terminaux ouverts : 1 pour le script et 1 htop
- time script Bash

Les tests furent effectués sur ce jeu de test (4), par ordre croissant de nombre de faces (5) :

- sphere.off -> 80 faces
- can1k5.off -> 1 536 faces
- bouddha10k.off -> 9 968 faces
- bouddha500k.off -> 499 948 faces
- bouddha1m.off -> 1 087 716 faces
- xyzrgb_statuette10M.off -> 10 000 000 faces
- lucy.off -> 28 055 742 faces

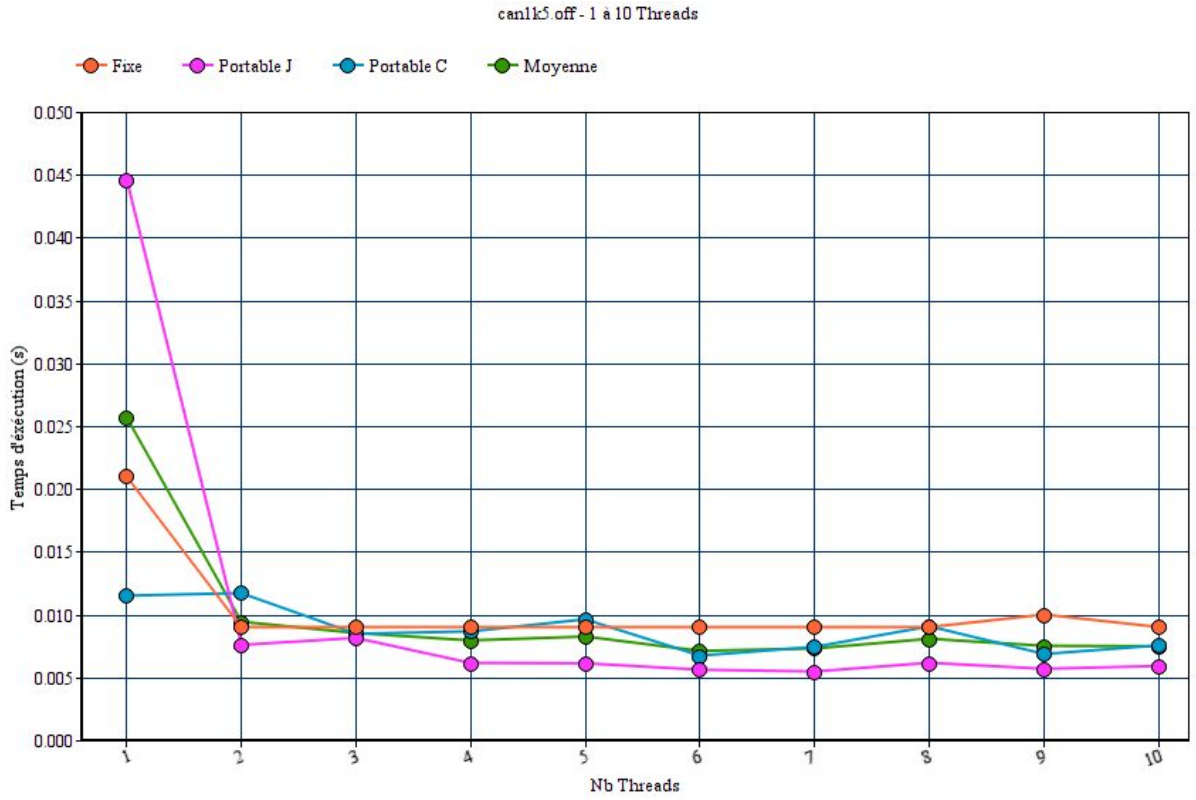
IV.4 - Résultats graphiques :

Sphere - 80 faces - 42 sommets



On voit que le séquentiel est clairement plus lent que le multi-threadé, on a un facteur 10 entre 1 et 2 threads.
Mais on remarque aussi que le programme n'est pas très efficace ensuite, le gain n'est pas significatif entre 2 et 3-4 threads, pire, on voit qu'il est plus lent avec 8-9 threads que avec 3.
C'est très probablement dû à la manière dont nous avons développé le programme, il y a beaucoup d'allocations et ce n'est pas très efficace pour des maillages avec peu de faces / sommets. (7)

Can - 1 536 faces - 768 sommets



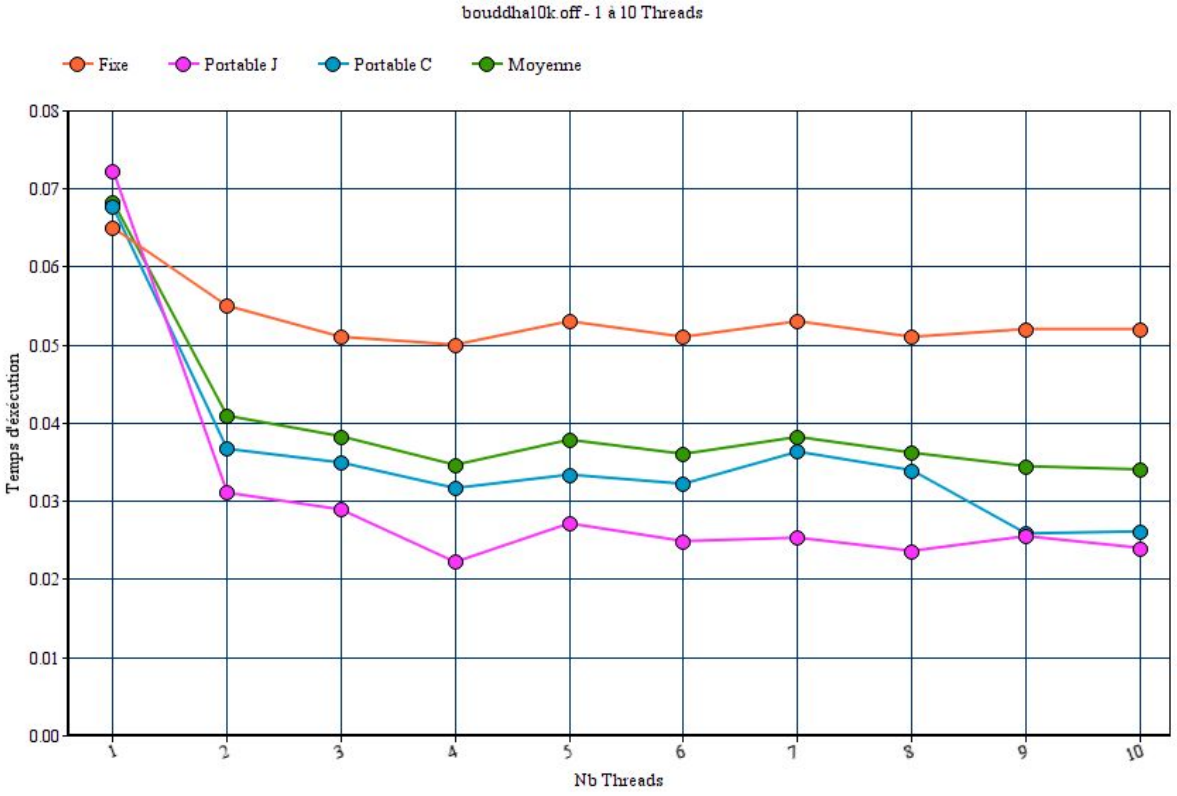
On voit ici encore que l'amélioration n'est pas très forte passé 2-3 threads. Cependant on peut remarquer que le Portable C est très efficace même en séquentiel, alors qu'à l'inverse, le Portable J est 3 fois plus lent.

Malgré cela dès 2 threads le Portable J sera le plus rapide.
La "lenteur" du PC Fixe par rapport aux 2 autres malgré son hardware légèrement meilleur peut peut-être s'expliquer par l'utilisation de Cygwin sous Windows donc peut-être plus de dialogues et "traductions" avec le processeur.

Bouddha10k

- 9 968 faces -

- 4 778 sommets -



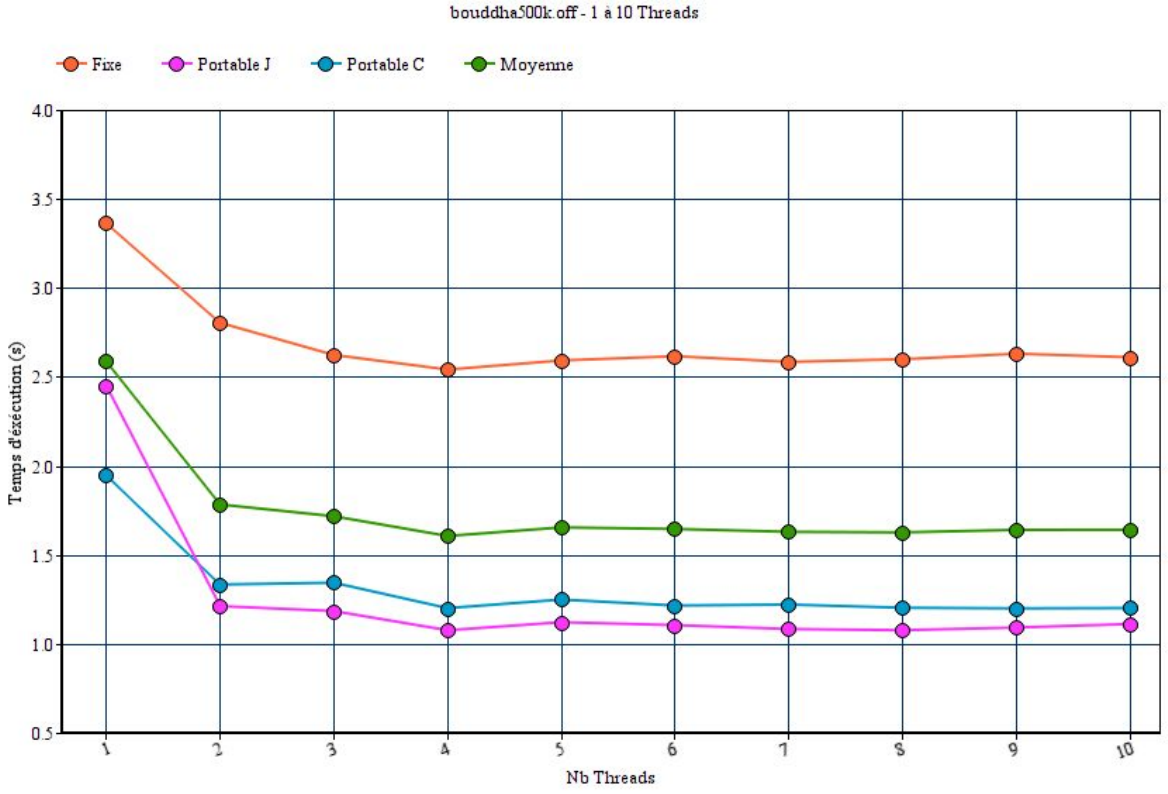
Ici les résultats commencent à devenir plus disparates : à l'inverse de Can, les 3 PC sont environ égaux en temps au séquentiel mais deviennent très différents dans le multi-threading.

Les améliorations ne sont toujours pas aussi fortes que ce à quoi l'on pourrait s'attendre.

Bouddha500k

- 499 948 faces -

- 249 768 sommets -

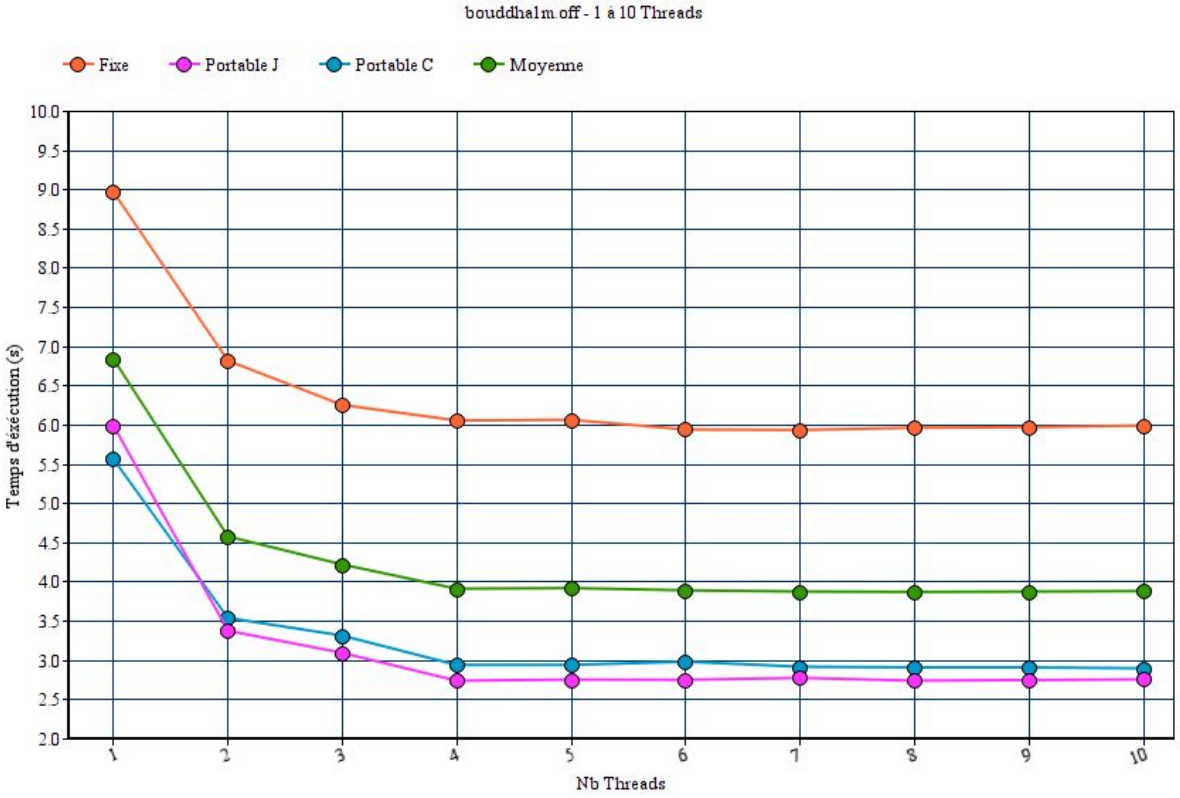


On commence à bien remarquer les différences entre les différentes machines. Nous considérons ce graphe et le suivant comme les plus représentatifs de la batterie de tests car ils montrent une très forte accélération entre 1 et 2 threads, puis des légères entre 2 et 4 et enfin quasiment plus au delà.

Bouddha1M

- 1 081 716 faces -

- 543 652 sommets -

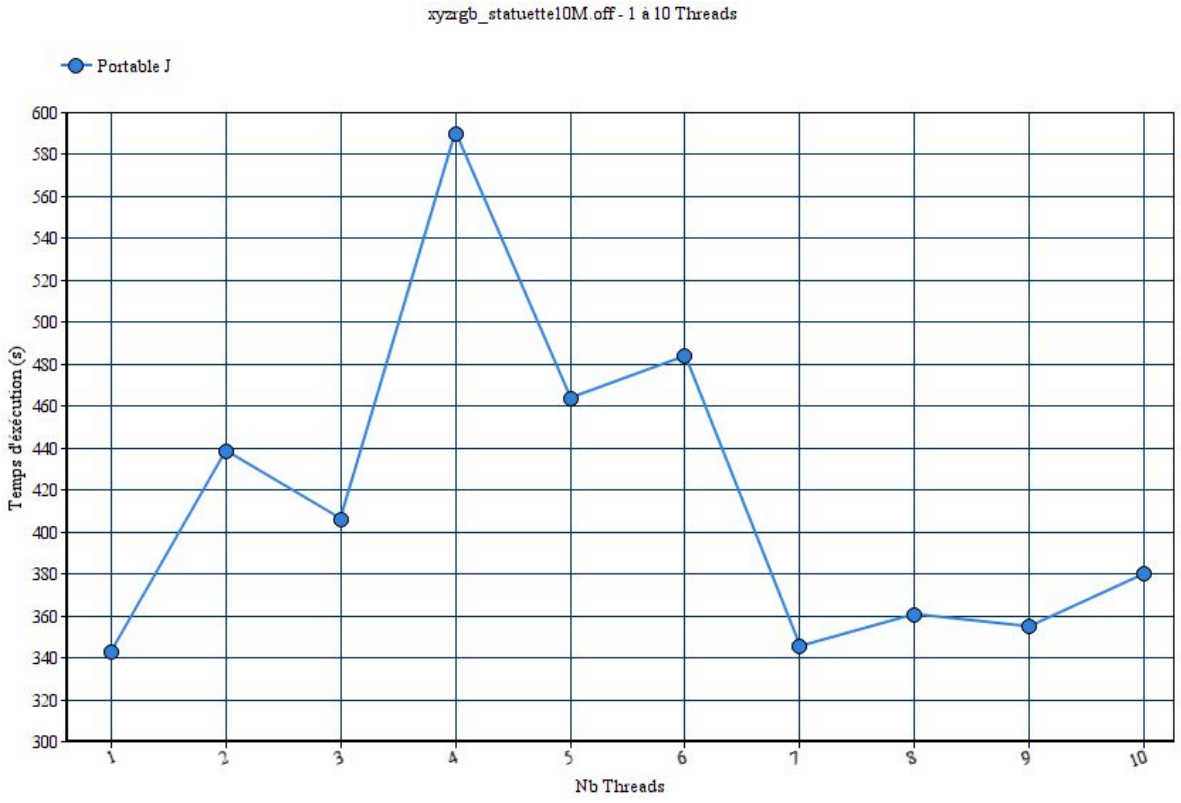


Comme mentionné précédemment ce graphe représente bien le comportement général du programme.
Hormis l'utilisation de scripts Linux sous Windows avec tout ce que ça implique, nous ne voyons pas d'explication au fait que le PC Fixe soit le plus lent.

xyzrgb_statuette

- 10 000 000 faces -

- 4 999 996 sommets -



Oui bon… Nous n'avons aucune idée de ce qu'il s'est passé entre 1 et 7 threads, le Portable J n'était pas d'humeur.

On voit cependant que comme les autres graphiques, passé 7-8 threads, la vitesse ne diminue plus, elle a même tendance à augmenter.

Lucy

- 28 055 742 faces -

- 14 027 872 sommets -

Bien que nous n'ayons pas de temps précis (Cf. (7)), le seul PC ayant réussi à correctement calculer les normales aux sommets de ce fichier (qui est colossal, tout de même) est le PC Fixe. Les autres sortant soit une erreur *bad_alloc()* (Portable C), soit le système mettait fin au process lui même et la seule sortie était "*Killed.*" (Portable J).

Malgré cela le PC Fixe mettait environ 2h40 pour calculer en séquentiel et descendait autour des 1h pour 6 threads.
Et à titre d'information le time du script Bash était à 547 minutes (oui oui) lors de son arrêt pendant le calcul de lucy avec 7 threads.

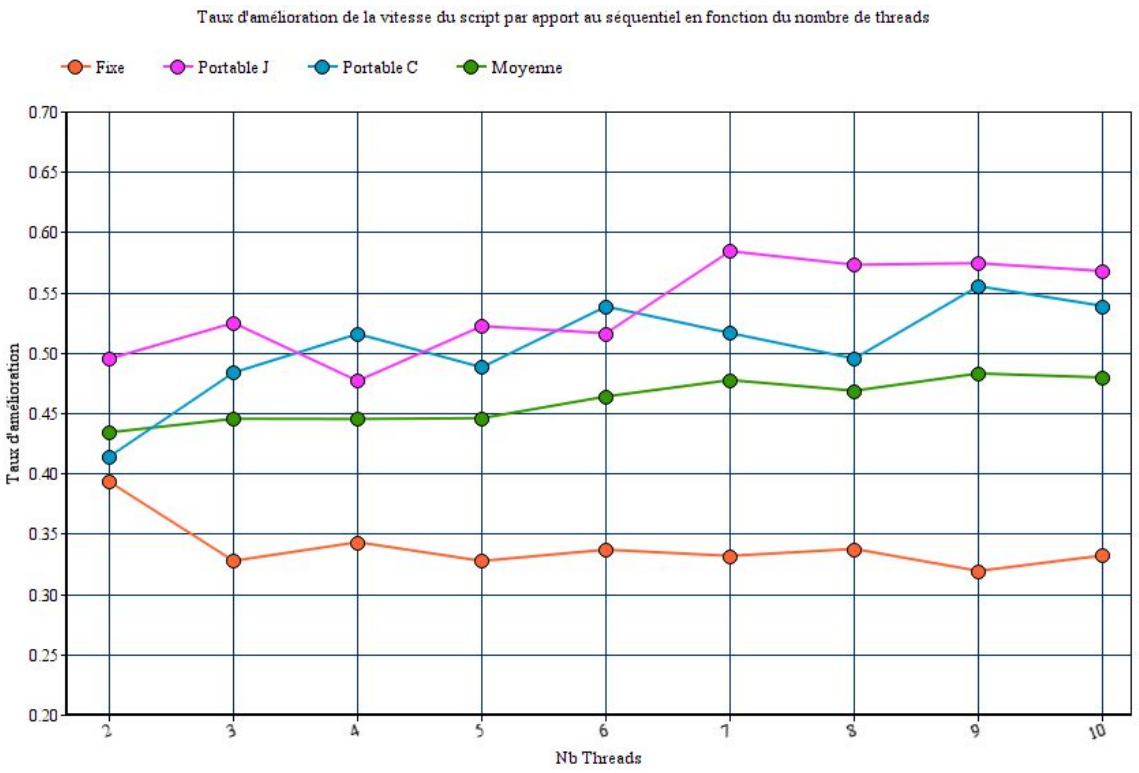
Nous n'avons pas pu récupérer les temps pour xyzrgb_statuette sur le PC Fixe.

Conclusion :

Nous avons pu constater que notre programme n'était pas optimisé car son comportement n'est pas ce à quoi l'on pourrait s'attendre, surtout pour des petites valeurs de faces / sommets.
De même nous pouvons conjecturer que la raison pour laquelle Lucy n'a fonctionné que sur le PC Fixe est l'une des raisons ci dessous :

1. Le fait d'avoir utilisé Cygwin au lieu d'une distribution Linux "native" à peut-être empêché des vérifications de sécurité quand aux threads lancés.
2. Le PC Fixe étant le seul à posséder plus de 4Go de RAM peut-être que c'est dû à cela. J'ai du mal à y croire car le fichier fait un peu plus de 1Go au total, donc même en comptant les deques créés etc je ne pense pas que ça puisse atteindre 4Go, et c'est sans compter le Swap.

On remarque que le fait d'avoir 4 coeurs (Portable J & Fixe) par rapport à 2 (Portable C) n'est pas si impactant pour l'accélération car même avec plus de 2 threads la différence d'accélération n'est pas flagrante entre Portable J et Portable C.
De plus, passé 5-6 threads, il n'y a presque plus d'accélération. Il n'y a donc pas d'intérêt à en utiliser plus dans le but que ca "aille plus vite".



Le graphique ci-dessus (6) montre que hormis l'accélération en passant du séquentiel à 2 threads, cette dernière n'augmente pas beaucoup au fur et à mesure de l'évolution du nombre de threads. On voit que l'utilisation du multi-threading sert surtout les machines Portable J et Portable C, et même que au dessus de 2 threads, il ralentit le PC Fixe par rapport à 1.

Nous tirons de ce projet et de ces tests que nous devons mieux penser notre conception, surtout en manipulant des "grandes" quantités de données et que l'utilisation de threads doit se faire en ayant pris en compte les capacités de la machine utilisée et les besoins du programme.

Dans l’absolu nous aurions dû utiliser OpenMP ou alors gérer tous les mutex et les communications entre threads pour pouvoir commencer les calculs de normales au sommets avant d’avoir fini de calculer celles aux faces.

Annexes :

(1) : Programme recompilé avec MinGW et lancé dans un terminal Cygwin.

(2) : En raison de la capacité des machines sur lesquelles nous avons effectué les tests et de l'optimisation de notre code, nous n'avons pas pu mener tous les tests à bien sur toutes les machines (cf. Annexes 5 et 6).

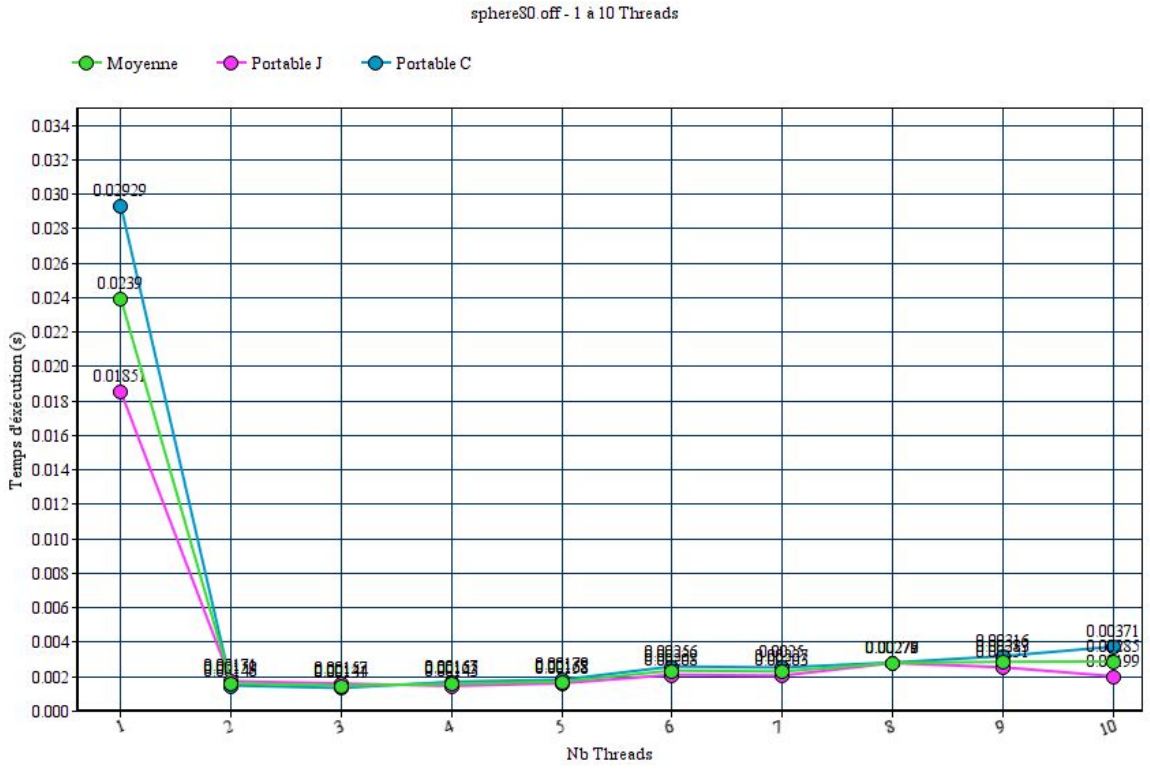
(3) : Extension de fichier tableur utilisé notamment par Libre Office Calc.

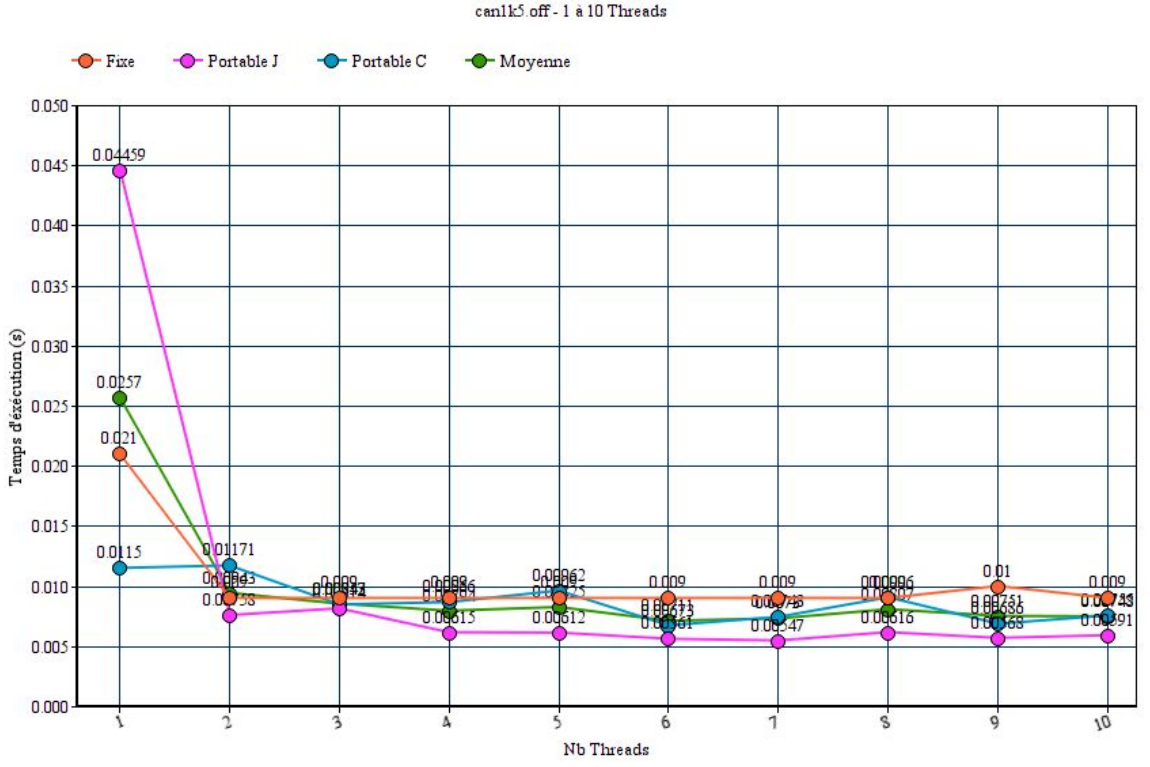
(4) : Le nombre après le nom représente le nombre de faces dans ce maillage (k = 1 000, M = 1 000 000), compter environ 2 sommets pour 1 face.

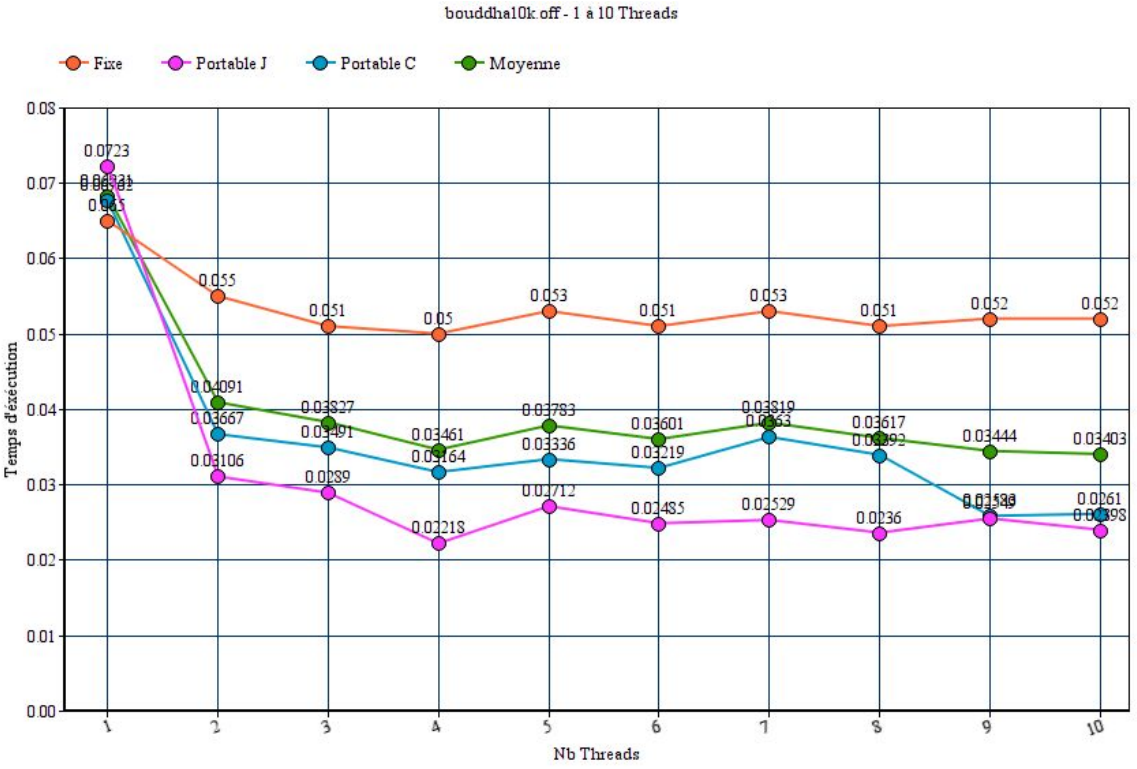
(5) : Nous avions aussi un fichier cubetri.off comportant 12 faces pour 8 points mais l'algorithme de séparation en threads gérait mal les petits nombres de sommets (<10) pour des grands nombre de threads (>8).

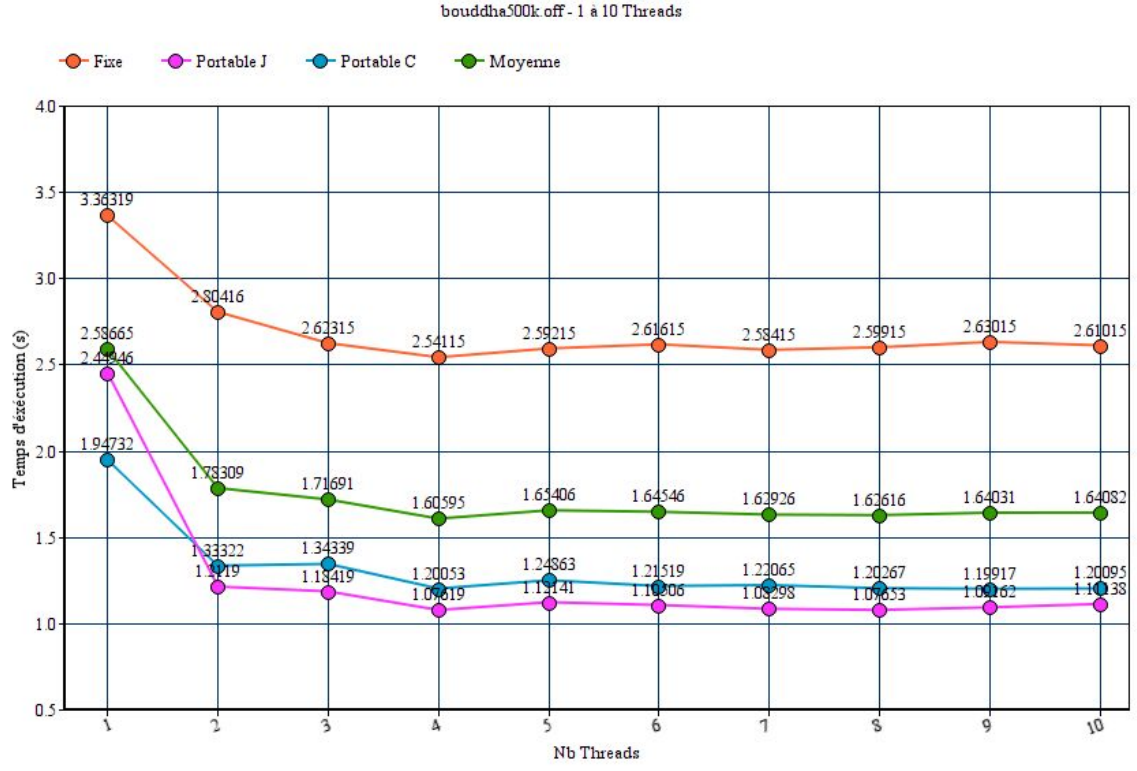
(6) : Tous les graphiques valorisés sont disponibles en Annexe.

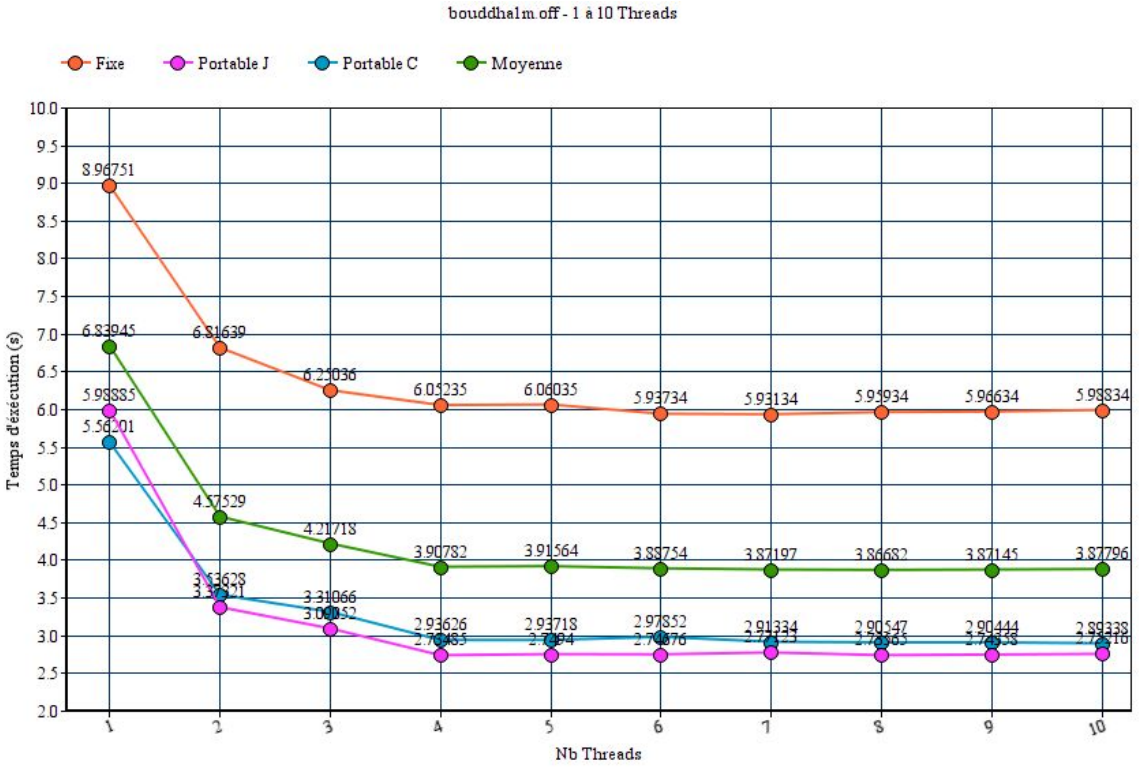
(7) : En raison de la conception (plutôt moyenne je l'avoue) du script Bash servant à regrouper et stocker les résultats, une erreur impromptue nous a empêchés de récupérer les temps du PC Fixe pour sphere, xyzrgb_statuette, et lucy.

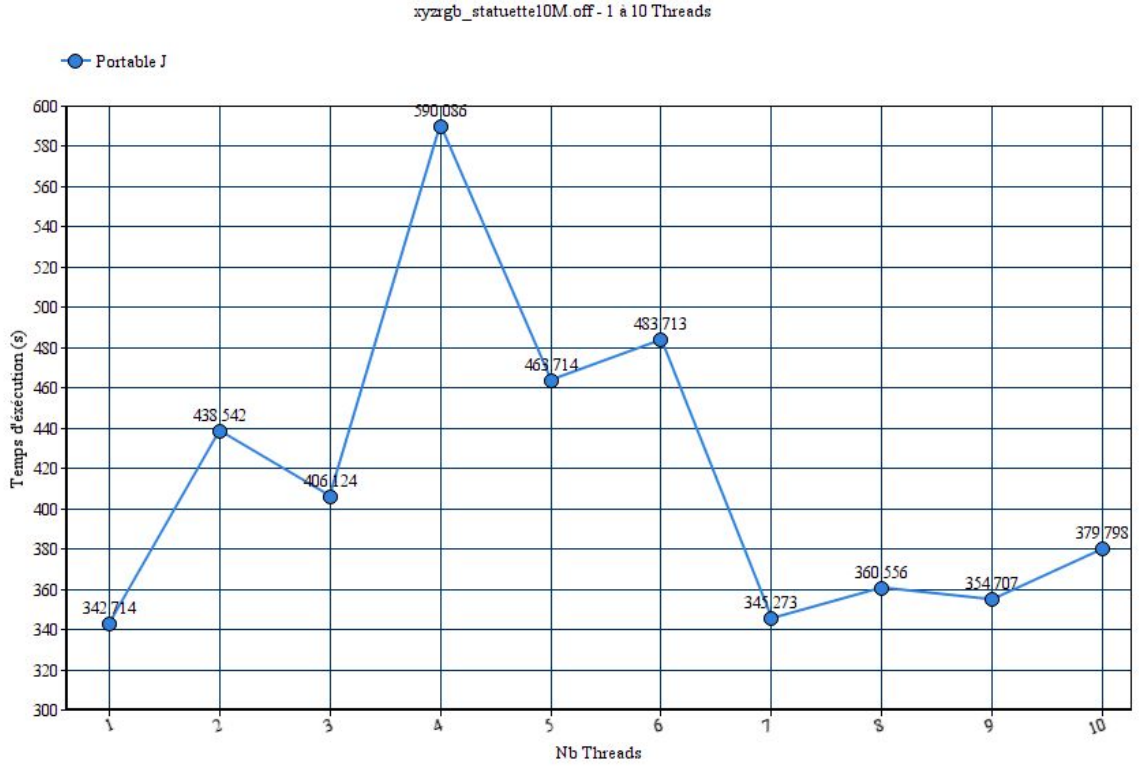

sphere80.off - 1 à 10 Threads
Moyenne
Portable J
Portable C
Temps d'éxécution (s)
Nb Threads
0.02929
0.0239
0.01851
0.00371

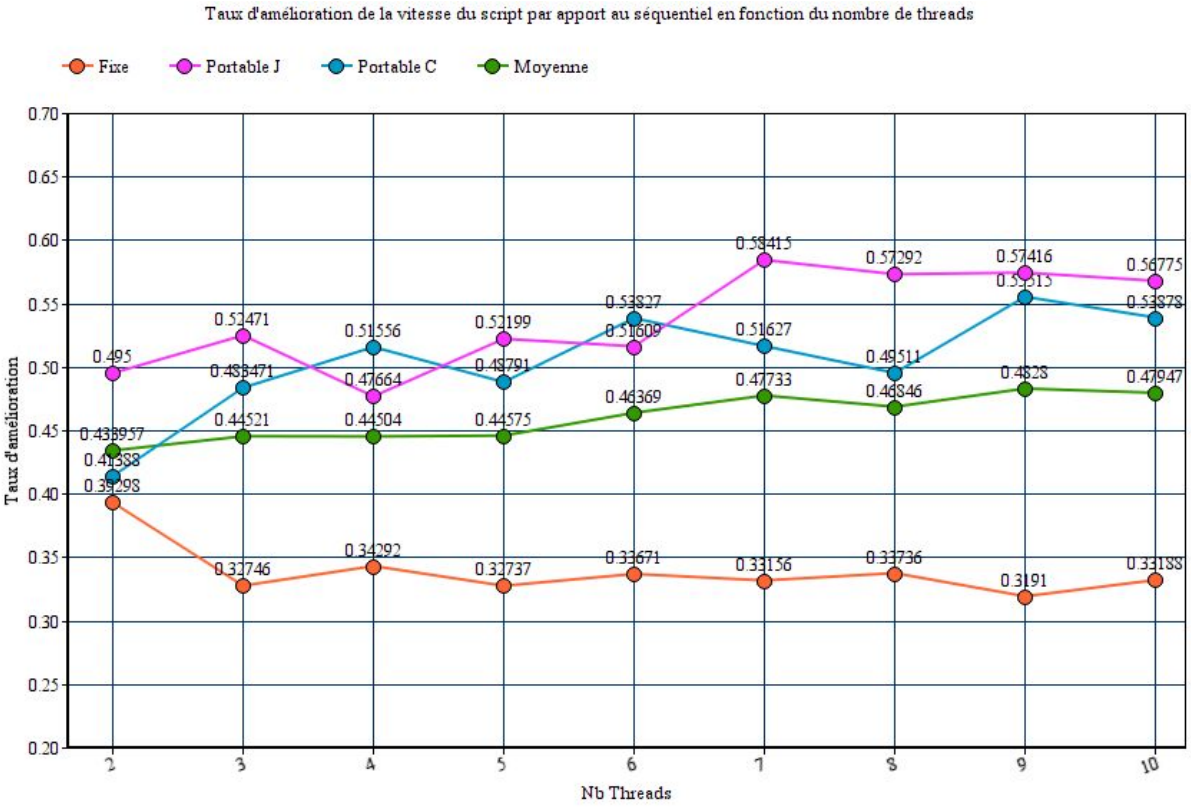

can1k5.off - 1 à 10 Threads
Fixe
Portable J
Portable C
Moyenne
Temps d'éxécution (s)
Nb Threads
0.04459
0.0257
0.021
0.0115
0.01171
0.01


bouddha10k off - 1 à 10 Threads
Fixe
Portable J
Portable C
Moyenne
Temps d'exécution
Nb Threads


bouddha500k off - 1 à 10 Threads
Fixe
Portable J
Portable C
Moyenne
Temps d'exécution (s)
Nb Threads


bouddha1m.off - 1 à 10 Threads
Fixe
Portable J
Portable C
Moyenne
Temps d'éxécution (s)
Nb Threads
8.96751
6.81639
6.25036
6.05235
6.06035
5.93734
5.93134
5.95934
5.96634
5.98834
6.83945
4.57529
4.21718
3.90782
3.91564
3.88754
3.87197
3.86682
3.87145
3.87796
5.98885
5.56201
3.53628
3.31066
3.00052
2.93626
2.93718
2.97852
2.91334
2.90547
2.90444
2.89338


xyzrgb_statuette10M.off - 1 à 10 Threads
Portable J
Temps d'éxécution (s)
Nb Threads
342.714
438.542
406.124
590.086
463.714
483.713
345.273
360.556
354.707
379.798


Taux d'amélioration de la vitesse du script par apport au séquentiel en fonction du nombre de threads
Fixe
Portable J
Portable C
Moyenne
Taux d'amélioration
Nb Threads
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
2
3
4
5
6
7
8
9
10
0.495
0.52471
0.51556
0.52199
0.51609
0.58415
0.57292
0.57416
0.56775
0.438471
0.47664
0.48791
0.53827
0.51627
0.49511
0.55515
0.53878
0.44521
0.44504
0.44575
0.46369
0.47733
0.46846
0.4828
0.47947
0.39298
0.32746
0.34292
0.32737
0.33671
0.33156
0.33736
0.3191
0.33188