



AIX-MARSEILLE UNIVERSITÉ

M3101- SYSTÈMES D'EXPLOITATION

Calcul de surface d'un objet 3D maillé

Auteurs :
Lucien Aubert
Thibaut Jallois

Enseignant :
Romain Raffin

Table des matières

1	Introduction	2
2	Environnement d'expérimentation	2
2.1	Machine 1	2
2.2	Machine 2	2
2.3	Machine 3	2
3	Algorithmique et implémentation	2
3.1	Algorithme séquentiel	2
3.1.1	Conception	3
3.2	Algorithmes parallèles	3
3.2.1	Threads	3
3.2.2	OpenMP	4
4	Résultats	4
5	Conclusion	5

1 Introduction

L'objectif consiste en l'optimisation, par parallélisation, du calcul de la surface d'un objet 3D maillé (triangles) au format OFF[1] à l'aide de la formule de Héron[2].

Le programme implémente trois algorithmes

- Classique, séquentiel
- Avec `pthread`[3], parallélisé
- Avec `OpenMP`, parallélisé également

2 Environnement d'expérimentation

La phase de test s'est déroulée sur trois machines dont voici les configurations

2.1 Machine 1

Intel i7-3612QM 2.10GHz, 8 CPU, 4 cœurs, L1 64K, L2 256K, L3 6144K
12Go RAM DDR3 800MHz

2.2 Machine 2

AMD FX(tm)-8350 4.20GHz, 8 CPU, 4 cœurs, L1 64K, L2 2048K, L3 8192K
8Go RAM DDR3 2133MHz

2.3 Machine 3

Intel i5-4590 3.30GHz, 4 CPU, 4 cœurs, L1 32K, L2 256K, L3 6144K 8Go
RAM DDR3 1600MHz

3 Algorithmique et implémentation

3.1 Algorithme séquentiel

De manière à pouvoir travailler sur les données contenues dans les fichier OFF on lit ce fichier et on place chaque sommet et chaque face dans deux `std::deque`.

```
1 class Solid {
2 private:
3     std::deque<Point> points;
4     std::deque<Face> faces;
5 }
```

On somme l'aire de chaque triangle du volume, calculée à l'aide de la formule de Héron, ce qui nous donne la surface totale du volume.

3.1.1 Conception

Dans la formule de Héron $S = \sqrt{p(p-a)(p-b)(p-c)}$ nous avons besoin des longueurs des côtés de chaque triangle. La méthode `Point::distanceFrom(Point*)` nous permet donc d'obtenir les termes a , b et c .

```
1 double Point::distanceFrom(Point* p) {
2     return sqrt(pow(p->x-x, 2)+pow(p->y-y, 2)+pow(p->z-z, 2));
3 }
```

Ces termes sont utilisés pour calculer $p = \frac{a+b+c}{2}$ et enfin l'aire de la face.

```
1 double Face::computeArea(Face *face) {
2     double area = 0;
3     double distances[face->nbVertices]; // Distances a, b et c
4
5     Point* last = face->back();
6     unsigned i = 0;
7     for(auto it = face->begin(); it != face->end(); it++) {
8         distances[i] = (*it)->distanceFrom(last);
9         area += distances[i];
10        last = (*it);
11        i++;
12    }
13
14    area /= 2.f;
15    double p = area;
16
17    for(unsigned i = 0; i < face->nbVertices; i++) {
18        area *= p-distances[i];
19    }
20
21    return sqrt(area);
22 }
```

3.2 Algorithmes parallèles

3.2.1 Threads

Le `std::deque` de faces est décomposé en n sous-ensembles correspondants au faces sur lesquelles chaque thread va travailler.

On lance les thread en leur donnant l'adresse de la fonction `computeSurface(void*)` puis on somme leur sortie en attendant la fin de leur exécution grâce à la fonction `pthread_join(pthread_t, void**)`

```
1 // On calcule le nombre de faces par thread
2 int range = faces.size()/nbThreads;
3 if(range == 0) { // En vitant de faire n'importe quoi
4     range = 1;
5     nbThreads = faces.size()-1;
6 }
7
8 double result = 0.f;
9 int last = 0;
10
11 // On declare des paquets de pointeurs vers des faces
```

```

12  std::vector<std::deque<Face*>*> facesBunch;
13
14  for(size_t i = 0; i < nbThreads; i++) {
15      last = (i+1)*range-1; // On fixe le debut de chaque paquet
16
17      // On ne doit pas dpasser du tableau de faces complet
18      if(last+range >= faces.size() && faces.size()-last > 0) {
19          last += faces.size()-last-1;
20      }
21
22      // On remplit notre paquet
23      facesBunch.push_back(new std::deque<Face*>());
24      size_t f = 0;
25      for(size_t j = i*range; j <= last; j++) {
26          facesBunch[i]->push_back(&faces.at(j));
27          f++;
28      }
29  }
30
31  // On cree les threads en leur passant les faces qu'ils
32  // doivent traiter en parametre
33  for(size_t i = 0; i < nbThreads; i++) {
34      pthread_create(&threads[i], NULL, computeFaces, (void*)facesBunch[i]);
35  }

```

3.2.2 OpenMP

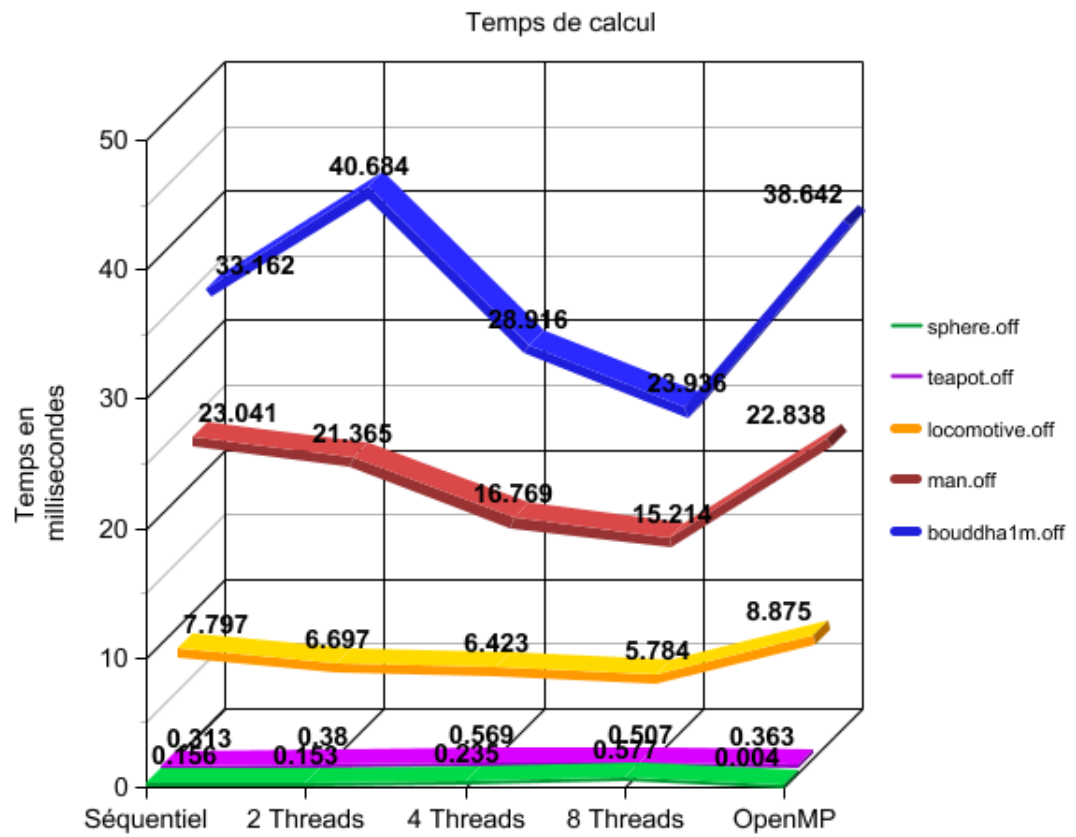
Il n'y a pas grand chose à faire pour utiliser OpenMP. L'ajout d'un `#pragma` suffit à paralléliser la boucle `for` qui somme les valeurs de chaque triangle.

```

1  #pragma omp parallel for reduction ( + : result )
2  double result = 0.f;
3  for(long i = 0; i < faces.size(); i++) {
4      result += Face::computeArea(&faces[i]);
5  }

```

4 Résultats



5 Conclusion

Références

- [1] Wikipedia. Spécification du format de fichier off.
[https://en.wikipedia.org/wiki/OFF_\(file_format\)](https://en.wikipedia.org/wiki/OFF_(file_format)).
- [2] Wikipedia. Formule de héron.
https://en.wikipedia.org/wiki/Heron%27s_formula.
- [3] Franck Hecht. Initiation à la programmation multitâche en c avec pthreads.
<http://franckh.developpez.com/tutoriels/posix/pthreads/>.