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 **5**
4.1 Machine 1 5
4.2 Machine 2 5

5 Conclusion **6**

1 Introduction

L'objectif consiste en l'optimisation, par parallèlisation, 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

Ces trois modes d'exécution sont activables grâce aux options `-t 0` (séquentiel), `-t n` (avec n threads) et `-omp` (avec OpenMP).

Les sources du projet sont disponibles sur le dépôt Github SurfaceMeasurer[4].

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.

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

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

```
double Face::computeArea(Face *face) {
  double area = 0;
  double distances[face->nbVertices]; // Distances a, b et c

  Point* last = face->back();
  unsigned i = 0;
  for(auto it = face->begin(); it != face->end(); it++) {
      distances[i] = (*it)->distanceFrom(last);
      area += distances[i];
      last = (*it);
      i++;
  }

  area /= 2.f;
  double p = area;

  for(unsigned i = 0; i < face->nbVertices; i++) {
      area *= p-distances[i];
  }

  return sqrt(area);
}
```

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**)`

```
// On calcule le nombre de faces par thread
int range = faces.size()/nbThreads;
if(range == 0) { // En vitant de faire n'importe quoi
  range = 1;
  nbThreads = faces.size()-1;
}
```

```
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 rempli 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 }
```

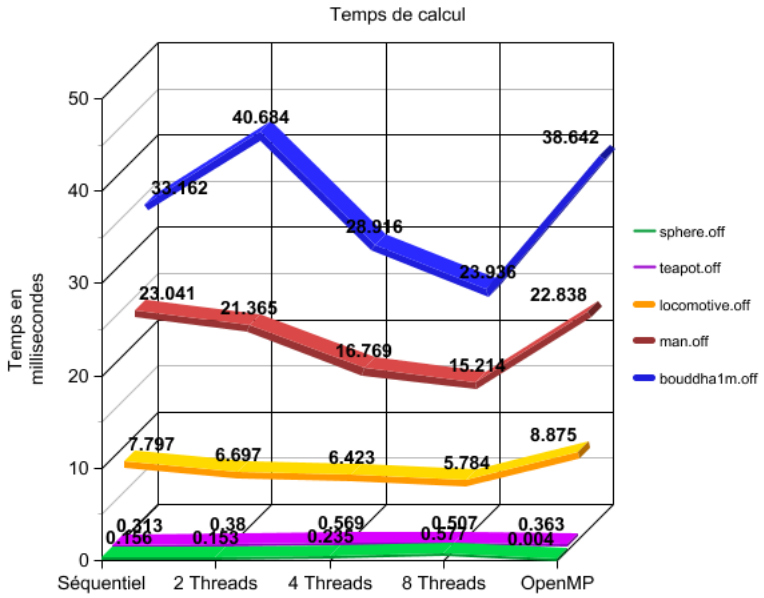
3.2.2 OpenMP

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.

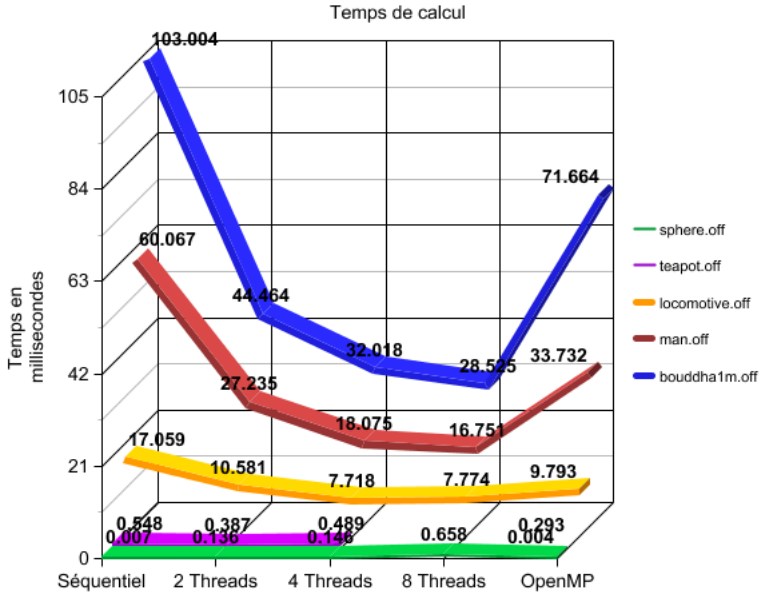
```
1 #pragma omp parallel for reduction ( + : result )
2 double result = 0.f;
3 for(long i = 0; i < faces.size(); i++) {
4     result += Face::computeArea(&faces[i]);
5 }
```

4 Résultats

4.1 Machine 1



4.2 Machine 2



5 Conclusion

D'après les tests que nous avons effectués, nous constatons que le temps de calcul est réduit lors de la parallélisation par threads, le temps d'exécution étant amélioré plus le nombre d'itération (nombre de faces à traiter) est grand.

Nottons que le temps passé à découper le conteneur de faces pour les distribuer aux threads (et, on peut légitimement penser que le scénario est similaire pour l'implémentation OpenMP) est compris dans le temps d'exécution car il est nécessaire au calcul avec threads mais pas au mode séquentiel.

L'implémentation OpenMP ne semble pas appropriée à cette tâche au vu des temps d'exécution équivalents ou peu meilleurs comparées à ceux de l'implémentation séquentielle.

Références

[1] Wikipedia. Spécification du format de fichier off.
`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/`.

[4] L.Aubert et T. Jallois. Dépôt github.
`https://github.com/ZanyMonk/SurfaceMeasurer`.