MAE31 Projet 1

Chuan Qin

March 2018

Introduction

1 Un simulateur de gestion de cache

Question 1.

Pour le produit scalaire de deux vecteurs, on a le résultat :

```
dot = 255.272
current cache
cache size 256 strategy 0
hits 0 misses 2048 hits-ratio 0.000%
Mean acces time 1
```

Pour le produit de deux matrice, on a le résultat :

```
current cache
cache size 256 strategy 0
hits 0 misses 524288 hits-ratio 0.000%
Mean acces time 1
```

Le mode `CACHE_DIRECT_MAPPING` avec le hash par défaut(modulo de l'adresse avec la taille du cache), donne un hits-ratio nul. De plus, pour ces deux applications, il n'y a que 32 blocks(soit 1/8 de blocks) qui sont utilisés, les autres ne sont pas accédés du tout. Comme chaque nombre de double a une longeur de 8 octets, donc on a

```
(long long int)(&(A[i+1])) - (long long int)(&(A[i])) = 8
```

Cela explique pourquoi il n'y que 1/8 de blocks sont utilisé sur les applications. Donc cela est un gros gaspillage, on a besoin de changer la fonction de hash pour ce mode.

Question 2.

Proposer une nouvelle clé de hash

Comme 8 peut divise 256(le nombre de blocs), cela entraîne à seule 1/8 blocs utilisés. Pour éviter ce cas, on peut modifier le dénominateur, soit un nombre impaire. La nouvelle clé de hash (code) est suivante.

```
int code = (long long int)(adr)  % (c->nbrblocks+1) - 1;
if(code <0) code = 0;
```

Dans ce cas, tous les blocs du cache sont utilisés, même si le bloc 0 est plus utilisé(En théorie, son probabilité d'utlisation est 1/256 plus d'autres). Mais le performance ne change pas, il a toujours le hit-ratio nul.

Cela peut être expliquer par la programme d'application. Sur l'application de produit scalaire de deux vecteur, elle n'utilise pas les données répétitif, et donc il n'y a aucun hit quand il n'y a pas un mécanisme de prédiction. Sur l'application de produit de deux matrice, elle utilise les données répétitif, mais la taille de matrice est trop grande pour le taille du cache, qui ne peut pas conserver les données utilisé d'avant.

Clé de Jenkins

la résultat en gros ne change pas, mais sur la deuxième application, il a une petite hit-ratio.

Pour le produit scalaire de deux vecteurs, clé de Jenkins donne

```
current cache
cache size 256 strategy 0
hits 0 misses 524288 hits-ratio 0.000%
Mean acces time 1
```

Pour le produit de deux matrice, clé de Jenkins donne

```
current cache
cache size 256 strategy 0
hits 5449 misses 518839 hits-ratio 1.039%
Mean acces time 1
```

Question 3.

Le résultat de clé de Jenkins en mode `CACHE_DIRECT_MAPPING` est présenté dans la question avant. Le résultat de clé de Jenkins en mode `CACHE_FULLYASSOCIATIVE` est comme suivant.

Pour le produit scalaire de deux vecteurs,

```
current cache
cache size 256 strategy 1
hits 0 misses 2048 hits-ratio 0.000%
Mean acces time 256
```

Pour le produit de matrice et vecteur,

```
current cache
cache size 256 strategy 1
hits 65408 misses 458880 hits-ratio 12.476%
Mean acces time 240.156
```

Analyse

Pour le mode `CACHE_DIRECT_MAPPING`, le temps d'accès est seulement 1, donc trouver le donnée dans le cache est rapide. Mais comme il existe un seule bloc de cache pour chaque adresse, il y a beaucoup de conflits sur le hash, et donc l'efficacité de cache est bas. Par exemple, quand il y a deux données, qui ont l'adresse de A et B, et ils sont beaucoup utilisés sur la programme. Il existe de cas où A et B sont

mappées dans un même bloc, et donc chaque fois il faut les recharger sur le cache, même s'il existe les autres blocs qui n'utilisent pas.

Pour le mode `CACHE_FULLYASSOCIATIVE`, le temps d'accès est beaucoup plus grand, qui est 240.156 en moyen. Mais grace au mécanisme de LRU, le bloc qui garde le données le plus moins utilisé est utilisé pour le nouveau donnée. Donc il a plus de hits-ratio normalement.

Complexités en temps de réponse

Pour le mode `CACHE_DIRECT_MAPPING`, comme chaque adress est associé à un seul bloc, il n'a besoin que de vérifier si le donné est gardé dans ce bloc. Donc ce mode a une compléxité de 1 en temps de réponse.

Pour le mode `CACHE_FULLYASSOCIATIVE`, tous les blocks peuvent stocker les données de chaque adresse. Donc au pire cas, il a besoin n opérations pour vérifier s'il existe un bloc qui garde le donné, où n est le nombre total du bloc. Donc ce mode a une complexité de $\mathcal{O}(n)$ en temps de réponse.

Question 4.

Une nouvelle approche de produit de vecteur vecteur, matrcie vecteur, et matrice matrice est réalisé dans les fonctions `cache_dotprod_block()`, `cache_matvec_block()`, `cache_matmat_block()`.

Dans ces approches, les matrices sont calculé en bloc, dont taille est

$$(m * num\_double\_in\_block)^2$$

où `num_double_in_block` est le nombre de "double" qui remplie un bloc de cache, et m est une coefficient.

Remarque

Comme l'implémentions de `CAHCE_FULLYASSOCIATIVE` n'est pas vrai une approche de RLU, elle ne compte que le nombre de fois qu'un cache est réutilisé, mais elle ne compte pas le temps de cache est utilisé de la dernière fois.

Donc je change l'implementation de `CAHCE_FULLYASSOCIATIVE` pour s'adapter à RLU, ici c'est la changement.

```
else if ( c->strategy == CACHE_FULLYASSOCIATIVE ) {
  int iMin   = INT_MAX;
  int iCode  = 0;
  for(i=0; i<c->nbrblocks; i++) {
    c->accesstime++;
    if ( c->blocks[i] == adr ) {
       c->used[i]++;
       c->hits++;
       c->lastUse[i] = c->accesstime;
       return;
    }
    if ( c->lastUse[i] < iMin ) {
      iMin  = c->lastUse[i];
      iCode = i;
    }
  }
```

```
    c->blocks[iCode] = adr;
    c->used[iCode] = 1;
    c->lastUse[iCode] = c->accesstime;
    c->misses++;
    return;
  }
```

Où $c- > used[iCode]$ stocke le "temps" de la dernière fois utilisé. Ici, je utilise *accesstime* comme le temps approximatif, car il augment monotonement.

L'efficacité

Pour ces simulation, on pense que la taille de bloc est 32 octets, et donc il peut contenir au plus de 4 "double".

Pour le produit scalaire de deux vecteurs en bloc,

```
current cache
cache size 256 strategy 1
hits 1536 misses 512 hits-ratio 75.000 %
Mean acces time 160.375
```

Pour le produit de matrice et vecteur en bloc,

```
current cache
cache size 256 strategy 1
hits 458624 misses 65664 hits-ratio 87.476 %
Mean acces time 144.531
```

Pour le produit de matrice (512*512) et matrice (512*512) en bloc, (la lecture de la matrice résultat AB à chaque pas de mis-à-jour est aussi pris en compte)

```
current cache cache
size 256 strategy 1
hits 276703897 misses 8508775 hits-ratio 97.017 %
Mean acces time 132.106
```

Utiliser le cache-blocking largement augmenter le hits-rate, et donc augmenter la vitess de program, il est très utile.

Du point de vue algorithme, cette approche utilise plus de boucle, et donc plus d'opération de "int" et d'opération comparaison pour le compteur qui contrôle les boucles. Donc si la taille de matrice est petite, cette approche perte du temps.

Question 5.

Dans cette partie, je utilise aussi "lastUse", qui stocke le temps de utilisé précédemment de chaque bloc de cache, pour gérer les conflit. Le bloc (entre 2 ou 4 blocs) qui a le plus vieux "lastUse" est remplacé, car il a plus de temps de n'avais pas utilisé.

`CACHE_2WAYASSOCIATIVE`

Pour le produit scalaire de deux vecteurs de taille 256 en bloc,

```
current cache
cache size 256 strategy 2
hits 1536 misses 512 hits-ratio 75.000 %
Mean acces time 1.572
```

Pour le produit de matrice(256*256) et vecteur en bloc,

```
current cache
cache size 256 strategy 2
hits 423877 misses 100411 hits-ratio 80.848 %
Mean acces time 1.595
```

Pour le produit de matrice (512*512) et matrice (512*512) en bloc, (la lecture de la matrice résultat AB à chaque pas de mis-à-jour est aussi pris en compte)

```
current cache
cache size 256 strategy 2
hits 262851099 misses 22361573 hits-ratio 92.160 %
Mean acces time 1.537
```

`CACHE_4WAYASSOCIATIVE`

Pour le produit scalaire de deux vecteurs de taille 256 en bloc,

```
current cache ]
cache size 256 strategy 3
hits 1536 misses 512 hits-ratio 75.000 %
Mean acces time 2.771
```

Pour le produit de matrice(256*256) et vecteur en bloc,

```
current cache
cache size 256 strategy 3
hits 423675 misses 100613 hits-ratio 80.810 %
Mean acces time 2.789
```

Pour le produit de matrice (512*512) et matrice (512*512) en bloc, (la lecture de la matrice résultat AB à chaque pas de mis-à-jour est aussi pris en compte)

```
current cache
cache size 256 strategy 3
hits 263642240 misses 21570432 hits-ratio 92.437 %
Mean acces time 2.613
```

On peut facilement voir que dans ce cas, `CACHE_2WAYASSOCIATIVE` et `CACHE_4WAYASSOCIATIVE` n'ont pas très grand différence sur le hits-ratio. Dans ce cas, les matrice sont beaucoup plus grande que le cache, donc le cache ne peux pas stocker les donner jusqu'à ils sont réutilisé (sauf les données dans un bloc de A et B).

Par contre, on peux voir que `CACHE_4WAYASSOCIATIVE` a besoin presque 2 fois plus de temps pour accéder les cache par rapport à `CACHE_2WAYASSOCIATIVE`.

Question 6.

Comme "la taille de cache de 32 à 128" est un peu ambigus, je choisit varier la taille de chaque bloc, de 32 à 128 bits. Les résultats sont présenté dans les figures 1 8. Il est facile de voir que le stratégie de `FULLASSOCITIVE` a toujours beacuoup plus de temps d'accès(1), et quand la taille de matrice est plus grande, et la taille de bloc est plus petite, le temps d'accès est plus grand aussi (voir le figure 3 et 2). Pour les autres modes de cache, le temps d'accès est presque stable, mais il existe aussi un petit tendance de la mode `4WAYASSOCIATIVE` tel que quand la taille de matrice diminue, et la taille de bloc augment, le temps d'accès diminue(voir le figure 4).

Pour le hits-ratio, la tendence est plus grande la taille de bloc, plus petite la taille de matrice, plus il est grand (voir le figure 5 et 6). Pour une taille de bloc petite (32), la stratégie influence beaucoup le hits-ratio (voir le figure 7), mais pour une taille de bloc grande, cette influence devient non évident (voir le figure 8).

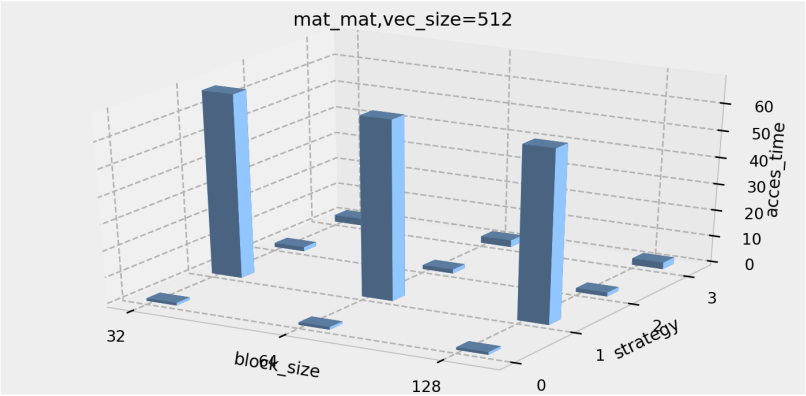

FIGURE 1 – mat_mat,vec_size :512

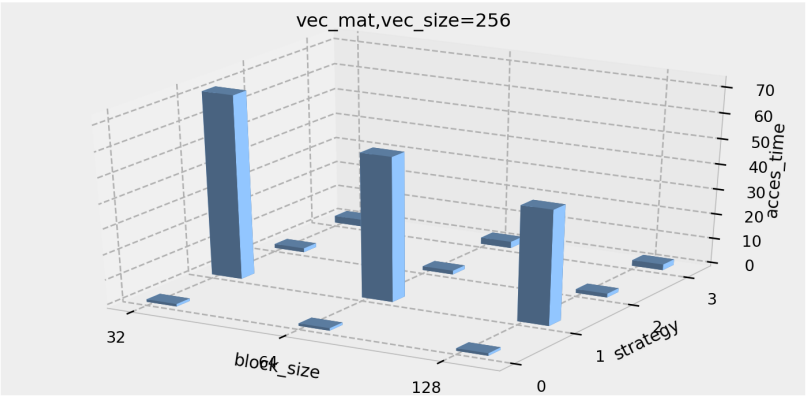

FIGURE 2 – vec_mat,vec_size :256

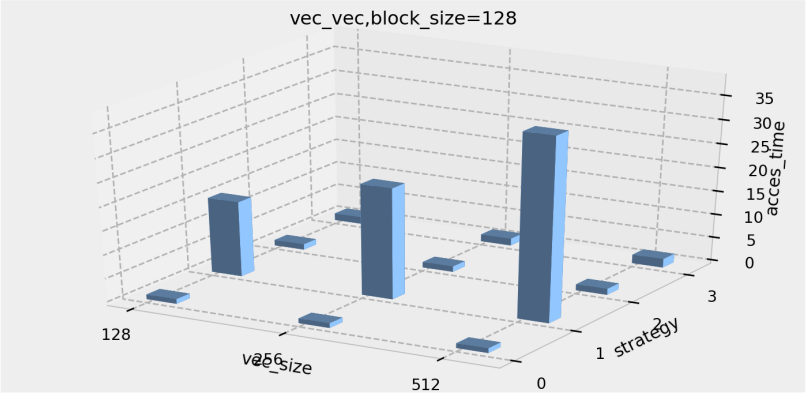

FIGURE 3 – vec_vec,block_size :128

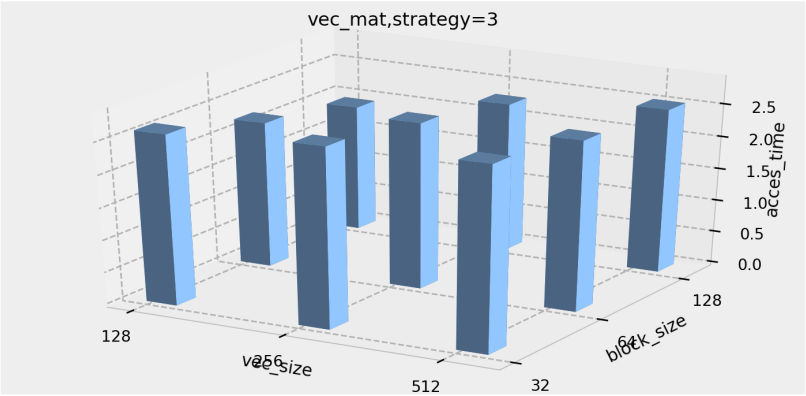

FIGURE 4 – vec_mat,strategy :3

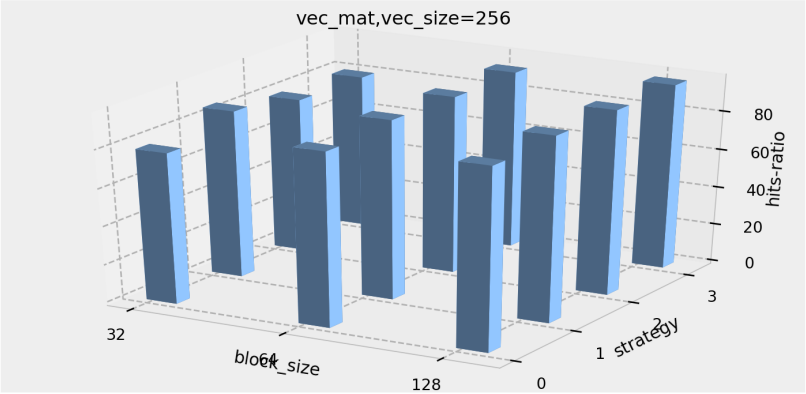

FIGURE 5 – vec_mat,hitsratio,vec_size :256

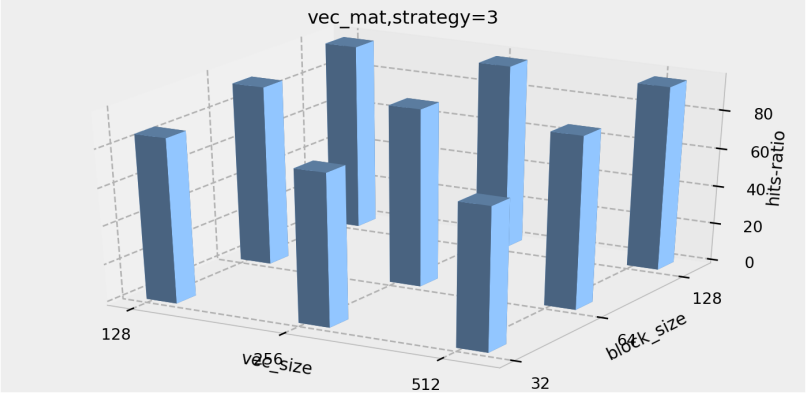

FIGURE 6 – vec_mat_hitsratio_strategy :3

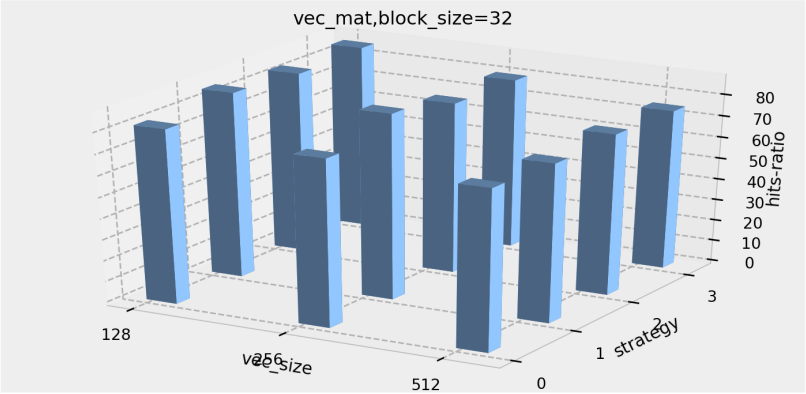

FIGURE 7 – vec_mat,hitsratio,block_size :32

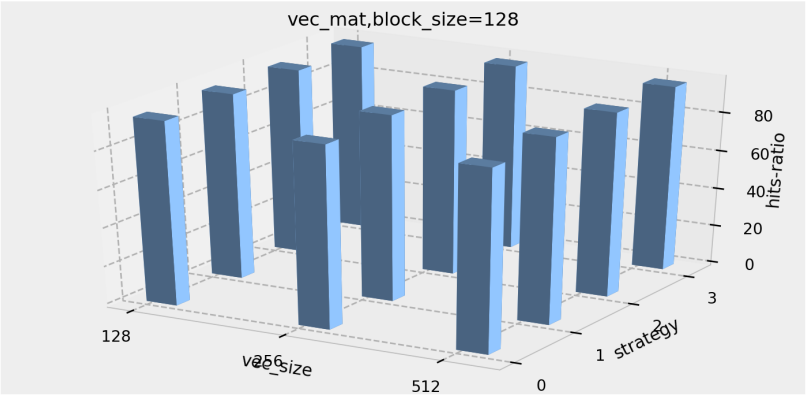

FIGURE 8 – vec_mat,hitsratio,block_size :128

2 Partie II

Question 1

Hilbert le résultat (exécuter en la machine `salle`) :

```
qin@salle:/tmp$ $MESHPATH/optet -in dodecadre06_hilbert.meshb -out tmp.meshb -nproc 1
    Read mesh                 :   1.61 s on 1 core
    Setup neighbours          :   4.13 s on 1 core
    Ball of vertices          :   3.10 s on 1 core(s)
    Compute elements quality  :   6.53 s on 1 core(s)
    Write mesh                :   1.89 s on 1 core
    Nmb Vertices              :  1803338
    Nmb Tetrahedra            : 10526832
```

Z le résultat (exécuter en la machine `salle`) :

```
qin@salle:/tmp$ $MESHPATH/optet -in dodecadre06_z.meshb -out tmp.meshb -nproc 1
    Read mesh                 :   1.92 s on 1 core
    Setup neighbours          :   4.19 s on 1 core
    Ball of vertices          :   2.79 s on 1 core(s)
    Compute elements quality  :   6.52 s on 1 core(s)
    Write mesh                :   1.77 s on 1 core
    Nmb Vertices              :  1803338
    Nmb Tetrahedra            : 10526832
```

random le résultat (exécuter en la machine `salle`) :

```
qin@salle:/tmp$ $MESHPATH/optet -in dodecadre06_random.meshb -out tmp.meshb -nproc 1
    Read mesh                 :   1.67 s on 1 core
    Setup neighbours          :  11.09 s on 1 core
    Ball of vertices          :   6.87 s on 1 core(s)
    Compute elements quality  :   8.17 s on 1 core(s)
    Write mesh                :   2.51 s on 1 core
    Nmb Vertices              :  1803338
    Nmb Tetrahedra            : 10526832
```

Remarque

Le temps d'exécution de Hilbert et en Z est dans meme ordre, mais le temps de renuméroter aléatoire donne une résultat très mal, il utilise plusieur fois de temps, notament sur le partie de "Setup neighbours" et "Ball of vertices". Dans cette partie, on peut conclure que la rénumérotation de sommets, triangles, et tétraèdres influence beaucoup de performance.

Question 2.

Dans cette question, on utilise le "z-path" pour renuméroter le maillage. On utilise une approche efficace en manipulant les bits directement.

```
//www.forceflow.be/2013/10/07/
//     morton-encodingdecoding-through-bit-interleaving-implementations/
    icrit_temp = 0;
    for (i = 0; i < (sizeof(unsigned long long int)* CHAR_BIT)/3; ++i) {
      icrit_temp |= ((x_index & ((unsigned long long int)1 << i)) << 2*i)
        | ((y_index & ((unsigned long long int)1 << i)) << (2*i + 1))
        | ((z_index & ((unsigned long long int)1 << i)) << (2*i + 2));
    }
    msh->Ver[iVer].icrit  = icrit_temp;
```

Dans cette implementation,`icrit_temp` est une variable de `int64`, et donc 21bits pour chaque coordonnée (x ou y ou z). Donc il faut adapter l'échelle de la maillage, j'utilise une proportion adapté pour contrôler l'échelle automatique.

Après trier et changer l'ordre de sommets en utilisant `icrit_temp`, la variable `Ver[]` de chaque triangles ou tétraèdre doit aussi change. Pour éviter plusieur loop, ici, je utilise un tableau `newIndex` dont l'indice est la position vieux, et dans laquelle la variable stocké est la position nouvelle. Donc l'étape de changement est simple comme suivante.

```
for(iTri=1; iTri<=msh->NbrTri; iTri++) {
  for(j =0; j < 3; j++){
    temp_index = msh->Tri[iTri].Ver[j];
    msh->Tri[iTri].Ver[j] = newIndex[temp_index];
  }
}
```

Pour la rénumérotation de triangles (ou tétraèdre), j'utilise la coordonnée de centre de chaque triangles (ou tétraèdre) pour calculer le `icrit_temp` qui va être trié suivant.

Le résultat est affiché ci-dessous (exécuter en la machine `salle`).

```
qin@salle:/tmp$ $MESHPATH/optet -in output.meshb -out tmp.meshb -nproc 1
   Read mesh                 :    1.93 s on 1 core
   Setup neighbours          :    4.19 s on 1 core
   Ball of vertices          :    3.06 s on 1 core(s)
   Compute elements quality  :    6.52 s on 1 core(s)
   Write mesh                :    1.58 s on 1 core
   Nmb Vertices              :  1803338
   Nmb Tetrahedra            : 10526832
```

On voit que ce résultat est prèsque le meme que celle précédente de la éxécutable (`hilbert`), qui est beaucoup mieux que la renumérotation aléatoire.

Question 3.

Je compléter la fonction de `msh_neighborsQ2()` comme suivante,

```
for(iTet=1; iTet<=msh->NbrTet; iTet++) {
  for(iFac=0; iFac<4; iFac++) {
    ip1 = msh->Tet[iTet].Ver[lnofa[iFac][0]];
    ip2 = msh->Tet[iTet].Ver[lnofa[iFac][1]];
    ip3 = msh->Tet[iTet].Ver[lnofa[iFac][2]];
    /* find the Tet different from iTet that has ip1, ip2, ip2 as vertices */
    break_flag = 0;
    //printf("debug 0\n");
    for(jTet=1; jTet<=msh->NbrTet;jTet++) {
      if(msh->Tet[iTet].Ver[0]){
        continue;
      }
      if ( iTet == jTet ) continue;
      if (break_flag == 1) break;
      for(jFac=0; jFac<4; jFac++) {
        jp1 = msh->Tet[jTet].Ver[lnofa[jFac][0]];
        jp2 = msh->Tet[jTet].Ver[lnofa[jFac][1]];
        jp3 = msh->Tet[jTet].Ver[lnofa[jFac][2]];
        /* compare the 6 points */
        if(ip1 == jp1 && ip2 == jp2 && ip3 == jp3){
          printf("find␣tet␣neighbors␣of␣%d␣:␣%d\n", iTet, jTet);
          msh->Tet[iTet].Voi[iFac] = jTet;
          break_flag = 1;
          break;
        }
```

```
        }
      }
    }
  }
```

Mais comme il y a deux boucles, cette implémentation consomme trop de temps, et donc je ne réussi pas à atteindre le finir.

Question 4.

Dans cette parite, on utilise une approche de Hashtable pour accélérer de trouver les voisinages.

Pour le clé de hashtable, je choisi la fonction suivante

```
SizeHead = 9999991;
key = (3*ip1 + 5*ip2 + 7*ip3)%SizeHead
```

où 3, 5, 7, 9999991 sont tous nombres premiers.

Le résultat de mon implémentation est comme suivante

```
time to re-order the mesh  7.53306 (s)
time hash tab neigh.  4.23801 (s)
```

Donc le temps de calculer les voisinages est très similaire à le temps de "Setup neighbours" qu'avant.

Les fonctions que j'ajoute associés à la Hashtable est comme suivantes.

```
HashTable* hash_init(int SizeHead, int NbrMaxObj){
  HashTable* ht = (HashTable*)calloc(1,sizeof(HashTable));
  ht->SizeHead = SizeHead;
  ht->NbrMaxObj = NbrMaxObj;
  //ht->Head = (int*)calloc(SizeHead,sizeof(int));
  ht->LstObj = (int*)calloc(NbrMaxObj, sizeof(int6));
  ht->NbrObj = ht->SizeHead;
  int i=0;
  return ht;
}

int is_equal(int *a, int b1, int b2, int b3){
  //a[0:2]
  int i =0;
  int count =0;
  for(i=0;i<3;i++){
    if(b1 == a[i]) count++;
    if(b2 == a[i]) count++;
    if(b3 == a[i]) count++;
  }
  if(count == 3) return 1;
  else return 0;
}

int hash_find(HashTable *hsh, int ip1, int ip2, int ip3, int iTet,int debug_switch)
{
  int key = (3*ip1 + 5*ip2 + 7*ip3);
  //printf("key: %d\n", key);
  int head = key%hsh->SizeHead;
  int old_head;
  while(1){
    if(is_equal(hsh->LstObj[head],ip1,ip2,ip3)){
      if(iTet >=0)
         hsh->LstObj[head][4] = iTet;
      return head;
    }

    else{
```

```
      old_head = head;
      head = hsh->LstObj[head][5];
      if(head == 0 ) break;
      if(debug_switch==1)
        printf("head: %d\n", head);
    }

  }
  return -1;

}
void hash_add(HashTable *hsh, int ip1, int ip2, int ip3, int iTet1)
//===> add this entry in the hash tab
{
  int key = (3*ip1 + 5*ip2 + 7*ip3) ;
  int head = key%hsh->SizeHead;
  int write_index;
  if(hsh->LstObj[head][0] <=0){
    write_index = head;
  }
  else{
    while(hsh->LstObj[head][5]>0)
      head = hsh->LstObj[head][5];

    hsh->LstObj[head][5] = hsh->NbrObj+1;
    write_index = hsh->NbrObj+1;
    hsh->NbrObj ++;
  }

  hsh->LstObj[write_index][0] = ip1;
  hsh->LstObj[write_index][1] = ip2;
  hsh->LstObj[write_index][2] = ip3;
  hsh->LstObj[write_index][3] = iTet1;
}
```