Projet de Programmation Multi-GPU

Elliott Blot, Mathieu Lasjaunias

29 avril 2016

Introduction

Le projet vise à paralléliser l'éboulement d'un "tas de sable abélien". On dispose d'une table virtuelle de dimension DIM $\times$ DIM et dont les cases sont capables d'accueillir seulement 4 grains de sable. Dans le cas contraire le pile de sable de cette case s'écroule sur ses voisines et à une échelle plus globale le tas de sable s'effondre.

FIGURE 1 – Un tas de sable

Règles d'éboulement

On travaille en 4-connexité : deux cases sont connexes si elles sont adjacentes par un de leurs 4 bords que nous appellerons Nord, Sud, Est, Ouest.

S'il y a au moins quatre grains dans une case donnée, un éboulement se produit : la case se vide de l'essentiel de ses grains qui sont répartis dans les quatre cases voisines équitablement. Plus formellement, s'il y a $n > 3$ grains dans une case, chacune des 4 cases voisines reçoit le quotient entier de n par 4 et la case initiale garde le reste de la division euclidienne.

Que se passe-t-il au bord ? Un des grains est définitivement perdu, ou deux s'il s'agit d'une case située dans l'un des coins.

Le lecteur attentif se doute qu'un éboulement peut déclencher de nouveaux éboulements de proche en proche : une avalanche peut ainsi se produire. Le fait que des grains disparaissent quand l'avalanche atteint le bord de la table (en supposant qu'elle parvienne jusque là) assure que ce processus finisse par s'arrêter i. e. se stabiliser. On atteint ainsi une configuration stable dans le sens où chaque case contient au maximum 3 grains.

Une question se pose assez naturellement : quel rôle joue l'ordre dans lequel on procède aux éboulements ? En effet, à un instant donné, plusieurs cases peuvent être instables et il faut bien (en séquentiel) choisir un ordre dans lequel opérer les éboulements. On peut démontrer que la configuration finale ne dépend pas de l'ordre de ces éboulements : les avalanches « commutent » entre elles ! (C'est de là que vient le qualificatif "abélien" du modèle.)

Simulation synchrone

Après avoir tenté d'optimiser l'algorithmique du programme nous passons à la partie *Simulation synchrone* i.e. les threads se synchronisent à chaque itération.

Ci-dessous la version algorithmique naïve donné dans le sujet.

Listing 1 – version séquentielle 0 (donnée)

```
float *compute (unsigned iterations)
  {
    static int step = 0;
    for (unsigned i = 0; i < iterations; i++)
      {
      step++;
      for (int x = 1; x < DIM-1; x++)
        for (int y = 1; y < DIM-1; y++)
          {
            if (table[x][y] >= 4)
              {
                int mod4 = table[x][y] % 4;
                int div4 = table[x][y] / 4;
                table[x][y] = mod4;
                table[x-1][y] += div4;
                table[x+1][y] += div4;
                table[x][y-1] += div4;
                table[x][y+1] += div4;
              }
          }
//print_tab();
     }
return DYNAMIC_COLORING; // altitude-based coloring
// return couleurs;
}
```

et ici une tentative d'optimisation algorithmique : on distribue les grains un par un sur les quatre cellules voisines.

Listing 2 – version séquentielle 1 (progressive)

```
float *compute (unsigned iterations)
  {
    static int step = 0;
    for (unsigned i = 0; i < iterations; i++)
      {
        step++;
        for (int x = 1; x < DIM-1; x++)
          for (int y = 1; y < DIM-1; y++)
            {
              if (table[x][y] >= 4)
                {
                  table[x][y] -= 4;
                  table[x-1][y] += 1;
                  table[x+1][y] += 1;
                  table[x][y-1] += 1;
                  table[x][y+1] += 1;
                }
            }
//print_tab();
      }
return DYNAMIC_COLORING; // altitude-based coloring
// return couleurs;
}
```

On constate, en testant ces deux versions, que seq1 est un peu plus rapide pour le premier cas à tester, c'est à dire la configuration plate (on peut penser qu'elle est plus rapide pour les cas homogènes) mais est beaucoup plus lente sur le deuxième cas, c'est à dire la configuration pointue (environ 6 fois plus lente dans la salle 203 du cremi).

Nous nous sommes donc davantage intéressé à paralléliser seq0 par la suite, car elle est plus polyvalente, même si légèrement moins rapide pour le cas homogène (configuration plate).

Programmation parallèle CPU

On passe maintenant à la programmation parallèle à "proprement" parler.

Synchronisation des threads à chaque itération

Voici nos différentes versions parallèles OpenMP qui synchronisent les threads à chaque itération. Nous donnons une courte description de chaque version après leur code respectif.

Cette version utilisant les tâches OpenMP est très mauvaise car très lentes. Elle attend que les deux cases précédentes aient été traité avant de traiter une case du tableau.

Listing 3 – version parallèle 1 (task)

```
float *compute (unsigned iterations)
  {
    int x,y;
    static int step = 0;
    for (unsigned i = 0; i < iterations; i++)
      {
        step++;
#pragma omp parallel
#pragma omp master
        for (x = 1; x < DIM-1; x++)
          for (y = 1; y < DIM-1; y++)
            {
              if(x==1 && y == 1){
#pragma omp task depend(out : table[1][1]) firstprivate(x,y)
                if (table[x][y] >= 4)
                  {
                    int mod4 = table[x][y] % 4;
                    int div4 = table[x][y] / 4;
                    table[x][y] = mod4;
                    table[x-1][y] += div4;
                    table[x+1][y] += div4;
                    table[x][y-1] += div4;
                    table[x][y+1] += div4;
                  }
              }
              else if(x==1 && y != 1){
#pragma omp task depend(in : table[1][y-1]) depend(out : table[x][y
    ]) firstprivate(x,y)

                if (table[x][y] >= 4)
                  {
                    int mod4 = table[x][y] % 4;
                    int div4 = table[x][y] / 4;
                    table[x][y] = mod4;
                    table[x-1][y] += div4;
                    table[x+1][y] += div4;
                    table[x][y-1] += div4;
                    table[x][y+1] += div4;
                  }
              }
              else if(x!=1 && y == 1){
#pragma omp task depend(in : table[x-1][1]) depend(out : table[x][y
    ]) firstprivate(x,y)

                if (table[x][y] >= 4)
                  {
                    int mod4 = table[x][y] % 4;
                    int div4 = table[x][y] / 4;
                    table[x][y] = mod4;
                    table[x-1][y] += div4;
                    table[x+1][y] += div4;
                    table[x][y-1] += div4;
                    table[x][y+1] += div4;
                  }
              }
              else if(x!=1 && y != 1){
```

```
#pragma omp task depend(in : table[x-1][y], table[x][y-1]) depend(
    out : table[x][y]) firstprivate(x,y)

                    if (table[x][y] >= 4)
                      {
                        int mod4 = table[x][y] % 4;
                        int div4 = table[x][y] / 4;
                        table[x][y] = mod4;
                        table[x-1][y] += div4;
                        table[x+1][y] += div4;
                        table[x][y-1] += div4;
                        table[x][y+1] += div4;
                      }
                    }
#pragma omp taskwait
                }
      }
return DYNAMIC_COLORING; // altitude-based coloring
// return couleurs;
}
```

Maintenant on passe à la deuxième version où chaque thread a son tableau et enregistre les écoulements dans celui ci avant de l'ajouter à la fin au tableau initial (avec un atomic pour éviter les conflits).

Listing 4 – version parallèle 2 (chaque thread sa copie)

```
float *compute (unsigned iterations)
{
    int x,y,M;
    int tranche;
    int copy[DIM][DIM];
    static int step = 0;
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            copy[ik][jk] = 0 ;
#pragma omp parallel
{
    int id,jd;
    int copy_thread[DIM][DIM];
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            copy_thread[ik][jk] = 0 ;
#pragma omp for schedule(dynamic) private(y)
    for (x = 1; x < DIM - 1;x++)
        for (y = 1; y < DIM - 1; y++)
        {
            if (table[x][y] >= 4)
            {
                int mod4 = table[x][y] % 4;
                int div4 = table[x][y] / 4;
                table[x][y] = mod4;

                copy_thread[x-1][y] += div4;
                copy_thread[x][y-1] += div4;
                copy_thread[x][y+1] += div4;
                copy_thread[x+1][y] += div4;
            }
        }
    for (id = 0; id < DIM;id++)
        for (jd = 0; jd< DIM; jd++)
            if(copy_thread[id][jd] != 0)
#pragma omp atomic
                table[id][jd] += copy_thread[id][jd];
}
    return DYNAMIC_COLORING; // altitude-based coloring
    //return couleurs;
}
```

Le problème avec cette version est que le *#pragma omp atomic* ralenti le programme(c'est bien pire avec *#pragma omp critical*), donc on a essayé par la suite de ne pas utiliser cette commande.

À présent une version "absorbante" : la grille est copié dans *copy* et pour chaque case de la table on enlève ce qui doit s'écouler puis on ajoute les écoulements en les calculant à partir de *copy* (qui reste inchangé donc il n'y a pas de conflit).

Listing 5 – version parallèle 4 (absorbante)

```
float *compute (unsigned iterations)
  {
    int x,y,M;
    int tranche;
    int copy[DIM][DIM];
    static int step = 0;
    for (unsigned i = 0; i < iterations; i++)
      {
        for (int ik=0; ik < DIM;ik++)
          for (int jk=0; jk < DIM;jk++)
            copy[ik][jk] = table[ik][jk] ;
        step++;
#pragma omp parallel for firstprivate(y)
        for (x = 1; x < DIM - 1;x++)
          for (y = 1; y < DIM - 1; y++)
            {
              table[x][y] = table[x][y] % 4;
              table[x][y] += copy[x-1][y] / 4;
              table[x][y] += copy[x+1][y] / 4;
              table[x][y] += copy[x][y-1] / 4;
              table[x][y] += copy[x][y+1] / 4;
            }
      }
    return DYNAMIC_COLORING; // altitude-based coloring
//return couleurs;
}
```

Cette version est assez efficace mais ne nous permet pas de synchroniser les thread après p itérations, et donc d'avoir une accélération supplémentaire, il faudrais pour cela essayer de separer le tableau en bande, comme nous le verrons dans la version par3 (7).

La prochaine version utilise quatre tableaux (un pour chaque direction). On enregistre tous les écoulements qui vont vers la droite dans *right*, etc, et à la fin on ajoute les quatre tableaux à la table.

Listing 6 – version parallèle 5 (4 copie de tableaux)

```
float *compute (unsigned iterations)
{
    int x,y,M;
    int tranche;
    int up[DIM][DIM];
    int down[DIM][DIM];
    int right[DIM][DIM];
    int left[DIM][DIM];
    static int step = 0;
    //for (unsigned i = 0; i < iterations; i++)
    //{
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            up[ik][jk] = 0;
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            down[ik][jk] = 0;
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            right[ik][jk] = 0;
    for (int ik=0; ik < DIM;ik++)
        for (int jk=0; jk < DIM;jk++)
            left[ik][jk] = 0;

    //step++;
#pragma omp parallel
{
#pragma omp for firstprivate(y)
    for (x = 1; x < DIM - 1;x++)
        for (y = 1; y < DIM - 1; y++)
        {
            if (table[x][y] >= 4)
            {
                int mod4 = table[x][y] % 4;
                int div4 = table[x][y] / 4;
                table[x][y] = mod4;
                up[x-1][y] = div4;
                down[x+1][y] = div4;
                left[x][y-1] = div4;
                right[x][y+1] = div4;
            }
        }
#pragma omp for
    for (int ia = 0; ia < DIM ;ia++)
        for (int ja = 0; ja < DIM ; ja++)
        {
            table[ia][ja] = table[ia][ja] + up[ia][ja] + down[ia][
    ja] + left[ia][ja] + right[ia][ja];
        }
}

    return DYNAMIC_COLORING; // altitude-based coloring
```

```
    // return couleurs;
}
```

Cette version est la plus lente des quatre (sauf la version avec les tâches) montré dans cette sections, et elle prend beaucoup de mémoire(a cause de toute les copie de tableau) mais nous a permis de trouver une alternative au *#pragma omp critical* pour la prochaine version, la plus efficace des cinq présentées.

Cette troisième version est la meilleur version que l'on ait actuellement, elle permet d'obtenir un speed up d'environ 1,4 par rapport à la version seq0, en utilisant le bon nombre de threads (qui est 8 en salle 203).

Elle s'inspire de la troisième version (copie de 4 tableaux) et le principe est le suivant : on coupe le tableau en tranches en fonction du nombre de thread, et chaque thread travaille sur sa tranche. si le thread est au bord de sa tranche, il copie l'écoulement dans un tableau séparé (un à droite (r) et un à gauche (l)) puis les tableaux sont additionnés (en séquentiel car les tests effectués montraient plus de rapidité).

Listing 7 – version parallèle 3 (copie de 2 tableaux)

```
float *compute (unsigned iterations)
  {
    int x,y,M,ja;
    int tranche;
    int l[DIM][DIM];
    int r[DIM][DIM];
    static int step = 0;
    for (unsigned i = 0; i < iterations; i++)
      {
        for (int ik=0; ik < DIM;ik++)
          for (int jk=0; jk < DIM;jk++)
            l[ik][jk] = 0;
        for (int ik=0; ik < DIM;ik++)
          for (int jk=0; jk < DIM;jk++)
            r[ik][jk] = 0;
#pragma omp parallel
    {
#pragma omp single
    {
      tranche = (DIM / (omp_get_num_threads()));
      if ((DIM % omp_get_num_threads()) !=0)
        tranche ++;
    }
#pragma omp for schedule(static,tranche) private(x,y)
    for (x = 1; x < DIM - 1; x++)
      for (y = 1; y < DIM - 1; y++)
        {
          if (x == 0)
            break;
          if (table[x][y] >= 4){
            if ((x % tranche) == 1)
              {
                int mod4 = table[x][y] % 4;
                int div4 = table[x][y] / 4;
                table[x][y] = mod4;
                l[x-1][y] += div4;
                table[x+1][y] += div4;
                table[x][y-1] += div4;
                table[x][y+1] += div4;
              }
              else if ((x % tranche) == 0)
                {
                  int mod4 = table[x][y] % 4;
                  int div4 = table[x][y] / 4;
```

```
                    table[x][y] = mod4;
                    table[x-1][y] += div4;
                    r[x+1][y] += div4;
                    table[x][y-1] += div4;
                    table[x][y+1] += div4;
                  }
                else
                  {
                    int mod4 = table[x][y] % 4;
                    int div4 = table[x][y] / 4;
                    table[x][y] = mod4;
                    table[x-1][y] += div4;
                    table[x+1][y] += div4;
                    table[x][y-1] += div4;
                    table[x][y+1] += div4;
                  }
              }
          }
      }
      for (int ia = 0; ia < DIM -1;ia++)
        for (ja = 0; ja < DIM -1; ja++)
          {
             table[ia][ja] += l[ia][ja] + r[ia][ja];
          }
      }
return DYNAMIC_COLORING; // altitude-based coloring
// return couleurs;
}
```

On améliorera cette version par la suite, en faisant en sorte que les threads se synchronise qu'après p iteration'.

Voici maintenant les courbes de speed-up correspondantes aux configurations 1 et 2 respectivement. On rappelle que la configuration 1 (ou plate) consiste en 5 grains de sable sur toutes les cases de la table tandis que la configuration 2 (ou pointue) correspond à celle où les cases sont vides sauf la case centrale qui contient 10^5 grains empilés.

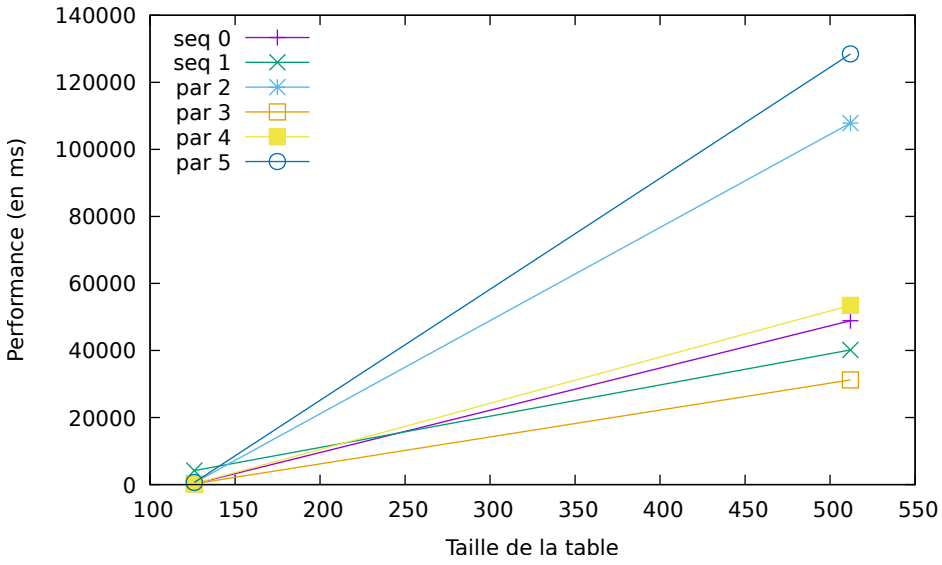

FIGURE 2 – Configuration plate

On voit que la troisième version parallèle est pour l'instant la seule plus efficace que les versions séquentielles tandis que la version 4 arrive presque à leur hauteur. La version parallèle 1 n'est pas représenté ici car trop lente par rapport à toutes les autres.

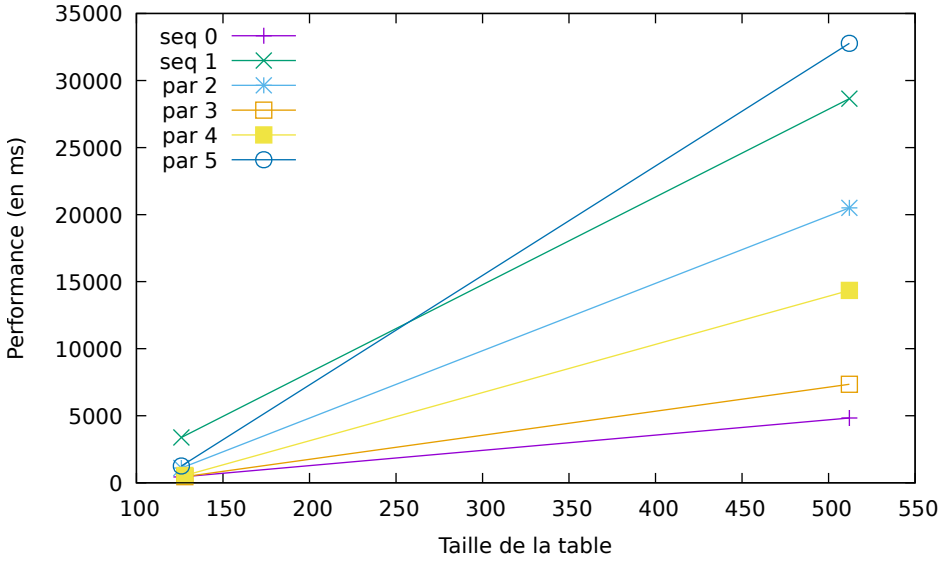

FIGURE 3 – Configuration pointue

Ici on retrouve à peu près les résultats de la configuration plate mis à part que voit que la version séquentielle 'optimisé' (seq 1 ou "progressive") n'est pas du tout adapté pour cette configuration pointue. La version seq0 reste la plus efficace pour ce cas.

Maintenant on peut regarder la version parallèle la plus efficace (version 3 ou "copie de 2 tableaux"). Voici les résultats pour la table de taille 128.

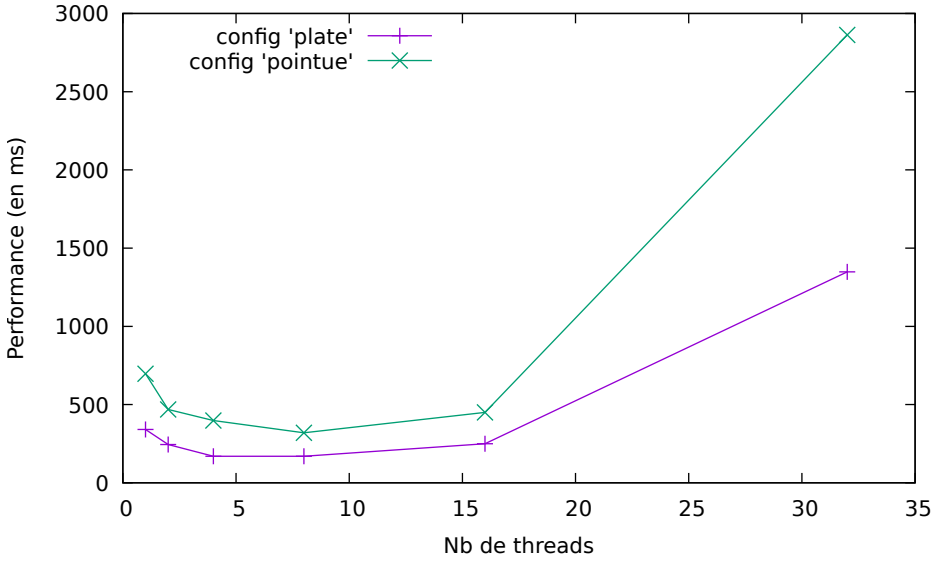

FIGURE 4 – Table de 128 par 128

et à présent les résultats pour la table de taille 512.

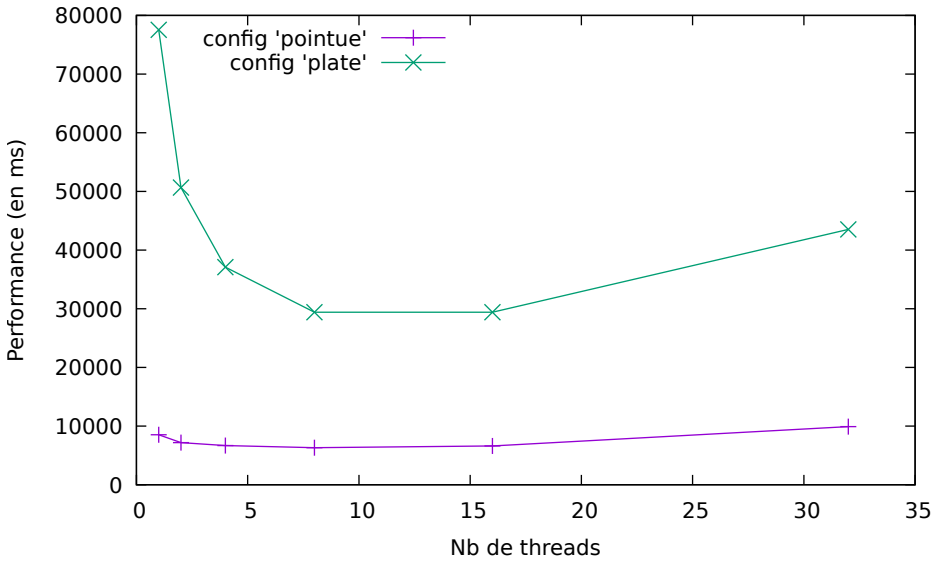

FIGURE 5 – Table de 512 par 512

On peut observer que le meilleur nombre de threads est autour de 8. On arrive à dépasser la version seq0 pour le premier cas avec 8 threads(seq0 : 231 ms et par3 : 170 ms quand DIM = 128, seq0 : 48888 ms et par3 : 29402 ms quand DIM = 512), cependant on y arrive pas avec le deuxieme cas avec une grande dimension (seq0 : 441 ms et par3 : 320 ms quand DIM = 128 ms ,seq : 4832 ms et par3 : 6312 ms quand DIM = 512).

Synchronisation des threads toutes les p itérations

Voici la sous-section traitant des versions parallèles avec synchronisation des threads toutes les p itérations.

Listing 8 – version 6 (*itérations*-synchronisation)

```
float *compute (unsigned iterations)
{
  int x,y,M,ja;
  int tranche;
  int l[DIM][DIM];
  int r[DIM][DIM];
  static int step = 0;
  for (int ik=0; ik < DIM;ik++)
    for (int jk=0; jk < DIM;jk++)
      l[ik][jk] = 0;
  for (int ik=0; ik < DIM;ik++)
    for (int jk=0; jk < DIM;jk++)
      r[ik][jk] = 0;
  step++;
#pragma omp parallel
  {
    for (unsigned i = 0; i < iterations; i++)
      {
#pragma omp single
  {
    tranche = (DIM / (omp_get_num_threads()));
    if ((DIM % omp_get_num_threads()) !=0)
      tranche ++;
  }
#pragma omp for schedule(static,tranche) private(x,y)
  for (x = 1; x < DIM - 1; x++)
    for (y = 1; y < DIM - 1; y++)
      {
        if (x == 0)
        break;
        if (table[x][y] >= 4){
        if ((x % tranche) == 1)
          {
            int mod4 = table[x][y] % 4;
            int div4 = table[x][y] / 4;
            table[x][y] = mod4;
            l[x-1][y] += div4;
            table[x+1][y] += div4;
            table[x][y-1] += div4;
            table[x][y+1] += div4;
```

```
          }
        else if ((x % tranche) == 0)
          {
            int mod4 = table[x][y] % 4;
            int div4 = table[x][y] / 4;
            table[x][y] = mod4;
            table[x-1][y] += div4;
            r[x+1][y] += div4;
            table[x][y-1] += div4;
            table[x][y+1] += div4;
          }
        else
            {
            int mod4 = table[x][y] % 4;
            int div4 = table[x][y] / 4;
            table[x][y] = mod4;
            table[x-1][y] += div4;
            table[x+1][y] += div4;
            table[x][y-1] += div4;
            table[x][y+1] += div4;
          }
          }
      }
      }
  }
  for (int ia = 0; ia < DIM -1;ia++)
    for (ja = 0; ja < DIM -1; ja++)
      {
  table[ia][ja] += l[ia][ja] + r[ia][ja];
      }
  return DYNAMIC_COLORING; // altitude-based coloring
  // return couleurs;
}
```

Les meilleurs p pour cette version sont 4 pour les tables de tailles 128 et 16 pour les tables de tailles 512 (ceci pour les configurations plates et pointues). On arrive a battre tout nos record précédents avec cette version avec pour la taille 128 :

- cas 1 : 158 ms (speedup de 1.5 par rapport à seq0 et de 1.1 par rapport à par3)
- cas 2 : 290 ms (speedup de 1.5 par rapport à seq0 et de 1.1 par rapport à par3)

et pour la taille 512 :

- cas 1 : 11615 ms (speedup de 3.7 par rapport à seq0 et de 2.5 par rapport à par3)
- cas 2 : 2068 ms (speedup de 2.3 par rapport à seq0 et de 3 par rapport à par3)

Programmation GPU en OpenCL

Voici le noyau OpenCL correspondant à la fonction *compute*.

Listing 9 – version OpenCL

```
__kernel void comp(__global int *table,
                   __global int *copie,
                __global int *output)
{
  int x = get_global_id(1);
  int y = get_global_id(0);

  int l = 0;
  int r = 0;
  int u = 0;
  int d = 0;

  int div = table[y*SIZE+x] % 4;


    l = copie[(y-1)*SIZE+x]/4;
    r = copie[(y+1)*SIZE+x]/4;
    u = copie[y*SIZE+x-1]/4;
    d = copie[y*SIZE+x+1]/4;

  output[y*SIZE+x] = div + l + r + u + d;
}
```

On a pus déterminer que le noyaux fait bien ce qui est demandé, mais cette version est vraiment beaucoup trop lente pour qu'on en parle d'avantage.

Simulation asynchrone

Carte blanche.

Conclusion

Pour plus de renseignements sur la théorie des tas de sables abéliens le lecteur intéressé pourra se référer à [Bak et al., 1987] ainsi qu'à [Pegden and Smart, 2011].

Bibliographie

[Bak et al., 1987] Bak, P., Tang, C., and Wiesenfeld, K. (1987). Self-organized criticality : An explanation of the $1/f$ noise. *Phys. Rev. Lett.*, 59 :381–384.

[Pegden and Smart, 2011] Pegden, W. and Smart, C. K. (2011). Convergence of the Abelian sandpile. *ArXiv e-prints*.