Projet de programmation multi-cœurs et GPU : tas de sables.

Pierre Montagne & Louis Nebout

1. Présentation de l'ensemble du projet.

Lors de notre projet, pour des soucis de simplicité, nous avons décidé de séparer dans différents fichiers nos différentes fonctions. Pour répondre aux consignes nous avons donc créé les fonctions suivantes:

- *naive*, la version asynchrone de base.
- *naive_sync*, la version naïve synchrone de base.
- *naive_openmp*, une version asynchrone parallélisée.
- *absorb*, la version synchrone séquentielle optimisée
- *absorb_openmp*, la version synchrone OpenMP
- *runtime_absorb_openmp*, une version OpenMP à schedule modifiable
- *numa_runtime_absorb_openmp*, une version OpenMP initialisant les tableaux de façon parallèle.
- *gpu*, une version basiqueOpenCL
- *gpu_overlap*, une seconde version OpenCL, plus sophistiquée
- *task.seq*, une version asynchrone utilisant une pile de tâches
- *outward_naive_sync*, une version asynchrone ratée

Toutes nos fonctions contiennent les mêmes options qui sont :

- -t : décide de l'initialisation de la grille de départ. Par défaut la grille sera initialisée de façon homogène, avec 5 grains de sables par case. Si l'option est utilisée, on travaille alors sur une tour de grains de sables sur la case centrale, la taille de la tour étant rentrée en paramètre.

- -i : règle le nombre d'itérations à effectuer à chaque appel de la fonction centrale.

- -g : pour utiliser l'interface graphique.

- -c : qui vérifie que la fonction appelée donne le résultat correct.

- -l et -h : les versions gpu proposent ces deux options supplémentaires, permettant de spécifier les dimensions des workgroups utilisés : *{l, h}*. Attention, des valeurs non valides empêchent les kernels de se lancer.

- -b : le programme *gpu_overlap* calcule des itérations k-synchrones, où *k* est donné par cette option.

La dimension des tas de sable étudiés est à préciser au moment de la compilation, par exemple avec la commande "*DIM=512 make*". La dimension par défaut est de 128.

2. Les différentes versions naïves.

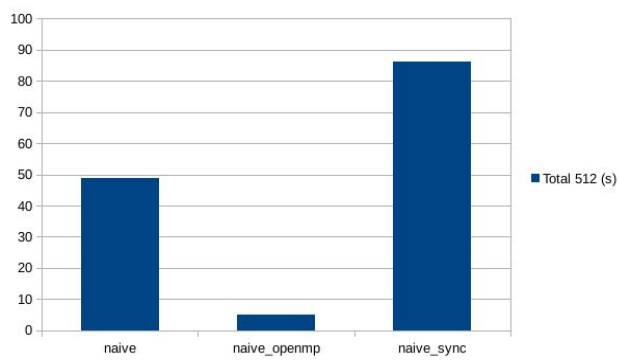
Pour débuter, nous avons utilisé l'exemple donné dans le sujet. La fonction *naive* est une implémentation directe de cet exemple donnée dans le sujet. On examine chaque case de la grille, en la faisant s'écrouler sur ses voisins si elle est trop pleine. C'est un programme asynchrone.

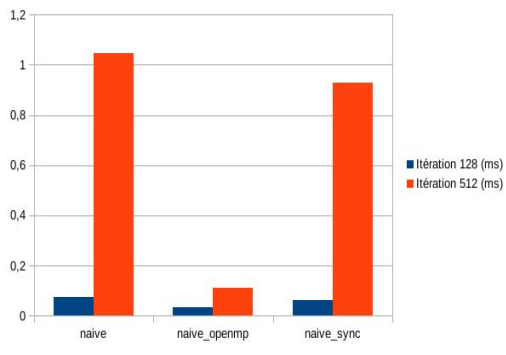
Nous avons ensuite modifié cette fonction pour lui faire gérer des itérations synchrones, c'est le but de la fonction *naive_sync*. Ce programme calcule, dans un tableau auxilliaire, les variations de quantité de grains de chaque case, et modifie seulement à la fin de l'itération le tableau de départ en lui appliquant ces variations.

Ces fonctions naïves ne sont pas immédiatement parallélisables de façon efficace. La difficulté vient du fait que deux cases différentes, se touchant par un coin, peuvent s'ébouler dans une même case. Si ces deux éboulement se produisent en même temps, il y aura un conflit, ce qui force à synchroniser les accès mémoires, ralentissant le programme.

Notre première idée était de confier à chaque thread une zone précise du tableau, ce qui fait qu'ensuite nous n'aurions plus qu'à synchorniser les bords de ces zone. Nous avons cependant trouvé une astuce permettant de contourner complètement le problème de synchronisation, implémentée dans la fonction *naive_openmp*. L'idée est de séparer le calcul de chaque itération en trois parties. On commencer par faire s'ébouler une ligne sur 3 de façon parallèle (avec une simple directive openmp), comme un éboulement n'affecte que les lignes directement au dessus et au dessus, les threads ne se gênent pas. Puis on recommence en se décalant d'une ligne, puis de 2, ce qui permet de traiter toute la table.

Voici le comparatif des temps de calculs de ces trois versions. Le premier histogramme montre le temps total, en secondes, nécessaire pour traiter le tas homogène de dimension 512, le second s'intéresse au temps par itération, pour des tas de taille 128 et 512.



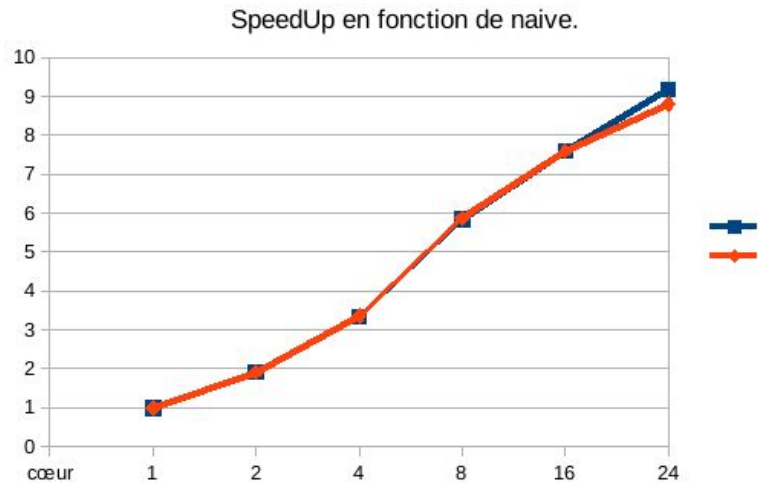


On observe un phénomène intéressant. La fonction *naive_sync* est, par itération, un peu plus rapide que la fonction *naive*, ce qui semble contre-intuitif, cette fonction *naive_sync*

utilisant les mêmes opérations, mais avec un tableau auxiliaire en plus. En fait, il semble que ces opérations supplémentaires soient absorbées en partie par le pipeline. De plus, le tableau restant plus régulier lors des itérations synchrones, la prédiction de branche est plus efficace.

Par contre, la version *naive* utilise deux fois moins d'itérations que l'autre pour traiter tout le tas, et est donc deux fois plus rapide. C'est logique, comme l'éboulement d'une case prend en compte les cases déjà éboulées dans l'itération en cours, les éboulements se propagent plus rapidement.

Intéressons nous maintenant au speed-up obtenu par la version *naive_openmp*, en fonction du nombre de threads utilisées.



Comme on peut le voir on arrive à un speed-up de 9. Ce n'est pas notre meilleure accélération. En effet, le découpage de chaque itération en trois, fait que chaque section parallèle ne traite qu'une petite partie du tas, et que la répartition des cases entre les threads n'est pas très bonne. Nous reviendrons en détail sur une courbe de speed-up openmp dans la section suivante.

3. Les fonctions *absorb*.

3.1 Principe de base

Une autre façon de régler le problème de la synchronisation est de renverser la façon dont on traite le problème. Au lieu de considérer qu'une case s'éboule sur ces voisins, on imagine que chaque case absorbe les grains excédentaires de ses voisines. Cela permet de calculer d'un coup l'évolution d'une case lors d'une itération. Toutes nos fonctions comportant *absorb* dans leur nom fonctionnent sur ce principe.

Concrètement, l'idée est de créer deux grilles, une grille source (appelée *src* dans notre code), et une grille destination (appelée *dst*), et de jongler entre ces deux grilles durant les calculs. À chaque itération :

- pour chaque case de *src*,
    - On calcule le nombre de grains de cette case modulo 4 (cela correspond à ce qu'il reste dans cette case après éboulement).
    - On calcule le quotient par 4 du nombre de grains des 4 cases adjacentes (cela correspond au nombre de grain de cette case voisine allant s'ébouler sur notre case).

  On fait ensuite la somme des ces 5 valeurs, qui correspond au nombre de grains de sables dans la case à l'itération suivante.
- On affecte cette valeur à la case correspondante du le tableau *dst*.
- On continue l'éboulement en échangeant seulement les deux tableau : *src* devient *dst* et vice-versa.
- On renvoie le bon tableau en fonction de la parité du nombre d'itérations totales effectuées.

On a donc calculé l'évolution de tous les grain de sables simultanément. Attention, l'étape élémentaire d'absorption vers une case, ne modifie pas les valeurs des voisines. Ainsi, les grains s'éboulant de la case centrales sont perdus, et ceux s'éboulant depuis les voisines sont dupliqués. Pour obtenir un résultat correct, il donc faut obligatoirement traiter à la fois toutes les cases du tableau. Ce principe d'absorption est donc fondamentalement synchrone.

Le programme *absorb* est une implémentation séquentielle directe de ce principe. Comparons-la aux versions naïves. Sur un tableau homogène de dimension 512 :

- Par itérations : *naive_sync* = 0.929 ms, *absorb* = 0.568 ms
- Au total : *naive_sync* = 86 s, *absorb* = 52s.

Pour un speed-up de 1.6 environ.

Ce speed-up provient de la diminution du nombre d'écritures en mémoire. En effet, examinons la double boucle de *naive_sync* :

```
for (int i = 1 ; i < DIM - 1 ; i++) {
   for (int j = 1 ; j < DIM - 1 ; j++) {
      if (table[i][j] >= 4) {
         finished = false;
         int mod4 = table[i][j] % 4;
         int div4 = table[i][j] / 4;
         temp[i][j] -= table[i][j] - mod4;
         temp[i-1][j] += div4;
         temp[i+1][j] += div4;
         temp[i][j-1] += div4;
         temp[i][j+1] += div4;
      }
   }
```

Lecture + écriture sur temp.

Et celle de *absorb* :

```
for (int k = 0 ; k < iterations; k++) {
    for (int i = 1 ; i < DIM - 1 ; i++) {
        for (int j = 1 ; j < DIM - 1 ; j++) {
            int left = src[i][j-1] / 4;
            int middle = src[i][j] % 4;
            int right = src[i][j+1] / 4;
            int up = src[i-1][j] / 4;
            int down = src[i+1][j] /4;
            dst[i][j] = left + middle + right + up + down;
```

Lecture seule

Ainsi, pour chaque case du tableau, *naive* fait 5 lectures et 5 écriture, mais *absorb*, en s'assurant que chaque case n'est écrite qu'une fois, ne fait que 5 lectures et 1 écriture.

D'où cette accélération massive, rendant cette version synchone presque aussi rapide que la version asynchrone, alors qu'elle calcule presque deux fois plus d'itérations.

À noter que nous utilisons une méthode différente pour détecter la stabilisation du tas. À chaque appel de notre fonction de calcul, nous recopions dans un tableau *init* la configuration initiale du tas. Puis, une fois les *iterations* étapes effectuées, nous comparons la configuration initiale à la configurations finale. Cette méthode évite complètement les tests, ce qui homogénéise le temps de calcul lié à chaque case, améliorant la parallélisation. Par contre, cela force à appeler le programme avec une valeur de *iterations* pertinente (donnée avec l'option -i) : trop petite, et les détections fréquentes ralentissent l'éxécution ; trop élevée et la stabilisation est détectée bien après la fin du calcul.

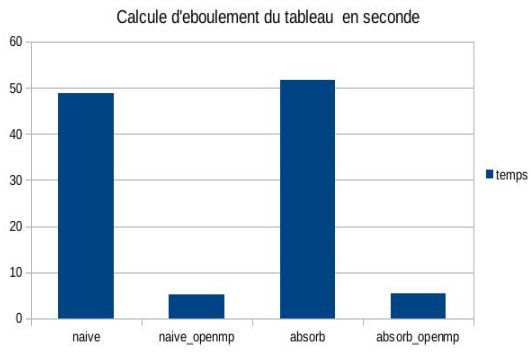
3.2 *Absorb* et OpenMP

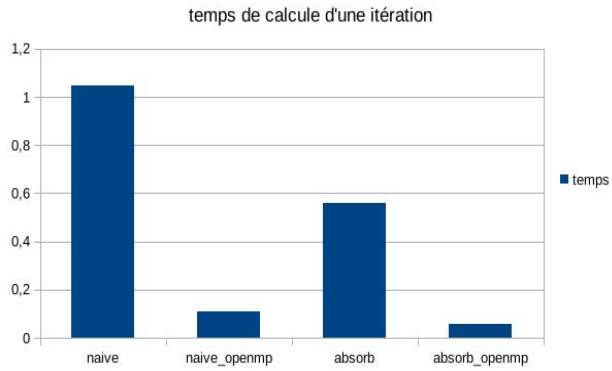
Ce principe d'absorption est de plus très agréable à paralléliser : comme on n'écrit jamais deux fois sur la même case, et que les valeurs écrites sont stockées sur un tableau auxiliaire, il n'y a pas du tout besoin de synchroniser les accès mémoires! La parallélisation ne nécessite donc qu'un simple pragma openmp.

```
#pragma omp parallel for
for (int i = 1 ; i < DIM - 1 ; i++) {
        for (int j = 1 ; j < DIM - 1 ; j++) {
            int left = src[i][j-1] / 4;
            int middle = src[i][j] % 4;
            int right = src[i][j+1] / 4;
            int up = src[i-1][j] / 4;
            int down = src[i+1][j] /4;
            dst[i][j] = left + middle + right +
up + down;
        }
    }
```

Comme les lignes du tableau sont continues dans le cache, augmenter la granularité avec un *omp for collapse(2)*, conduit à un nombre de défauts de cache excessif, et à des performances diminuées).

Pour comparaison :

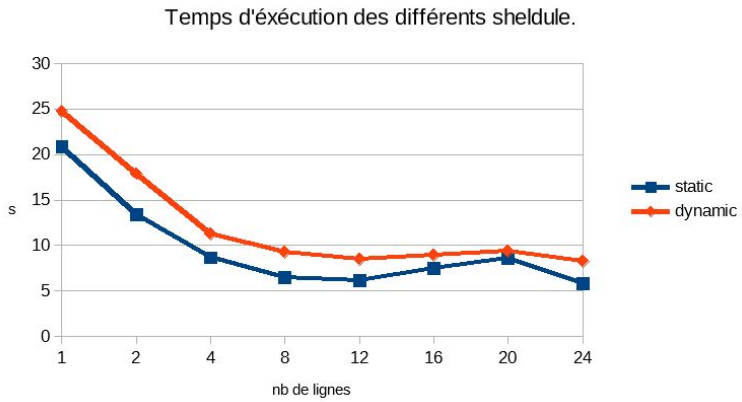




On a un speed-up équivalent entre *naive* et *naive_openmp,* et entre *absorb* et *absorb_openmp*. La courbe de speed-up de *absorb_openmp* est d'ailleurs similaire à celle de *naive_openmp*.

Ce programme *absorb_openmp* étant notre programme le plus efficace, étudions sa parallélisation plus en détail. Pour cela, nous avons crée une variante *runtime_naive_openmp*, identique à cette fonction à ceci près que les boucles for open mp sont mises en *schedule(runtime)*. Les performances de cette nouvelle version étant abominable avec le *schedule* par défaut, nous avons préféré la mettre à part.

Nous nous sommes servi de cette version pour tracer le graphe suivant, montrans les performances de *schedule(static, k)* et de *schedule(dynamic, k)* pour des tailles de blocs *k* variables.



Comme chaque ligne du tas se traite en un temps équivalent, les versions statiques sont naturellement les meilleures. Notons de plus que la meilleure version statique, aussi que la meilleure version statique est plus lente d'une seconde par rapport à la fonction

absorb_openmp. C'est parce que les boucles de cette version ont été optimisées aggressivement à la compilation.

La courbe de la version statique est irrégulière, pour des raisons de cache (une ligne de cache contient plusieurs lignes de la grille, utiliser des tailles de bloc petites force ces lignes de cache à être chargées inutilement par plusieurs threads. On observe également des problèmes de divisibilité. Dans cet exemple, 24 threads essayent de se partager 510 lignes de la matrice. Comme 510/24 = 21,25, ce partage sera plus équitable lorsque la taille des blocs est légèrement supérieure à un "diviseur" de 21,25. D'où les optima aux voisinages de 11 et 22, valeurs vraisemblablement choisies par *absorb_openmp* à la compilation.

Ces problèmes de divisibilités, de caches (la ligne initiale non traitée empêche les blocs statiques d'être parfaitement alignés avec les lignes du cache) ainsi que les problèmes de variabilité inhérentes aux politiques de *scheduling* statiques, font qu'il est difficile de prévoir à l'avance le comportement de telle où telle variante. Certains choix de dimension et de tailles de blocs, que nous avons malheureusement perdus, permettent ainsi d'obtenir des accélérations proches de 11, contre 8,5 pour l'exemple ci-dessus de la dimension 512.

Une constante tout de même : les accélérations sont bonnes, proche d'être optimales, pour des nombre de threads inférieurs à 12 (en dimension 512, travailler avec 10 threads donne une speed-up de 8), mais ensuite cette accélération stagne, voire chute. Travailler avec 12 threads semble, globalement, être le plus rapide, avec une chute marquée lorsque l'on passe de 12 à 13 en mode statique (environ 75% plus lent).

Nos machines comportant 24 cœurs logiques, séparés en 2 nœuds NUMA, nous avons d'abord supposé que cette diminution de l'accélération venait de cette architecture. Pour tester cela, nous avons crée le programme *numa_runtime_absorb_openmp*, qui initialise les tableaux dans les nœuds NUMA qui les traiteront ensuite, mais cela n'a absolument rien changé.

Nous en sommes arrivé à l'hypothèse suivante. Les machines comportent 24 cœurs logiques, mais seulement 12 cœurs physiques. Les threads de nos programmes, très gourmantes en opérations arithmétiques et accès mémoire, saturent sans doute les ALU et unités de traitement mémoire des pipelines, ce qui empêche le multi-threading d'être efficace. Au delà de 12 threads, les threads affectées sur le même coeur se marchent sur les pieds, et les cœurs sur lesquels tournent 2 threads finissent après les autres si la politique de *scheduling* est statique.

4. Les versions OpenCL

4.1 Programme *gpu*

Comme le programme *absorb_openmp* s'approchait des limites de ce qui est faisable sur le CPU, nous avons ensuite décidé de nous concentrer sur la programmation GPU.

Nous avons donc crée le programme *gpu*, qui est une réécriture du programme *absorb*, chaque thread s'occupant de calculer la nouvelle quantité de sable dans une case. L'astuce est, quand on travaille sur un tas de dimension *DIM*, de lancer *(DIM-2)*(DIM-2)* threads, pour ne pas être gêné par les cases du bord (qui forceraient, sans cela, à rajouter des tests couteux dans le kernel). Par conséquence, les options *l* et *h* données au programme doivent impérativement diviser *DIM-2*. De plus, les appels mémoires de la plupart des workgroups seront mal alignés en mémoire.

Au lieu d'alterner, comme *absorb*, entre deux tableaux pour faire les calculs, gpu utilise deux buffers alloués directement sur la mémoire de la carte graphique. Au bout de *iterations* étapes, un de ces buffers est envoyé au CPU pour l'affichage, une opération coûteuse. Il est donc essentiel quand on se sert de ce programme de fixer une valeur importante de *iterations* (par défaut, 1000).

Sur un tas homogène de taille 512, *gpu* met 0.088 ms par itérations. C'est 4,1 fois plus rapide que notre meilleure version synchrone, mais 1,76 fois plus lent que *absorb_openmp*. Ce résultat est obtenu en confiant une ligne à chaque workgroup, mais dépend peu de la taille des workgroups.

4.2 Programme gpu_overlap

Déçu par ces performances, nous avons essayé de créer une version plus optimisée. Notre hypothèse pour expliquer cette lenteur est la grande quantité d'appels mémoires entre la mémoire centrale de la carte graphique et chaque workgroup. Au total, 5 lectures par cases et par itération synchrone! Pour limiter cela, nous avons décidé d'essayer de nous servir de la mémoire locale de chaque workgroup, en implémentant des itérations k-synchrones : on commencer par recopier dans un tableau local une partie du tas de sable, puis on calcule, en interne, un certain nombre d'itérations locales, avant de tout renvoyer à la mémoire centrale.

Nos avons d'abord essayé de modifier directement *gpu*, sans succès. En effet, une fois que chaque thread a recopié en local le contenu de la case correspondante, il faut en plus recopier le contenu d'un certain nombre de cases frontières, dont les indices ne correspondent pas de façon naturelle à ceux des threads. Si vous voulez rire un peu, un tel essai de kernel infructueux est conservé dans le projet sous le nom *gpu_square_kernel.cl*.

Nous avons donc choisi, avec ce programme *gpu_overlap* de changer d'approche, en nous arrangeant pour que chaque case de frontière soit traitée par sa propre thread. Cela demande de lancer plus de threads qu'il n'y a de cases réelles dans le tas de sable. Par exemple, pour des itérations 2-synchrones, et des workgroups calculant des zones de taille 8*8, il faut affecter (8+4)*(8+4) = 144 threads par workgroup. Cela nécessite de faire un petit peu de gymnastique de décalages sur les indices, mais une fois cela fait le code du kernel est remarquablement simple.

Attention : les options b, l et h données au programme doivent vérifier des contraintes fortes pour que le programme se lance. En dimension *DIM*, l et h doivent diviser DIM-2*b. Les valeurs par défaut du programme, 1, 2 et 2, sont prudentes, mais très lentes en pratique.

- En dimension 514, avec des itérations 1-synchrones, notre programme est le plus rapide pour l=8 et h=16, et met 0.191ms par itération.
- En dimension 516, avec des itérations 2-synchrones, notre programme est le plus rapide pour l=8 et h=32, et met 0.165ms par itération.
- En dimension 518, avec des itérations 3-synchrones, notre programme est le plus rapide pour l=8 et h=16, et met 0.176ms par itération.
- En dimension 520, avec des itérations 4-synchrones, notre programme est le plus rapide pour l=8 et h=32, et met 0.203ms par itération.

Ces résultats sont, dans le meilleur cas, environ deux fois plus lent que ceux de *gpu*. Cela à notre avis pour deux raisons.

- Notre hypothèse sur la lenteur des accès mémoire de la fonction *gpu* n'était pas correcte. Les 5 lectures par threads de *gpu* sont compensées par les écritures et lectures locales, barrières et autres surcoûts introduits par *gpu_overload*.
- Ce type d'algorithme est fondamentalement inadapté aux cartes graphiques. En effet, ces cartes fonctionnent mieux avec des workgroups de petites taille. Cela empêche de traiter simultanément un grand nombre d'itérations, et fait que les threads supplémentaires ajoutées pour traiter les frontières créent un surcoût important. Par exemple, en dimension 516, notre version 2-synchones optimale crée 70% plus de threads que le programme *gpu* (et est 87% plus lente).

Ces explications ne sont pas complètement satisfaisantes. Notre compréhension limitée du hardware de la carte graphique nous empêche d'analyser plus finement le programme. En particulier, contrairement à notre intuition et à ce qui se passe avec *gpu*, la fonction *gpu_overload* est plus rapides quand h est grand et l petit. Par exemple, en dimension 512 et en 1-synchrone, l=128 et h=4 est presque deux fois plus lent que l=4 et h=128.

5. Versions asynchrones

Ces programmes GPU nous ayant pris beaucoup de temps, nous n'avons pas pu étudier de façon satisfaisante le problème de l'éboulement asynchrone. Signalons quand même une piste qui nous semble intéressante, illustrée par le programme *task_seq*. Ce programme commence par s'initialiser en stockant dans une pile (appelée *stack*) la liste de toutes les cases devant s'ébouler.

Ensuite, il parcourt cette pile, en testant, pour chaque case, si l'éboulement de cette case va déclencher un éboulement sur ces voisins. Si c'est le cas, et que le voisin en question n'est pas dans la pile (on le vérifie à l'aide d'un tableau auxiliaire appelé *scheduled*), on la rajoute dans la pile.

Ainsi, à chaque étape, le programme traite une case en éboulement, ou proche d'une case en éboulement, ce qui réduit drastiquement le nombre de traitement inutiles. En contrepartie, les nombreux tests supplémentaires font que le traitement de chaque case est bien plus lent que dans les programmes précédents.

Cette approche est catastrophique sur le tableau homogène, où les éboulements interviennent partout dans le tas, mais est excellente sur des tas plus localisés.

Ainsi, en dimension 512, sur la tour de taille 10^5, ce programme est 7,9 fois plus rapide que *naive*, et 16,5 fois plus rapide que *absorb*! Il rivalise donc avec nos meilleures versions parallèles. Même avec une tour bien plus grande, de taille 10^6, il reste 75% plus rapide que *absorb* (mais est 30% plus lent que *naive*).

L'étape suivante, que nous n'avons pas eu le temps de traiter, est de réfléchir à comment paralléliser ce type d'approche. Une idée est de crée divers piles pour différentes zones disjointes du tas de sable, et d'appeler périodiquement une fonction éboulant la frontière entre ces zones. Le risque étant, bien sûr, que les grains s'accumulent dans cette frontière.

Une autre approche serait une version hybride, où chaque tâche de la pile correspond par exemple à une zone carrée du tas de sable, qui serait traitée par une autre de nos fonctions, par l'intermédiaire par exemple d'une tâche openmp. Si chaque zone est suffisament grande, on pourrait, entre chaque étape, créer une nouvelle pile, dont les zones sont décalées par rapport à l'étape précédente, ce qui éliminerait peut-être les problèmes de frontière.