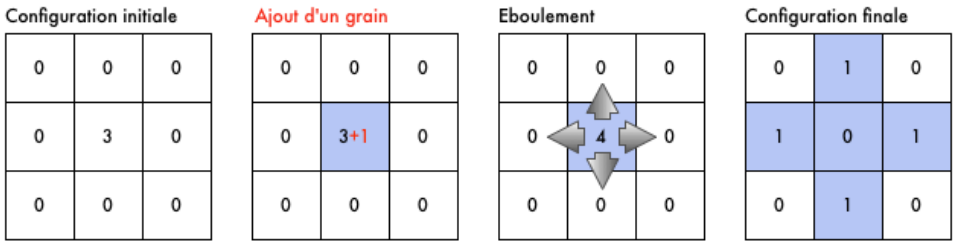
Projet de Programmation Multicoeur et GPU
Travail en binôme à rendre avant le 29 avril minuit

Tas de sable abéliens

On cherche à modéliser la dynamique d'éboulement des grains de sable organisé en tas sur une table. Pour cela on discrétise l'espace de la table au moyen d'un tableau 2D et on impose les règles suivantes :

- Tout grain appartient à exactement une case du tableau ;
- Toute case du bord peut accepter un nombre quelconque de grains (cela simule la chute des grains de la table) ;
- Toute case interne contenant plus de 4 grains s'éboule en cédant 1 grain à chacune de ces voisines.



Voici un exemple de programme calculant une séquence d'éboulements jusqu'à stabilisation :

```
int table[DIM+1][DIM+1];

void initialiser()
{
  ……
}


void
afficher()
{
  …
}

int main()
{
initialiser();

 int i=0;
do
  {
   printf("**** %d ****\n",i++);
   afficher();
  } while(traiter(1,1,DIM,DIM));

return 0;
}
```

```
int traiter(int i_d, int j_d, int i_f, int j_f)
{
 int i,j;
 int changement = 0;
 for (i=i_d; i < i_f; i++)
    for (j=j_d; j < j_f; j++)
            if (table[i][j] >= 4)
              {
                int mod4 = table[i][j] % 4;
                int div4 = table[i][j] / 4;
                table[i][j] = mod4;
                table[i-1][j] += div4;
                table[i+1][j] += div4;
                table[i][j-1] += div4;
                table[i][j+1] += div4;
                changement = 1;
              }
 return changement;
}
```

```
Trace d'exécution :
********* 0 *********
0 0 0 0 0
0 0 7 0 0
0 7 7 7 0
0 0 7 0 0
0 0 0 0 0
********* 1 *********
0 0 1 0 0
0 2 5 3 0
1 5 5 2 2
0 3 2 0 1
0 0 2 1 0
********* 2 *********
0 0 2 1 0
0 4 3 1 1
2 3 5 0 3
1 1 0 2 1
0 1 3 1 0
********* 3 *********
0 1 3 1 0
1 2 1 2 1
3 1 3 1 3
1 2 1 2 1
0 1 3 1 0
```

Dans cette version on a accéléré le calcul en utilisant la division euclidienne.

Ce modèle, dit du « *tas de sable abélien* », a été introduit en physique par Bak, Tang et Wiesenfeld comme un modèle type du phénomène d'auto-organisation critique (self-organized criticality). Ce modèle présente de belles propriétés algébriques qui ont inspirées les *combinatoriciens*, des chercheurs mi-mathématiciens mi-informaticiens qui s'intéressent ainsi à la physique statistique. Par exemple, on peut montrer qu'après un nombre fini d'éboulements, la situation se stabilise, et le tas de sable admet une forme limite qui ne dépend pas de l'ordre d'éboulement des cases. On trouvera sous http://www.espace-turing.fr/Tas-de-sable-et-criticalite-auto.html une introduction au problème tas de sable ainsi qu'un simulateur.

Simulation synchrone

Étant donnée une configuration, on appelle *itération synchrone* l'éboulement simultané de toutes les cases. En itérant suffisamment ce procédé de calcul on est certain d'arriver à la forme limite où plus aucun éboulement n'est possible – et puis surtout partant d'une configuration symétrique on atteint une autre configuration symétrique agréable à observer.

Notre premier objectif sera de calculer le plus rapidement possible la configuration obtenue après un nombre k d'itérations synchrones. Pour cela on mettra en œuvre deux techniques de parallélisation :

Parallélisation d'une itération synchrone. Les threads se partagent le travail en découpant la table en différents domaines (sous forme de bandes ou de tuiles) et se synchronisent après chaque itération synchrone.

Parallélisation de p itérations synchrones. Il s'agit ici d'accélérer le calcul en réduisant la partie séquentielle du programme, en réduisant par exemple le nombre de barrières nécessaires à la synchronisation du travail des threads. Les threads se donnant rendez-vous toutes les p étapes, on pourra même autoriser une certaine désynchronisation. En effet, lors d'une itération synchrone le prochain état d'une case est déterminé par son état et l'état de ses voisins, soit les cases situées dans une boule de rayon 1 autour de la case en question. On remarque aussi que l'on peut déterminer l'état d'une case après deux itérations synchrones si l'on connaît l'état de toutes les cases autour de la case en question dans une boule de rayon 2 (c'est à dire l'état de la case, l'état de ses voisines et l'état des voisines des voisines). De façon générale, l'état d'une case après p itérations synchrones est déterminé par l'état des cases appartenant à la boule de rayon p centrée sur la case en question. Ainsi pour calculer l'état des cases d'un domaine après p itérations synchrones, il suffit de connaître l'état des cases du domaine ainsi que celui de toutes les cases situées à distance inférieure ou égale à p du domaine considéré.

L'idée est de permettre aux threads de calculer dans leur coin, sans synchronisation, durant p itérations synchrones. Pour cela chaque thread doit connaître initialement l'état des cases à distance p de son domaine, pour le calcul de la première itération, à $p-1$ pour la seconde et de façon générale à distance $p-k$ pour la $k^{ième}$ itération. Il s'agit donc d'effectuer du calcul en plus : l'état des cases au bord du domaine sont *indépendamment* calculés plusieurs fois. Il s'agit d'échanger du temps de synchronisation contre du temps et de l'espace de calcul.

Simulation asynchrone

Il s'agit de calculer la configuration limite le plus vite possible peu importe la manière.

Cas à étudier

On considère des tableaux 128 x 128 (bords inclus, c'est à dire 126 x 126 cases utiles) et 512 x 512. Deux configurations à étudier :

- Configuration homogène où chaque case comporte initialement 5 grains,
- Configuration où une case centrale comporte 10^5 éléments, les autres étant vides.

On mesurera le temps mis par le programme pour atteindre la configuration limite. Pour ces mesures on désactivera tout affichage.

Travail demandé

1. Simulation synchrone
    a. Programme séquentiel optimisé : il s'agit de présenter différentes versions séquentielles et de mesurer leur performance sur le cas 128 x 128 (homogène + case centrale). On pourra optimiser le programme tant au niveau code qu'au niveau algorithmique (repérer les zones stabilisées par exemple).
    b. Programme parallèle où les threads se synchronisent à chaque itération synchrone. On cherchera à déterminer le nombre optimal de threads pour les quatre cas à étudier en présentant des courbes de *speed-up* (accélération obtenue en fonction du nombre de threads employés).
    c. Programme parallèle où les threads se synchronisent toutes les p itérations synchrones. L'objectif est de réduire le nombre d'appel aux barrières. On cherchera à déterminer expérimentalement la meilleure combinaison (p, nombre de threads).
    d. Programme GPU en OpenCL. On optimisera les transferts mémoires en évitant de rapatrier autant que possible les données sur le CPU.

2. Simulation asynchrone : carte blanche.

Interface graphique

Une interface graphique est mise à votre disposition. Utilisez la pour faire vos tests, mettre en valeur votre travail. Cette interface permet de faire varier la fréquence de rafraichissement et la possibilité d'afficher les configurations périodiquement (une sur dix par exemple). Vous pouvez enrichir cette interface pour ajouter des fonctionnalités pour par exemple faire varier le nombre de threads utilisés ou encore de basculer interactivement d'une technique de calcul à une autre afin de bien mettre en valeur votre travail d'optimisation.

Rendu

Rapport et codes sources à transmettre à déposer dans le répertoire ~pwacreni/RENDU_PMG avant le 29 avril minuit. Les soutenances seront organisées la semaine des examens. Les étudiants sont encouragés à transmettre aux enseignants une pré-version de leur projet pour commentaires. Les soutenances auront lieu lors de la semaine des examens.