Tas de sable abéliens

Résumé

Nous cherchons à optmiser la simulation numérique de l'écoulement d'un tas de sable. La modélisation est constituée d'un tableau 2D de cellules dont le comportement est régi par la simple règle suivante : *une cellule contenant plus de 4 grains s'éboule en cédant 1 grain à chacune de ces voisines*. Ce modèle, dit du *tas de sable abélien* a été introduit en physique en 1987 par Bak, Tang et Wiesenfeld comme un modèle type du phénomène d'auto-organisation critique car il présente une belle propriété : après un nombre fini d'éboulements la situation se stabilise dans une configuration unique, et donc indépendante de l'ordre d'éboulement. On trouvera sous http ://www.espace-turing.fr/Tas-de-sable-et-criticalite-auto.html une introduction à ce modèle.

1 Cadre du mini-projet

Ce mini-projet est à faire en binôme et à rendre pour le mardi 20 mars 23h59. Votre rapport, sous forme de fichier au format PDF, contiendra les parties principales de votre code, la justification des optimisations réalisées, les conditions expérimentales et les graphiques obtenus accompagnés chacun d'un commentaire le décrivant et l'analysant.
Les machines utilisées doivent être des machines du CREMI dotées d'au moins 12 cœurs physiques. Les expériences menées devront être peu bruitées : il est inutile de faire des expériences sur une machine chargée. On pourra comparer différentes configurations matérielles.
Vous placerez ce rapport `/net/cremi/pwacreni/PROJET-PAP` le nom de votre rapport sera de la forme : `binome1-binome2-clef-secrete.pdf`
Less sources de l'application sont sous `~pwacreni/PAP-2018/projet-tas .`

2 Description du modèle

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.

3 Objectifs du projet

Le but de ce projet est de calculer le plus vite possible la configuration limite peu importe l'ordre des calculs. Dans un premier temps on produira des programmes séquentiels efficaces : on pourra optimiser le code mais aussi l'algorithme. On pourra par exemple éviter de calculer les zones temporairement stabilisées ou encore faire temporairement plus de calcul sur une zone de calcul que sur une autre : le calcul est alors asynchrone, la notion d'itération n'a plus globalement de sens. Dans un second temps on cherchera à paralléliser le calcul en faisant bien attention à son exactitude : tout algorithme erroné sera lourdement sanctionné. Pour cela on traitera méticuleusement les cellules partagées par deux threads en essayant de limiter au strict minimum l'utilisation de mutex et de variables atomiques.

4 Travail d'expérimentation et d'analyse

Votre rapport détaillera une à une l'intérêt des optimisations apportées aux versions successives des programmes séquentiels et parallèles. De plus on comparera la technique de parallélisation choisie à d'autres. Enfin, comme lors du premier mini-projet, il s'agira de produire et d'analyser des graphiques montrant l'accélération obtenues en fonction du nombre de threads utilisés par rapport à la version séquentielle optimisée. Pour toutes les expériences on utilisera de préférence les deux configurations suivantes :

`./prog -k sable -s 256 -p 4partout -n` : chaque case comporte initialement 4 grains ;

`./prog -k sable -s 4096 -p alea -n` : configuration pseudo-aléatoire où 1024 cases comportent au plus 4096 éléments, les autres étant vides.

On pourra augmenter le nombre de grains si, à force d'optimisations, les expériences deviennent peu significatives.

5 Quelques indications

5.1 Éléments d'analyse

Il s'agit de raisonner à partir de la description des courbes et de formuler des hypothèses pour expliquer les comportements observés et, idéalement, de valider / infirmer les hypothèses à l'aide de nouvelles expériences. Voici quelques exemples d'observations et de pistes d'interprétations :

— Un speedup supérieur au nombre de cœurs est constaté. Avez-vous utilisé le meilleur algorithme séquentiel ? Gagne-t-on grâce au non-déterminisme du parallélisme ? Bénéficie-t-on d'effets de cache favorables ou d'un meilleur débit mémoire ?

— On observe qu'une distribution dynamique est beaucoup moins performante qu'une distribution statique. Est-ce un problème de contention sur un mutex ? Est-ce un problème de localité mémoire (cache / NUMA) ?

— On constate qu'une distribution statique est beaucoup moins performante qu'une distribution dynamique. La charge de travail est-elle bien équilibrée ? la mémoire est-elle bien répartie sur la machine ?

— On observe que la courbe d'accélération croit puis décroit. Y'a-t-il suffisamment de travail pour compenser le surcoût de la parallélisation ? Y-a-t-il un déséquilibre dû à l'hyperthreading ? Y a-t-il de la contention (synchronisation / cache / mémoire) ?

5.2 À propos du code

Dans le code fourni, certaines fonctions sont appelées de façon générique ; lors d'un appel `./prog -k kernel -v version -p config` le programme cherchera à d'abord à utiliser la fonction `kernel_fun_version()` ou, à défaut, `kernel_fun()`. Voici les fonctions concernées :

kernel_init_version() pour allouer les structures de données ;

kernel_ft_version() pour appliquer une stratégie de placement de type *first touch* ;

kernel_draw_config() pour appliquer une configuration initiale ;

kernel_compute_version() pour calculer un certain nombre d'itérations ;

kernel_refresh_img_version() pour créer une image à partir d'une configuration ;

kernel_finalize_version() pour libérer les données allouées dynamiquement.

5.3 À propos des options de compilation

Pour activer des options de compilation sur un ensemble précis de fonctions :

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-all-loops")
void foo()
{
...
}
void bar()
{
...
}
#pragma GCC pop_options
```

5.4 Présentation des codes dans un rapport

On pourra utiliser le paquet latex `listing` pour formater des codes dans votre rapport; voici des paramètres

```
\lstset{
 language=C,
 extendedchars=true,
 basicstyle=\tt, %\tt\small,
 %numbers=left, numberstyle=\tiny, stepnumber=2, numbersep=5pt,
 lineskip=-2pt
}
```

5.5 Exemple de code

Voici un exemple illustratif de programme calculant une séquence d'éboulements jusqu'à stabilisation. Dans cette version on a accéléré le calcul en utilisant la division euclidienne et on a simplifié le code en ne calculant pas les cases du bord.

```
int table[DIM][DIM];
...

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;
}

...
i=0;
do
    {
      printf("**** %d ****\n",i++);
      afficher_table();
    } while(traiter(1,1,DIM-1,DIM-1));
```

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
```

L'étudiant attentif observera qu'à chaque éboulement les modifications portent sur trois lignes et trois colonnes. La parallélisation via la seule directive `parallel for` ménerait donc à un calcul potentiellllement erroné : deux threads travaillant simultanément sur des colonnes ou des lignes voisines peuvent être en conflit pour (lire / écrire) ou (écrire / écrire) la même case mémoire. Ce code ne se parallélise donc pas trivialement.
Pour éviter ces conflits il faut soit utiliser plus de mémoire (un second tableau de taille identique, par exemple), ou bien utiliser des primitives de synchronisation (mutex + petit tableau, opérations atomiques). Une autre technique bien adaptée ici est de partitionner habillement l'ensemble des cases et d'enchainer séquentiellement le traitement en parallèle de chaque partie en distribuant ses cases aux threads de façon à ce qu'aucun thread ne soit en conflit avec un autre. Voici deux exemples de partitionnement - distribution, saurez-vous trouver une meilleure configuration ?

1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9

Toutes les cases numérotées identiquement peuvent être traitées en parallèle.

1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3
1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3

Toutes les lignes numérotées identiquement peuvent être traitées en parallèle à condition que toute ligne soit attribué à un seul thread.