pap-rapport-1

Odrian Tarrieu, Abdoulaye Soumah

February 2024

1 Introduction

Dans ce premier rapport, nous intéressons à l'optimisation des méthodes `asandPile_do_tile_opt()` et `ssandPile_do_tile_opt()`

2 (4.1) ILP optimization

Dans un premier temps, nous remarquons que l'ajout du mot clé restrict sur la variable static `TABLE` nous permet grandemant améliorer la vitesse d'éxécution du code.

```
static TYPE *restrict TABLE = NULL;
```

En effet, nous obtenons les temps suivant avec la machine MODIGLIANI (salle 008) :

fonctions	temps sans restrict	temps avec restrict
asand default	38896 ms	23946 ms
ssand default	182802 ms	43978 ms

On nous fait remarquer dans un second temps, que les fonctions `asandPile_do_tile_opt()` et `ssandPile_do_tile_opt()` possèdent des répétition d'appel à une même méthod qui peuvent être optimisées.

Ainsi, nous obtenons les codes suivants:

```
int asandPile_do_tile_opt(int x, int y, int width, int height)
{
  int change = 0;
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++){
      if (atable(i, j) >= 4)
      {
        TYPE tmp = atable(i, j);
        TYPE tmp_div = tmp/4;
        atable(i, j - 1) = atable(i, j - 1) + tmp_div;
        atable(i, j + 1) = atable(i, j + 1) + tmp_div;
        atable(i - 1, j) = atable(i - 1, j) + tmp_div;
        atable(i + 1, j) = atable(i + 1, j) +  tmp_div;
        atable(i, j) = tmp % 4;
```

```
        change = 1;
      }
    }
  return change;
}
```

```
    int ssandPile_do_tile_opt(int x, int y, int width, int height)
{
  int diff = 0;

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
    {
      TYPE tmp = table(in,i,j);
      TYPE tmp_mod = tmp%4;
      TYPE add1 = table(in, i + 1, j) / 4;
      TYPE add2 = table(in, i - 1, j) / 4;
      TYPE add3 = table(in, i, j+1) / 4;
      TYPE add4 = table(in, i, j-1) / 4;
      table(out,i,j) = tmp_mod + add1 + add2 + add3 + add4;
      if (table(out, i, j) >= 4)
        diff = 1;
    }

  return diff;
}
```

Afin de tester la rapidité de cette version optimisée par rapport à la version par défault nous utilisons ces commandes :

Tests des versions non-optimisées :

```
    --> ./run -k asandPile -wt default -s 512 -n
    --> ./run -k ssandPile -wt default -s 512 -n
```

Tests des versions optimisées :

```
    --> ./run -k asandPile -wt opt -s 512 -n
    --> ./run -k ssandPile -wt opt -s 512 -n
```

Ainsi, nous obtenons ces résultats en utilisant la machine MODIGLIANI (salle 008):

tests	temps asand default	temps asand opt
1	23946 ms	20954 ms
2	23835 ms	20928 ms
3	23892 ms	20874 ms

tests	temps ssand default	temps ssand opt
1	43978 ms	19511 ms
2	44044 ms	19403 ms
3	44042 ms	19549 ms

Nous pouvons ainsi remarquer que les fonctions ont bien été optimisées.

3 (4.2) OpenMP implementation of the synchronous version

Dans cette section nous nous intéressons à synchroniser les fonctions `ssandPile_compute_seq()` et `ssandPile_compute_tiled()`.

Tout d'abord, nous avons implémenté une version synchronisée de `ssandPile_compute_seq()`.

```
    unsigned ssandPile_compute_omp(unsigned nb_iter)
{
  unsigned it;
  int change;
  #pragma omp parallel for shared(change) schedule(runtime)
  for (it = 1; it <= nb_iter; it++)
  {
    if (change == 0)
      continue;
    change = do_tile(1, 1, DIM - 2, DIM - 2);
    swap_tables();
  }
  if (change == 0)
    return it;
  return 0;
}
```

Afin de tester les performances de la versions synchronisée, nous avons réaliser un graphe de vitesse d'éxécution du code en fonction du nombre de threads, et nous avons comparé ces vitesse avec celle d'une éxécution en séquentielle de la même fonction.

Donc nous obtenons le graphe suivant :

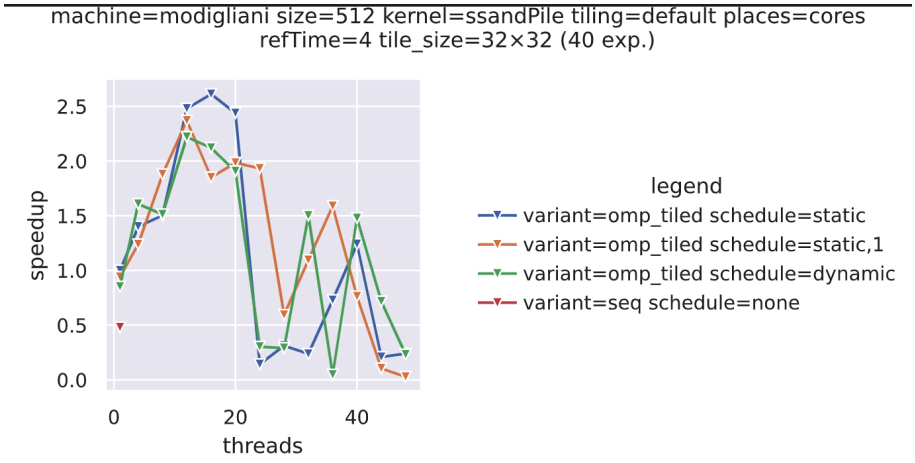

Figure 1: Speed-up graph ssandPile-omp

Nous remarquons que la vitesse maximale est atteinte avec l'utilisation de la méthode static et avec 15 threads.

Ensuite nous avons implémenté une version synchronisée de `ssandPile_compute_tiled()`.

```
    unsigned ssandPile_compute_omp_tiled(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;
    #pragma omp parallel for collapse(2) schedule(runtime) shared(
        change)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        change |=
            do_tile(x + (x == 0), y + (y == 0),
                    TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                    TILE_H - ((y + TILE_H == DIM) + (y == 0)));
    swap_tables();
    if (change == 0)
      return it;
  }

  return 0;
}
```

Afin de tester les performances de cette version synchronisée, nous avons généré une carte de chaleur qui traduit le rapport de la vitesse d'éxécution du code avec la fonction synchronisée sur la vitesse d'éxécution du code avec la même fonction séquentielle en fonction de la taille des tuilles. De plus nous avons générée 3 cartes de chaleur en fonction de différentes approche de synchronisation dynamic, static et (static,1).

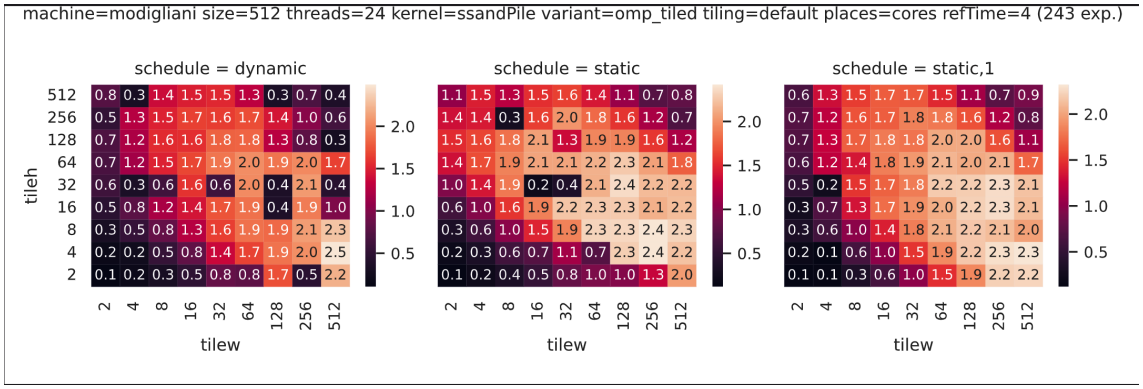

Figure 2: Heat map ssandPile-omp-tiled

Nous pouvons remarquer sur la carte de profondeur avec la méthode (static,1) propose une meilleur régularité de zone de couleur chaude, qui se traduit par de meilleurs performances sur l'ensemble des tailles de tuiles.

Sur carte de chaleur dynamic, la meilleur performance est celle avec une largeur de tuiles à 4 et un longueur de tuiles à 512.

Sur carte de chaleur static, la meilleur performance est celle avec une largeur de tuiles à 8, et un longueur de tuiles à 256.

Sur carte de chaleur (static,1), la meilleur performance est celle avec une largeur de tuiles à 4 et un longueur de tuiles à 512.