Programmation Parallèle - Tas de sable abéliens
Etape 1 (4.1 et 4.2)

Corentin Drezen, Mouhamadou Mansour Gueye

Fevrier 2024

1 IPL Optimization

Dans cette section on va mentionner ***asandPile*** a plusieurs reprise au début car la consigne ne pas le mentionner a été donné après l'essentiel de sa redaction.

Pour optimiser le calcul des tuiles on s'est d'abord interessé aux macros et fonctions inline suivantes:

```
typedef unsigned int TYPE;

static TYPE *TABLE = NULL;

static inline TYPE *atable_cell(TYPE *restrict i, int y, int x)
{
  return i + y * DIM + x;
}

#define atable(y, x) (*atable_cell(TABLE, (y), (x)))

static inline TYPE *table_cell(TYPE *restrict i, int step, int y, int x)
{
  return DIM * DIM * step + i + y * DIM + x;
}

#define table(step, y, x) (*table_cell(TABLE, (step), (y), (x)))
```

On peut voir que des pointeurs de cellule peuvent être calculés à chaque fois lorsque l'on utilise les macros ***atable*** et ***table*** et que les fonctions ***_do_tile_default*** les utilisent à de multiples reprises avec plus ou moins les mêmes arguments.

```
int asandPile_do_tile_default(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)
      {
        atable(i, j - 1) += atable(i, j) / 4;
        atable(i, j + 1) += atable(i, j) / 4;
        atable(i - 1, j) += atable(i, j) / 4;
        atable(i + 1, j) += atable(i, j) / 4;
        atable(i, j) %= 4;
        change = 1;
      }
  return change;
}
```

Ainsi dans la fonction ci dessus on a commencé par définir un pointeur vers la cellule redondante à l'aide de la fonction inline ***atable_cell***

On a aussi rajouté une variable pour évité de recalculer la même division par 4 plusieurs fois mais ça ne semblait pas améliorer le temps d'execution en premier lieu, car on avait oublié le mot clé 'const' et le compilateur effectuait sans doute déjà cette optimisation.

```
int asandPile_do_tile_opt_old(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++)
    {
      TYPE *cell = atable_cell(TABLE, i, j);

      if ( *cell >= 4)
      {
        const TYPE cell_quarter = *cell / 4;

        atable(i, j - 1) += cell_quarter;
        atable(i, j + 1) += cell_quarter;
        atable(i - 1, j) += cell_quarter;
        atable(i + 1, j) += cell_quarter;
        *cell %= 4;
        change = 1;
      }
    }
```

```
  return change;
}
```

Avec le code ci dessus (sans parallelisation) sur une machine de la salle 008 le dimanche 11/02/24 on obtient $\pm 31100ms$ et $\pm 38400ms$ avec la version par défault asynchrone en utilisant la commande "**./run -k asandPile -s 512 -n**", soit à peu près 7 secondes de différence.

Ensuite on a voulu calculer plus rapidement les autres adresses '*atable*' et on a déplacé la déclaration du pointer '*cell*' dans la boucle superieur pour l'incrementer 'manuellement' afin de ne pas recalculer totalement l'adresse à chaque sous iteration.

On a aussi pu rajouté le mot clé ***restrict*** à notre pointeur '*cell*' (et à *TABLE*) qui permet d'informer le compilateur qu'il n'y aura pas d'autres pointeur vers cette objet pendant sa durée de vie. Cela permettrais au compilateur de faire des optimisations en vectorisant.

```
int asandPile_do_tile_opt(int x, int y, int width, int height)
{
  int change = 0;

  TYPE *restrict cell = atable_cell(TABLE, y, x);

  for (int i = y; i < y + height; i++)
  {
    for (int j = x; j < x + width; j++)
    {
      if (*cell >= 4)
      {
        const TYPE cell_quarter = *cell / 4;

        *(cell - 1) += cell_quarter;
        *(cell + 1) += cell_quarter;
        *(cell - DIM) += cell_quarter;
        *(cell + DIM) += cell_quarter;
        *cell %= 4;
        change = 1;
      }
      cell++;
    }
    cell+=2;
  }
  return change;
}
```

Avec le code ci dessus dans les même conditions que précedement on a obtenu $\pm 21200ms$ soit à peu près $16s$ de moins que la version asynchrone de base.

On a utilisé le même raisonement pour l'optimisation de la version synchrone ci dessous.

```
1  int ssandPile_do_tile_opt(int x, int y, int width, int height)
2  {
3    int diff = 0;
4
5    TYPE *restrict cell_in = table_cell(TABLE, in, y, x);
6    TYPE *restrict cell_out = table_cell(TABLE, out, y, x);
7
8    for (int i = y; i < y + height; i++)
9    {
10     for (int j = x; j < x + width; j++)
11     {
12       // table(out, i, j) = table(in, i, j) % 4;
13       *cell_out = *cell_in % 4;
14       // table(out, i, j) += table(in, i, j + 1) / 4;
15       // table(out, i, j) += table(in, i, j - 1) / 4;
16       *cell_out += *(cell_in + 1) / 4;
17       *cell_out += *(cell_in - 1) / 4;
18
19       //*cell_out += table(in, i + 1, j) / 4;
20       //*cell_out += table(in, i - 1, j) / 4;
21       *cell_out += *(cell_in + DIM) / 4;
22       *cell_out += *(cell_in - DIM) / 4;
23
24       if (*cell_out >= 4)
25         diff = 1;
26
27       cell_in++;
28       cell_out++;
29     }
30     cell_in+=2;
31     cell_out+=2;
32   }
33   return diff;
34 }
```

Mais on s'est rendu compte que ces changement était moins performant pour cette version synchrone et on a réalisé que les lignes 16 et 17 était en cause, *table(in, i, j + 1) / 4* etant plus perfomant que **(cell_in + 1) / 4*

On a voulu factoriser les division successives par 4 en additionant tout puis en divisant mais ce calcul ne donnait plus toujours le même résultat car l'arrondie à l'entier le plus proche pouvait être différent.

On a donc essayé de mettre tout les calculs dans une seule assignation et

cela semble avoir permis au compilateur d'optimiser les operations car le temps d'execution a été réduit par 2 avec la fonction ci dessous.

On a aussi essayé de transformer les division par 4 en shift de 2 bits (>>2) et le modulo 4 en 'and 3' (& 3) mais cela n'a rien changé, le compilateur effectuant sans doute déjà la même chose.

```
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 *restrict cell_in = table_cell(TABLE, in, i, j);
      TYPE *restrict cell_out = table_cell(TABLE, out, i, j);

      *cell_out = table(in, i, j) % 4
                + (table(in, i, j - 1) / 4)
                + (table(in, i, j + 1) / 4)
                + (*(cell_in - DIM) / 4)  // table(in, i - 1, j) / 4
                + (*(cell_in + DIM) / 4); // table(in, i + 1, j) / 4

      if (*cell_out >= 4)
        diff = 1;
    }

  return diff;
}
```

On est revenu uniquement à des calcul à l'interieur du for imbriqué car cela semblait géner OpenMP notament si on voulait "faire un *collapse(2)*" et cela semblait identique en performance.

2 Implementation OpenMP de la version synchrone

On nous a demander de faire une version parallelisé **ssandPile_compute_omp** qui n'utilise pas les tuiles. On en a donc déduit qu'il fallait paralleliser **ssandPile_compute_seq** car c'est la nomenclature habituel des fonction en TD. De plus **ssandPile_compute_seq** n'utilise effectivement pas vraiment de tuile car elle utilise techniquement des tuiles mais elle sont de la taille totale de l'image.

Cepandant on ne pouvait pas paralleliser le for de cette fonction tel quel car chaque iteration depend du resulat de la précédante et on y modifie la lecture du tas de sable.

On a donc copié notre **ssandPile_do_tile_opt** à l'interieur.

```
unsigned ssandPile_compute_omp(unsigned nb_iter)
{
  int res = 0;

  for (unsigned it = 1; it <= nb_iter; it++)
  {
  //#pragma omp parallel master
    int change = 0;
    int x = 1; int y = 1; int width = DIM - 2; int height = DIM - 2;

    #pragma omp parallel for collapse(2) shared(change) schedule(runtime)
    for (int i = y; i < y + height; i++)
      for (int j = x; j < x + width; j++)
      {
        TYPE *restrict cell_in = table_cell(TABLE, in, i, j);
        TYPE *restrict cell_out = table_cell(TABLE, out, i, j);

        const TYPE calc = *cell_in % 4
        + (table(in, i, j - 1) / 4)
        + (table(in, i, j + 1) / 4)
        + (*(cell_in - DIM) / 4)
        + (*(cell_in + DIM) / 4);

        //#pragma omp atomic write
        *cell_out = calc;

        if (*cell_out >= 4)
          //#pragma omp atomic write
          change = 1;
      }

    //#pragma omp critical
    swap_tables();

    if (change == 0)
        res = it;
        break;
  }
  return res;
```

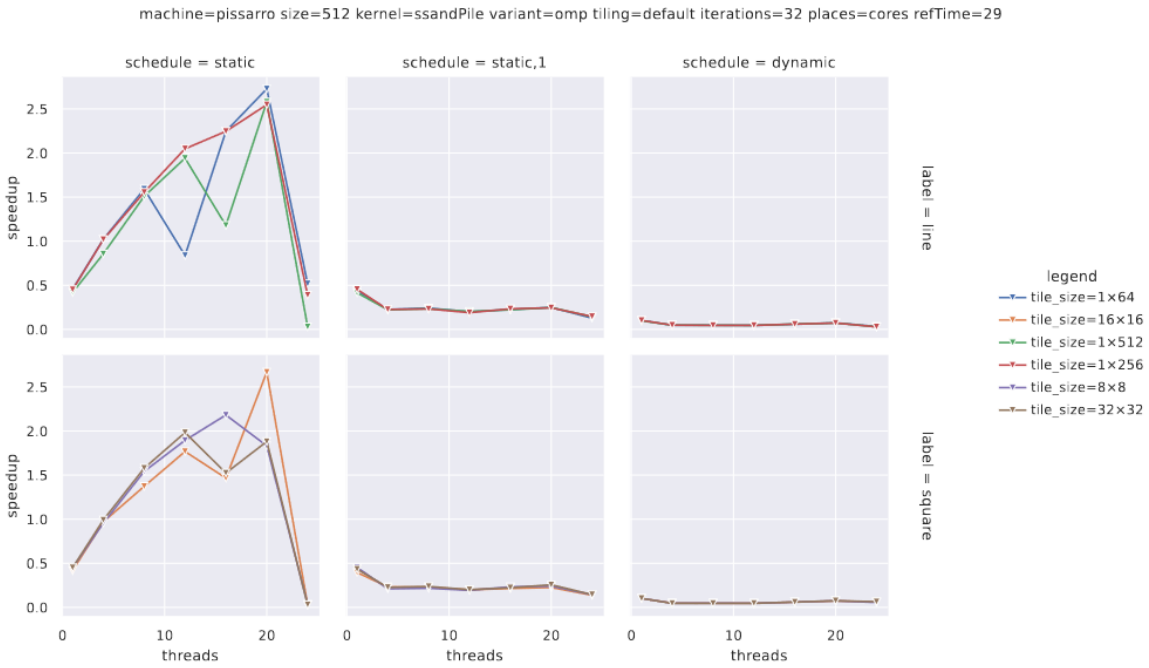

Figure 1: Enter Caption

On peut voir que pour une image de taille 512x512, au bout de 32 iterations, notre version semble plus rapide avec des tuiles 1x64 ou 16x16 en ordonancement "static" et avec 20 threads.

Par exemple on peut utiliser ***OMP_NUM_THREADS=20 OMP_SCHEDULE=static ./run -k ssandPile -v omp -tw 1 -th 64 -s 512 -i 32 -n***

Voici notre parallelisation de **omp_tiled**

```
unsigned ssandPile_compute_omp_tiled(unsigned nb_iter)
{
  int res = 0;
  //#pragma omp parallel master
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;

    #pragma omp parallel shared(change)
    #pragma omp for collapse(2) schedule(runtime)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        #pragma omp atomic
        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)
    {
      res = it;
      break;
    }
  }

  return res;
}
```

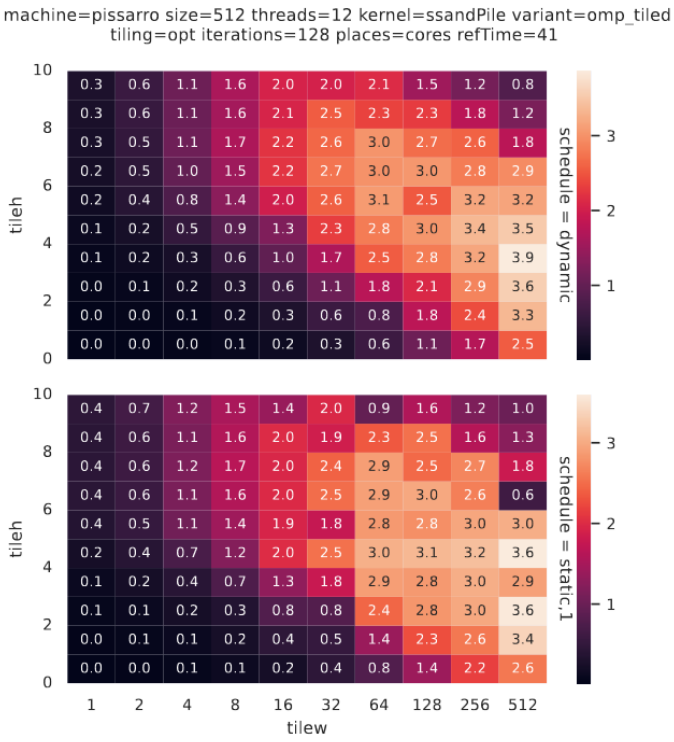

Figure 2: Meilleurs tuiles pour omp_tiled

On a aussi fait une version taskloop.

```
unsigned ssandPile_compute_omp_taskloop(unsigned nb_iter)
{
  int res = 0;

  #pragma omp parallel master
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;

    #pragma omp taskloop collapse(2) grainsize(4) shared(change)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        #pragma omp atomic
        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)
    {
      res = it;
      break;
    }
  }

  return res;
}
```