Rapport Rendu 2

Kristo Dhima
Garion Goubard

March 2022

1 Introduction

4.1 ILP optimization

Nous avons implémenté les fonctions `asandPile_do_tile_opt` et `ssandPile_do_tile_opt`.

```
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++)
    {
      table(out, i, j) = table(in, i, j) % 4 + table(in, i + 1, j) / 4 +
                         table(in, i - 1, j) / 4 + table(in, i, j + 1) / 4 +
                         table(in, i, j - 1) / 4;
      if (table(out, i, j) >= 4)
        diff = 1;
    }

  return diff;
}
```

```
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ ./run -k ssandPile -s 512 -n
Using kernel [ssandPile], variant [seq], tiling [default]
Computation completed after 69190 iterations
179156.591
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ ./run -k ssandPile -wt opt -s 512 -n
Using kernel [ssandPile], variant [seq], tiling [opt]
0Computation completed after 69190 iterations
91517.444
```

Nous pouvons voir que la version otpimisée de la version synchrone est beaucoup plus rapide et efficace que la version non-optimisée, environ 2 fois plus rapide pour etre exact.

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

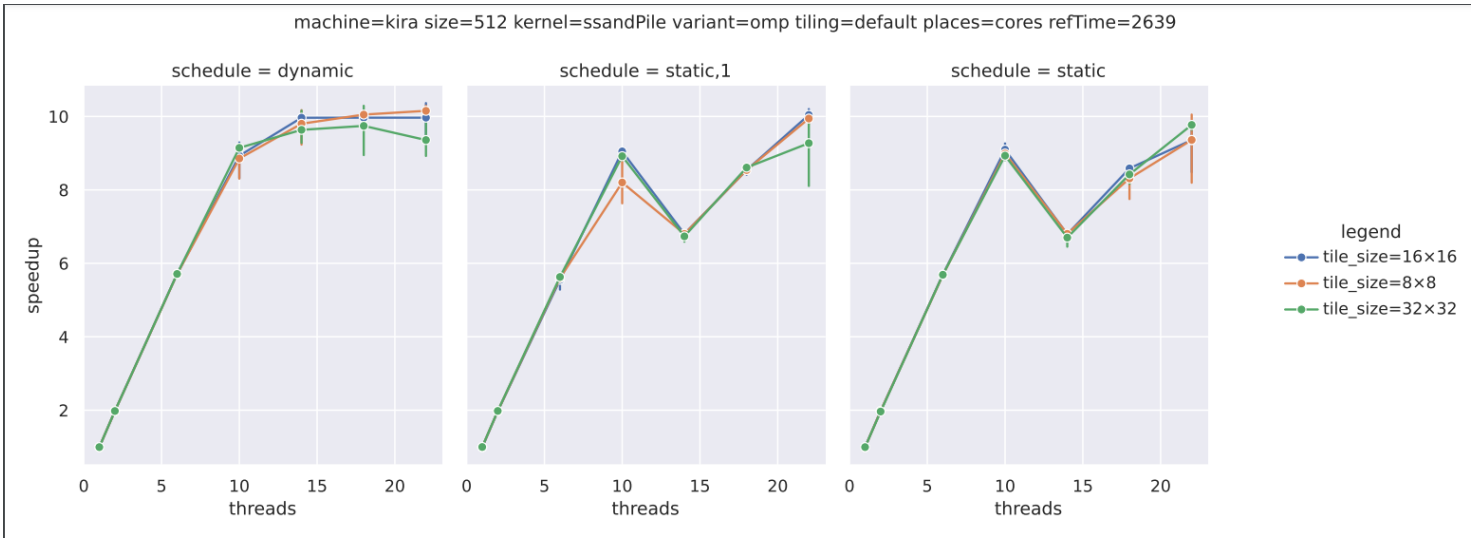
```
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ ./run -k asandPile -s 512 -n
Using kernel [asandPile], variant [seq], tiling [default]
Computation completed after 34938 iterations
38154.392
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ ./run -k asandPile -wt opt -s 512 -n
^[[FUsing kernel [asandPile], variant [seq], tiling [opt]
Computation completed after 34938 iterations
31195.747
```

Nous remarquons aussi une plus légère augmentation de vitesse pour l'optimisation de la version asynchrone par rapport a celle d'avant, mais qui ne peut quand meme pas etre négligée.

4.2 OpenMP implementation of the synchronous version

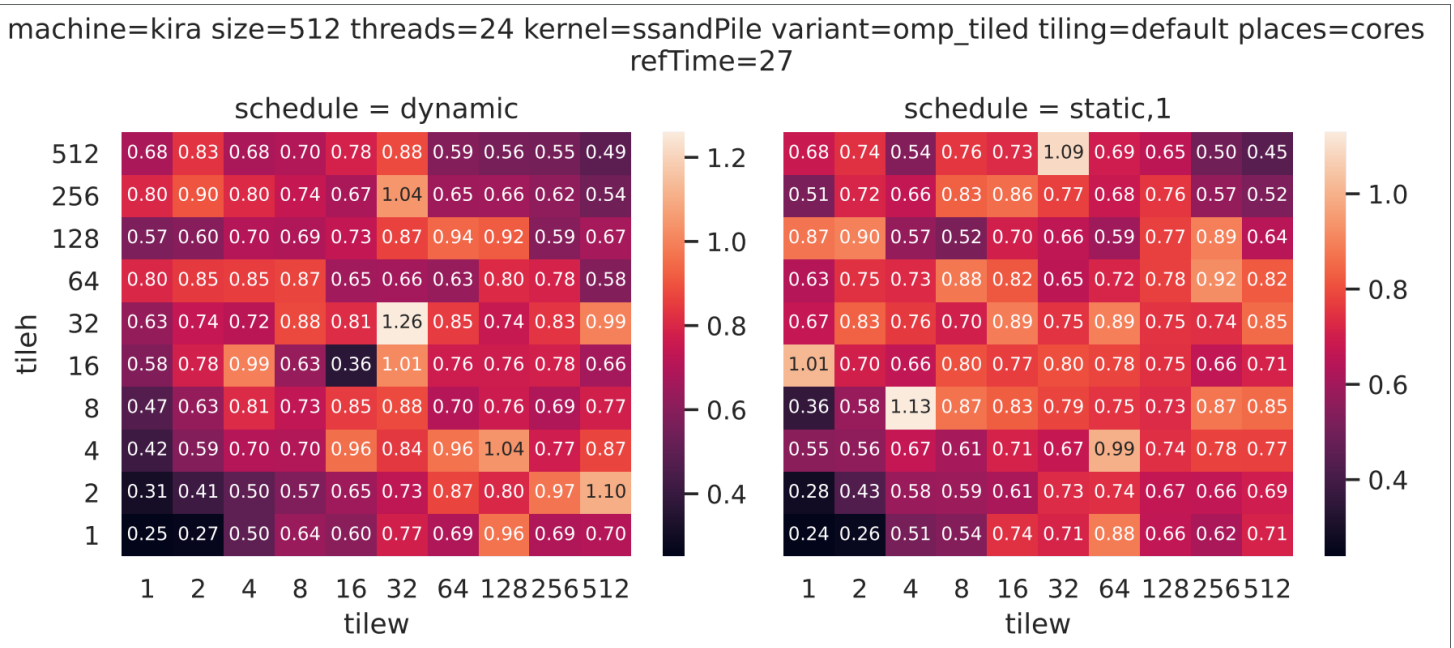
```
unsigned ssandPile_compute_omp(unsigned nb_iter)
{
  unsigned res = 0;
  #pragma omp parallel for schedule(runtime)
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = do_tile(1, 1, DIM - 2, DIM - 2, omp_get_thread_num());
    swap_tables();
    if (change == 0){
      res = it;
  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

```
  return res;
}
```



Nous pouvons voir qu'en utilisant du parallelisme jusqu'a 10 threads, tous les differents types de schedule ont une optimisation importante. Pour plus de 10 threads, soit l'optimisation est legerement amelioré, par exemple dans le cas du schedule dynamic, soit trés deterioré d'un coup et apres amelioré d'un coup egalement(voir graphe au dessus).

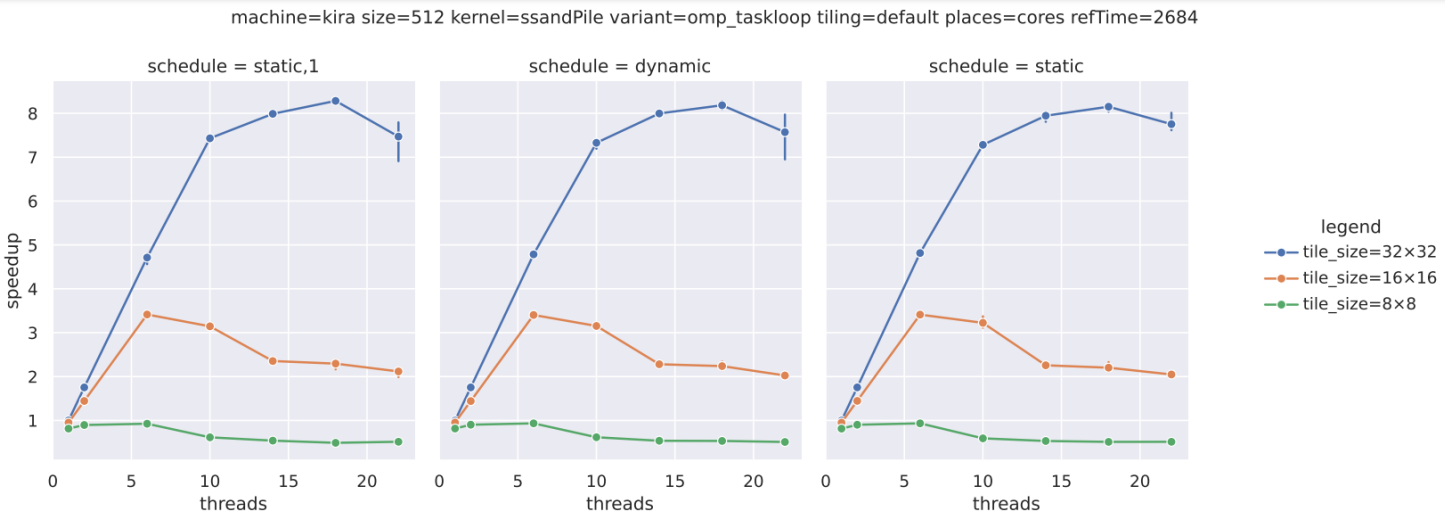
```
unsigned ssandPile_compute_omp_tiled(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;
    #pragma omp parallel for collapse(2)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        change |=
            do_tile(x + (x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

  return 0;
}
```



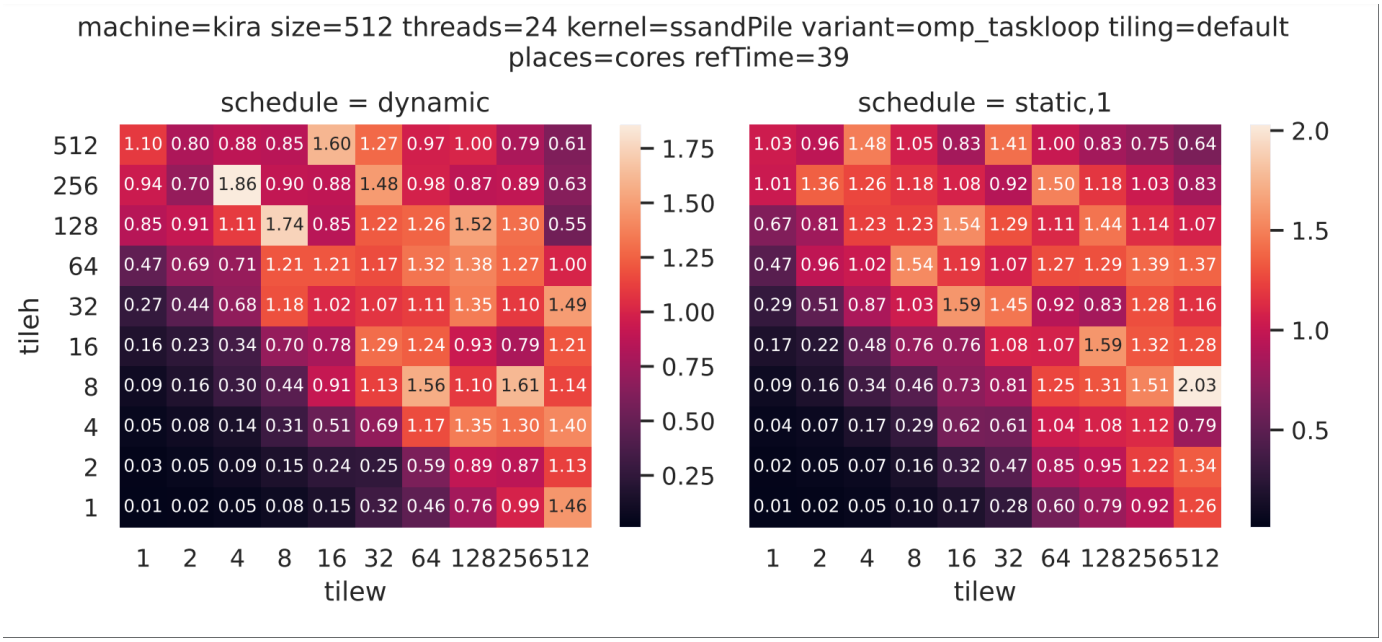
D'apres ce heatmap, nous pouvons deduire qu'en utilisant le tileh et tilew egaux a 32 pour schedule dynamic, et 4 pour static, nous obtenons des meilleurs resultats, ce qui revient a dire que ces valeurs exacts de la taille des tuilles sont les idéales.

```
unsigned ssandPile_compute_omp_taskloop(unsigned nb_iter)
{

  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;
    #pragma omp parallel
    {
      #pragma omp single
      {
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
      #pragma omp task
        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)), omp_get_thread_num());
    }
    }
```

```
      swap_tables();
      if (change == 0)
        return it;
    }

    return 0;
  }
```



D'apres ce plot, nous pouvons deduire que pour `tile_size = 32x32` nous avons une acceleration importante pour toutes les types de schedule, contrairement aux autres `tile_size, 16x16 et 8x8`. Nous pouvons aussi remarquer que les accelerations sont pareilles pour toutes les trois types de schedule. Le seul parametre qui change est la marge d'erreur pour `tile_size = 32x32`.



D'apres ce heatmap, nous pouvons remarquer que les paramètres idéaux pour le type de schedule `dynamic` sont `tilew = 8 & tileh = 128 ou tilew = 4 & tileh = 256` et pour le type de schedule `static` les parametres ideaux sont `tilew = 512 & tileh = 8`. Nous pouvons facilement distinguer que l'algorithme galére en utilisant un `tilew` inferieur a 16 et un `tileh` inferieur à 16.

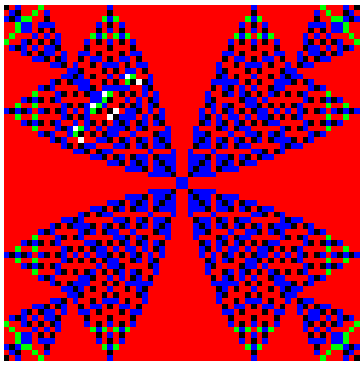
4.3 OpenMP implementation of the asynchronous version

```
unsigned asandPile_compute_omp(unsigned nb_iter)
{

  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;

    #pragma omp parallel for schedule(runtime) collapse(2)
    for(int y = TILE_H; y < DIM; y += TILE_H*2)
      for (int x = TILE_W; x < DIM; x += TILE_W*2)
      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)), omp_get_thread_num());
```

```
    #pragma omp parallel for schedule(runtime) collapse(2)
    for (int y = 0; y < DIM; y += TILE_H*2)
      for (int x = 0; x < DIM; x += TILE_W*2)
        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)), omp_get_thread_num());

    #pragma omp parallel for schedule(runtime) collapse(2)
    for(int y = 0; y < DIM; y += TILE_H*2)
      for (int x = TILE_W; x < DIM; x += TILE_W*2)
      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)), omp_get_thread_num());

    #pragma omp parallel for schedule(runtime) collapse(2)
    for(int y = TILE_H; y < DIM; y += TILE_H*2)
      for (int x = 0; x < DIM; x += TILE_W*2)
      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)), omp_get_thread_num());


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

```
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ OMP_NUM_THREADS=8 OMP_SCHEDULE=dynamic
./run -k asandPile -v omp -s 512 -ts 32 -n
Using kernel [asandPile], variant [omp], tiling [default]
Computation completed after 34919 iterations
9783.082
kdhima@kira:/autofs/unitytravail/travail/kdhima/Master/GL_S8/Programmation_Architectures_Parallelles/pap-3$ OMP_SCHEDULE=dynamic ./run -k asandPile
  -s 512 -ts 32 -n
Using kernel [asandPile], variant [seq], tiling [default]
Computation completed after 34938 iterations
38288.646
```

Nous avons implementé `omp_compute_omp` avec 4 double for. Le programme est plus rapide et plus efficace que la version séquentielle sur des grandes tailles, mais très rarement un bug arrive qui impacte une très petite partie de l'image(voir *dump_bad.png*) plus bas.



4.4 Lazy OpenMP implementations

Nous avons dans cette partie essayé d'implémenter la version séquentielle de lazy. La première version ne correspond pas exactement a l'énoncé mais est plutôt efficace malgré quelques imperfections. Le principe de cette première version est qu'elle vérifie que chaque case dans la tuile est stable. Si une case est instable, alors on calcule toute la tuile et on passe a la suivante. Malheureusement cette version effectue donc plus d'opérations que la version séquentielle, mais elle n'effectue pas les `do_tile` sur les tuiles qui n'en ont pas besoin.

```
unsigned asandPile_compute_lazy(unsigned nb_iter)
{ //Version aussi rapide mais sans effectuer de calculs inutiles
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;
    for (int y = 0; y < DIM; y += TILE_H){
      for (int x = 0; x < DIM; x += TILE_W){
        for(int i = y; (i < y + TILE_H) && i < DIM; i++){
          bool leave = false;
          for(int j = x; (j < x + TILE_W) && j < DIM; j++){
            if(atable(i,j) >= 4){
              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)), 0);
              leave = true;
              break;
            }
          }
          if(leave) break;
        }
      }
    }
```

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

La seconde version de lazy est plus respectueuse de l'énoncé mais nous avons quelques problèmes que nous n'avons pas pu régler. Nous vérifions chaque voisins de la tuile que nous voulons calculer pour voir si à la dernière itération elle était stable ou non. On peut le savoir en sauvegardant la valeur du `do_tile` dans un tableau de booléens. On vérifie donc à chaque fois les les valeurs booléennes des tuiles voisines sont vrai. Si elles sont vrai, c'est qu'elles sont stables, sinon elles sont instables et donc ont besoin de calcul. Pour la première itération, on initialise le tableau à `false` pour que toutes les tuiles soient instables et donc que notre algorithme les calcule toutes. A la prochaine itération, les valeurs sont donc à jour.

```
unsigned asandPile_compute_lazy2(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;

    for (int y = 0; y < DIM; y += TILE_H){
      for (int x = 0; x < DIM; x += TILE_W){
        bool calculated = check_unstable_neighbors(x,y) &&
            do_tile(x + (x == 0), y + (y == 0),
                    TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                    TILE_H - ((y + TILE_H == DIM) + (y == 0)), 0 /* CPU id */);
        change |= calculated;
        setStable(y,x,!calculated);
      }
    }
    if (change == 0)
      return it;
  }
  return 0;
}
```