Comment faire de jolis pâtés

Aguibou Barry Lucas Palacz

22 mars 2022

Première partie

Optimisations et parallélisme de base

1 Optimisations du pipeline

1.1 Avec le noyau synchrone

```
1  int ssandPile_do_tile_default(int x, int y, int width, int height) {
2      int diff = 0;
3
4      for (int i = y; i < y + height; i++)
5          for (int j = x; j < x + width; j++) {
6              table(out, i, j) = table(in, i, j) % 4;
7              table(out, i, j) += table(in, i + 1, j) / 4;
8              table(out, i, j) += table(in, i - 1, j) / 4;
9              table(out, i, j) += table(in, i, j + 1) / 4;
10             table(out, i, j) += table(in, i, j - 1) / 4;
11             if (table(out, i, j) >= 4)
12             diff = 1;
13         }
14
15     return diff;
16 }
```

Listing 1 – Code original de PA. Wacrenier

À la vue de ce code, la première idée que nous avons est de supprimer la série de `+=` sur les lignes 8 à 11, qui crée un effet de dépendance dans l'ordre des instructions. En remplacant par un humble `+` on permet au processeur d'éxécuter les opérations dans l'ordre qu'il l'arrange, voir de les faire en parallèle.

Dans un seconde temps, un camarade avec qui nous échangons sur le projet, Hugo Devidas, nous conseille d'ajouter l'attribut `restrict` au pointeur `TABLE` (qui contient la représentation des gains de sables). Cela a pour effet de diminuer considérablement le temps de calcul (voir graphique ci-dessous). En effet, nous pensons que l'usage de cet attribut nous évite beaucoup d'accès mémoire, dans lesquels on vérifie que la variable n'a pas été modifiée par concurrence entre deux calculs.

Finalement la dernière optimisation réalisée est celle des directives `#pragma` pour dérouler les boucles à la compilation et ainsi gagner légèrement en temps de calcul.

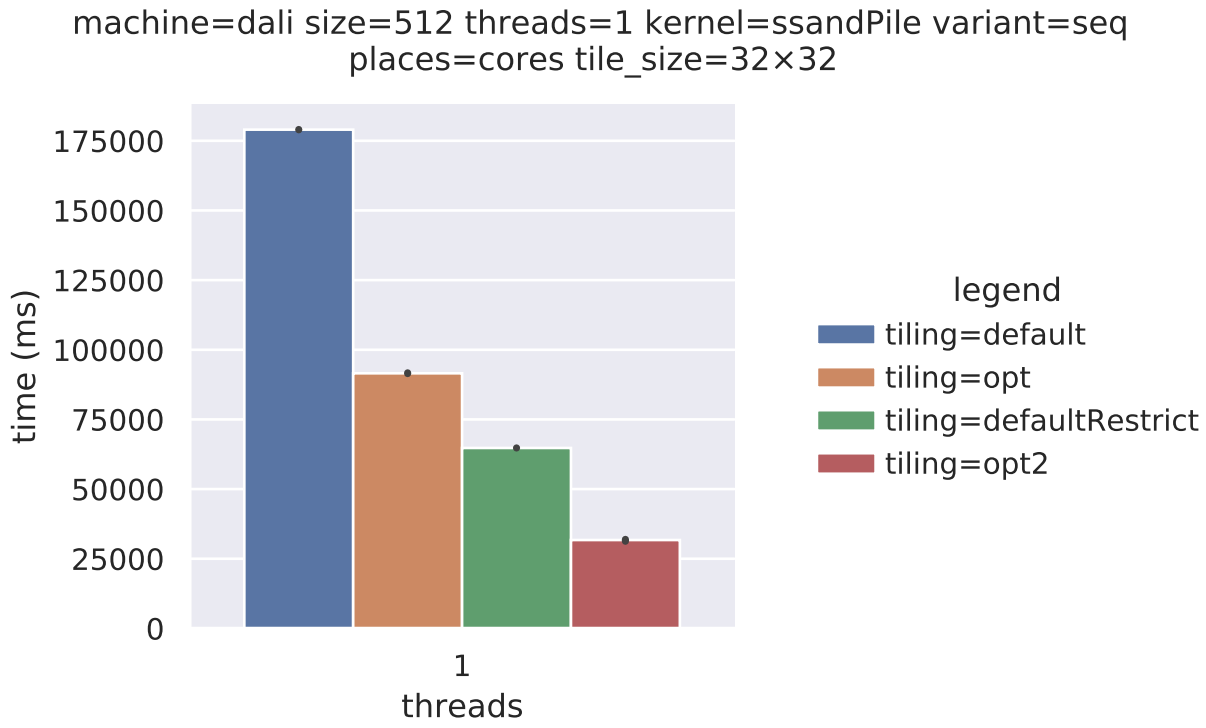

FIGURE 1 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
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;
}
#pragma GCC pop_options
```

Listing 2 – Code optimisé final

1.2 Et pour le noyau asynchrone

```
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;
```

```
}
```

Listing 3 – Code original de PA. Wacrenier

La première idée que nous avons est de supprimer la partie de code répétitif `atable(i, j) / 4;` en utilisant une variable locale à la place. Cela a permis de gagner 10 % de performances. Nous avons aussi remarquer que, par l'utilisation de cette variable, nous ne sommes plus contraint d'avoir l'opération `atable(i, j) %= 4;` à la fin. En la placant au début nous remarquons un léger gain de vitesse aux alentours de 5 %. Nous pensons que cela peut-être lié au fait qu'on n'a optenu la valeur à l'initiation de la variable et que cette dernière se trouve maintenant dans un registre. Tandis que précédament nous devions aller la chercher plus loin dans les caches.

Nous avons ensuite rendu `TABLE restrict` et déroulé les boucles comme pour le noyau synchrone. Cela a égélement eu de bon résultats sur notre temps de calcul.

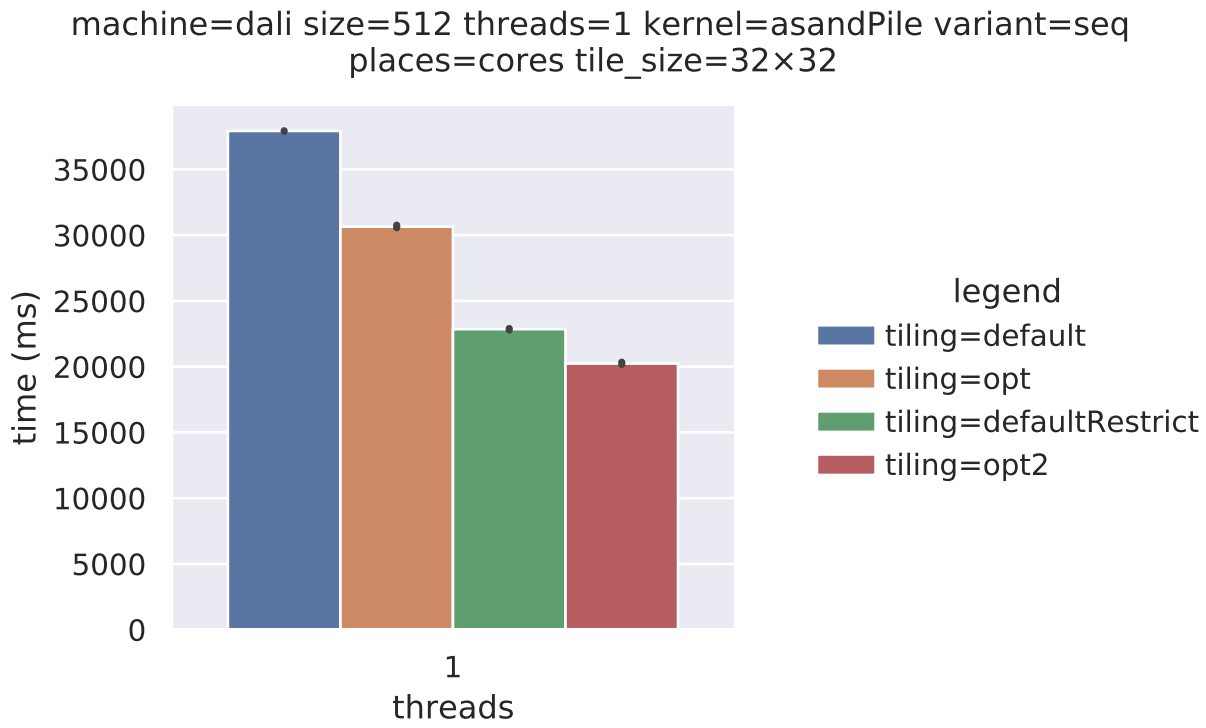

FIGURE 2 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```c
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
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 distributedSand = atable(i, j) / 4;
                atable(i, j) %= 4;
                atable(i, j - 1) += distributedSand;
                atable(i, j + 1) += distributedSand;
                atable(i - 1, j) += distributedSand;
                atable(i + 1, j) += distributedSand;
                change = 1;
            }
    return change;
}
#pragma GCC pop_options
```

Listing 4 – Code optimisé final

2 Parallélisation de base

2.1 Avec le noyau synchrone

Suite aux différentes optimisations réalisées précédemment et surtout au retour qu'on a eu sur le premier rendu concernant l'accès récurrent à la variable *change* par les threads, nous avons apporté quelques modifications sur la parallélisation de ce noyau en utilisant une clause (*reduction(| :)*) sur la variable *"change"* cela nous a permis d'obtenir le même nombre d'itérations que la version séquentielle ce qui n'était forcément pas le cas d'une exécution à l'autre avec l'ancienne implémentation même si l'image finale était celle attendu. Nous utilisons la directive *#pragma omp parallel* avec une politique de distribution qui sera défini lors de l'exécution du programme(*schedule(runtime)*) pour nos différentes implémentations.

— **sandPile_compute_omp()** : Fait à la base de la version sequentielle en utilisant une seule boucle de parcours donc pas de tuile.

```
unsigned ssandPile_compute_omp(unsigned nb_iter) {

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

    #pragma omp parallel for schedule(runtime) reduction(|:change)
    for (int y = 1; y < DIM - 1; ++y)
        change |= do_tile(1, y, DIM - 2, 1, omp_get_thread_num());

    ...

}
```

Listing 5 – Parallélisation de base sans les tuiles

— **sandPile_compute_omp_tiled()** : Cette parallélisation est similaire à la précédente à la différence que dans celle-ci le traitement est fait sur des tuiles et en plus on a rajouté la clause *"collapse(2)"* pour faire la fusion des deux boucles.

```
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) reduction(|: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)), omp_get_thread_num() );

    ...

}
```

Listing 6 – Parallélisation de la variante tuilée

— **sandPile_compute_omp_taskloop()** : Cette parallélisation est faite sur la base de la version avec des tuiles en utilisant la clause *"taskloop"* qui consiste à créer de tâches qu'il y a autant de traitement.

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

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

          #pragma omp taskloop collapse(2) shared(res, change)
          for (int x = 0; x < DIM; x += TILE_W)
            for (int y = 0; y < DIM; y += TILE_H)
              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());

            ...
          }
      }
      return res;
}
```

Listing 7 – Parallélisation avec les tâches

2.2 Pour le noyau asynchrone

```
unsigned asandPile_compute_omp_4tiled(unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    int change = 0;
    #pragma omp parallel
    {
        #pragma omp for collapse(2) schedule(runtime) reduction(|:change)
        for (int y = 0; y < DIM; y += 2*TILE_H)
          for (int x = 0; x < DIM; x += 2*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() );

        #pragma omp for collapse(2) schedule(runtime) reduction(|:change)
        for (int y = 0; y < DIM; y += 2*TILE_H)
          for (int x = TILE_W; x < DIM; x += 2*TILE_W)
            change |= do_tile(x, y + (y == 0), TILE_W - (x + TILE_W == DIM),
                      TILE_H - ((y + TILE_H == DIM) + (y == 0)), omp_get_thread_num() );

        #pragma omp for collapse(2) schedule(runtime) reduction(|:change)
        for (int y = TILE_H; y < DIM; y += 2*TILE_H)
          for (int x = TILE_W; x < DIM; x += 2*TILE_W)
            change |= do_tile(x, y, TILE_W - (x + TILE_W == DIM),
                      TILE_H - (y + TILE_H == DIM), omp_get_thread_num());

        #pragma omp for collapse(2) schedule(runtime) reduction(|:change)
        for (int y = TILE_H; y < DIM; y += 2*TILE_H)
          for (int x = 0; x < DIM; x += 2*TILE_W)
            change |= do_tile(x + (x == 0), y,
                      TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                      TILE_H - (y + TILE_H == DIM), omp_get_thread_num());
    }
    if (change == 0)
      return it;
  }
  return 0;
}
```

Listing 8 – Parallélisation OpenMP du noyau asynchrone

3 Version parallèle optimisée en évitant de calculer les zones stabilisées

Pour pouvoir éviter de calculer les zones stables de la grille, nous avons utiliser un tableau ayant une dimension *size = (NB_TILES_X+2) * (NB_TILES_Y +2)* le plus 2 nous permet de ne pas s'embêter à faire des tests pour savoir si on est sur les cases du bord de la grille.

3.1 Noyau synchrone

3.2 Noyau asynchrone

```
unsigned asandPile_compute_lazy(unsigned nb_iter) {
  for (unsigned it = 1; it <= nb_iter; it++) {
    bool global_change = false; // Has any tile changed during the iteration
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W) {
        int cur_tile = tile(y, x);
```

```
        int row_size = NB_TILES_X + BORDERING_TILES;

        if (CHANGED_TABLE[cur_tile]) {
            global_change |= CHANGED_TABLE[cur_tile] = 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);
      }
      CHANGED_TABLE[cur_tile - row_size] |= CHANGED_TABLE[cur_tile];
      CHANGED_TABLE[cur_tile - 1] |= CHANGED_TABLE[cur_tile];
      CHANGED_TABLE[cur_tile + 1] |= CHANGED_TABLE[cur_tile];
      CHANGED_TABLE[cur_tile + row_size] |= CHANGED_TABLE[cur_tile];
    }
    swap_tables();
    if (!global_change)
      return it;
  }
  return 0;
}
```

Listing 9 – OpenMP : implémentations paresseuses du noyau asynchrone

4 Présentation des expériences

4.1 Noyau synchrone

Pour pouvoir comparer nos différentes implémentations de ce noyau, nous avons commencé par réaliser des expériences sur chaque implémentation en utilisant dans un premier temps une même politique de distribution parmi *"static, dynamic, guided"* en faisant varier le nombre de partage dans les boucles de 1 à 4 pour pouvoir détecter laquelle est meilleure pour ladite politique et dans un second temps nous effectuons une nette comparaison entre le meilleur partage de chaque politique aux autres pour en tirer une conclusion.
Suite à cela, nous réalisons une comparaison des différentes versions de parallélisation implémentées avec des graphiques, carte de chateur et autres pour déterminer laquelle est meilleure.

— **ssand_compute_omp()** : Nos expériences pour cette version sont réalisées sur la machine **gauguin**(24 coeurs, ...) dans la salle 008 du CREMI, donc nous limitons le nombre de threads au nombre de coeurs de la machine.

— **Distribution statique :** Le graphique **??** présente une comparaison selon le nombre de partages(1 à 4) de la politique de distribution *"static"*, 15k mesures ont été faites par points, celles-ci sont stables jusqu'à environ 16 threads et sont parfois bruitées au-delà en raison de .... . De 1 à environ 22 threads nous notons une nette croissance de la courbe de speedup qui est approximativement la même pour les tailles de tuiles (8,16,32,64). Au-delà de 22 threads nous constatons dans un certains cas une décroissance de la courbe et aussi une légère différence entre les tailles de tuiles en raison du non-équilibrage de la charge de travail. Suite à cette analyse, nous voyons que la politique de distribution *"static"* présente une meilleure courbe avec un speedup approximativement égal 11.

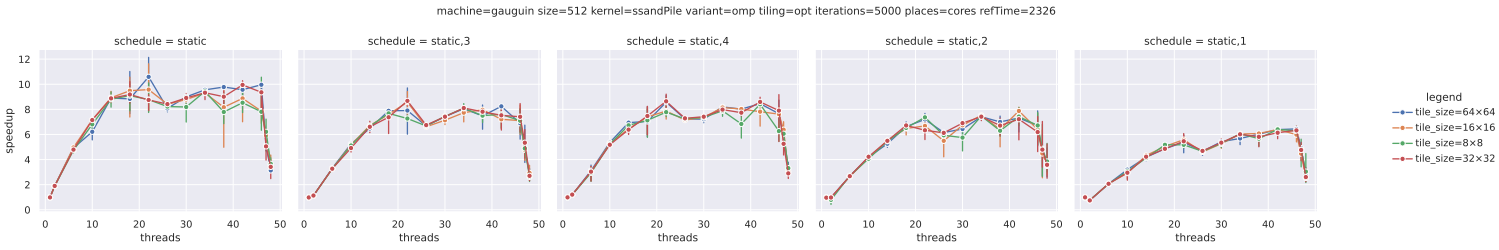

FIGURE 3 – Static, Courbes de speedup de la parallélisation sans tuile

— **Distribution dynamique :** Même constat que la politique de distribution *"static"* avec une légère différence qui est que sur celle-ci il y a moins de bruitages et une énorme chute sur la distribution "nonmonotonic :dynamic" au-delà de 22 threads. Sur celle-ci *"dynamic,4"* présente une meilleure courbe de speedup(voir **??**).

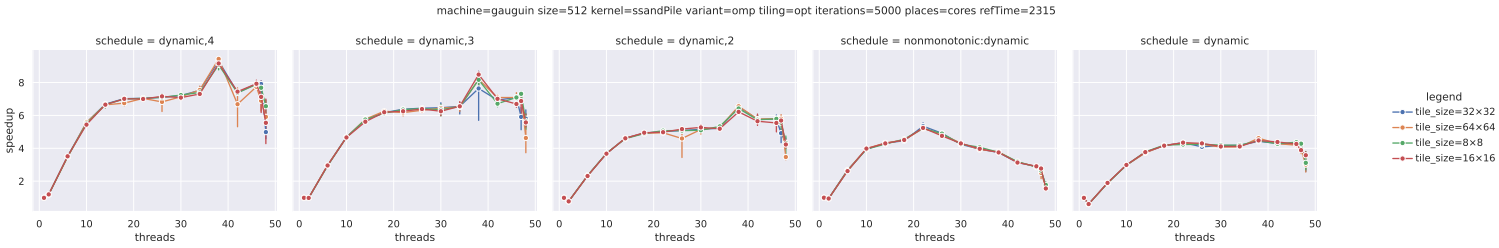

FIGURE 4 – Dynamic, Courbes de speedup de la parallélisation sans tuile

— **Conclusion :** Nous présenterons ci-dessous des graphiques issues de la comparaison des meilleures répartitions obtenues précédemment pour chaque politique de distribution dans le but de déterminer par point laquelle est meilleure.
Dans un premier temps nous avons généré des courbes(voir **??**) présentant les meilleurs stratégies de détecter le nombre de threads présentant un meilleur *"speedup"* pour chaque stratégies et il en sort que sur la distribution *"static"* nous avons deux points(threads : 22 et 46) où on a un meilleur *"speedup"* et threads = 22 et 38 pour la distribution *"dynamic,4"*.
Suite à cela nous avons fait une comparaison entre les points obtenus précédemment pour voir de plus près quelle tailles de tuiles présente vraiment un meilleur *"speedup"*(voir **??** et **??**). Finalement pour la distribution *"static"* la taille 8*8 est meilleur et pour *"dynamic"* c'est quasiment le même pour toutes les tailles.

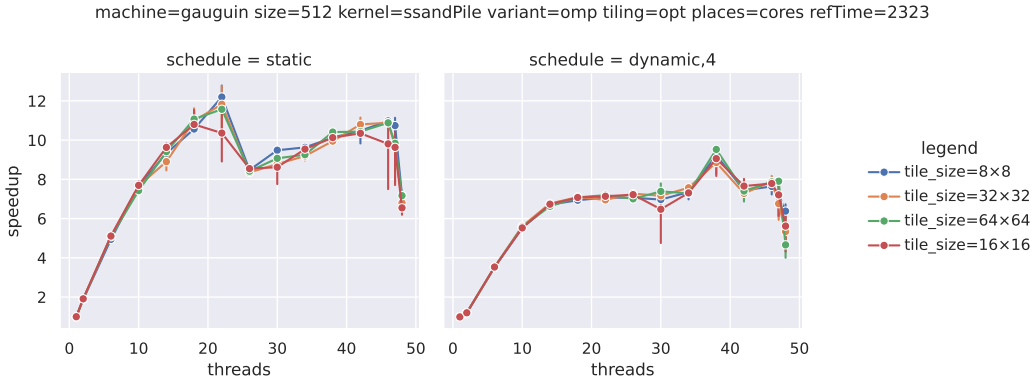

FIGURE 5 – Courbes de speedup de la parallélisation sans tuile

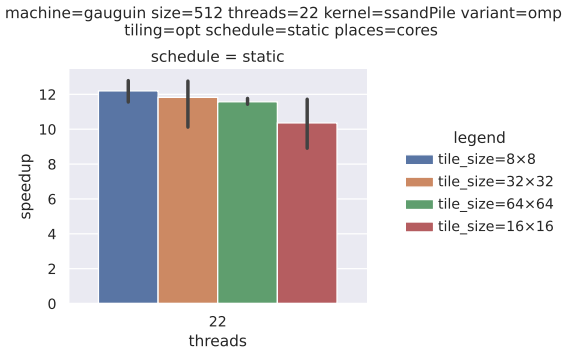

FIGURE 6 – Comparaison de la distribution "static" pour différentes tailles de tuiles

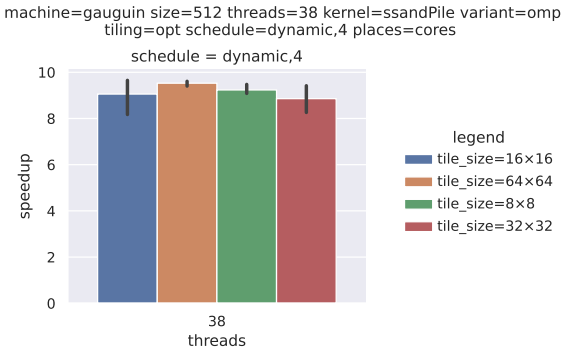

FIGURE 7 – Comparaison de la distribution "dynamic,4" pour différentes tailles de tuiles

— **ssand_compute_omp_tiled()** : Pour cette version, les graphiques ci-dessous sont issus d'expériences réalisées sur la machine **quark**(12 coeurs, CPU : ....) située dans la salle 203 au CREMI, donc nous limitons le nombre de threads au nombre de coeurs de la machine.

Comme précédemment, la même procédure d'expérience a été réaliser pour cette parallélisation et nous avons gardé les meilleures distributions que nous présenterons ci-dessous. Ici nous constatons qu'il y a moins de bruitages sur les stratégies *"dynamic,3"*, *"dynamic,4"* et *"guided"* qui ont aussi des courbes croissantes et on constate également une différence entre les courbes selon le partage des tuiles

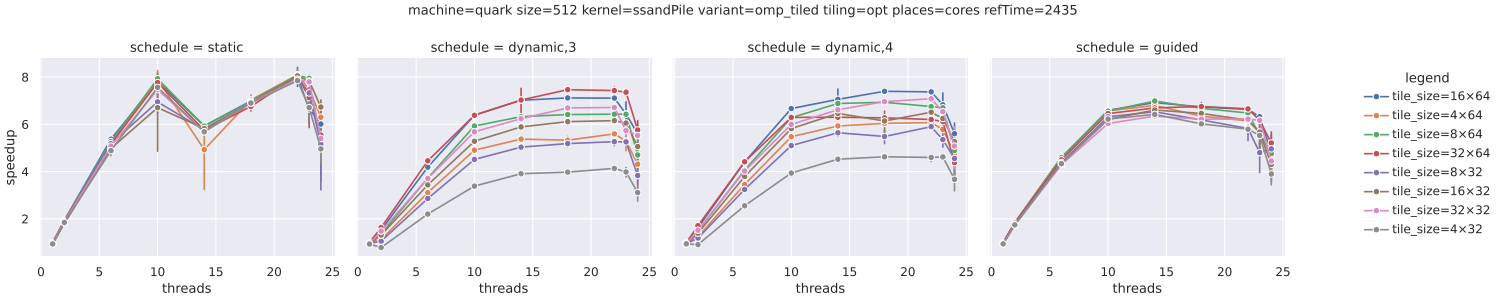

FIGURE 8 – Courbes de speedup de la parallélisation sans tuile

4.2 Noyau asynchrone

4.3 OpenMP implementations

— **asand_compute_omp_tiled()** :

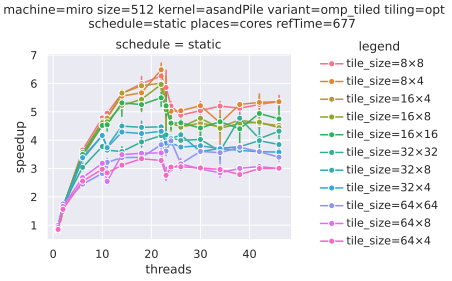

FIGURE 9 – Comparaison de la distribution "static" pour différentes tailles de tuiles

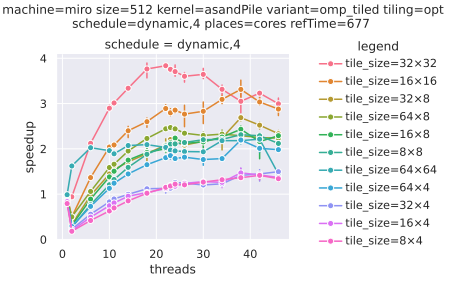

FIGURE 10 – Comparaison de la distribution "dynamic,4" pour différentes tailles de tuiles

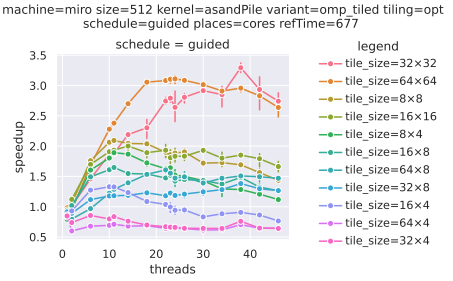

FIGURE 11 – Comparaison de la distribution "guided" pour différentes tailles de tuiles

4.3.1 Lazy OpenMP implementations

Les courbes ci-dessous correspondent à des expériences de la version paresseuse ne traitant pas les zones stabilisées, on voit que la répartition *"static"* présente un meilleur speedup.

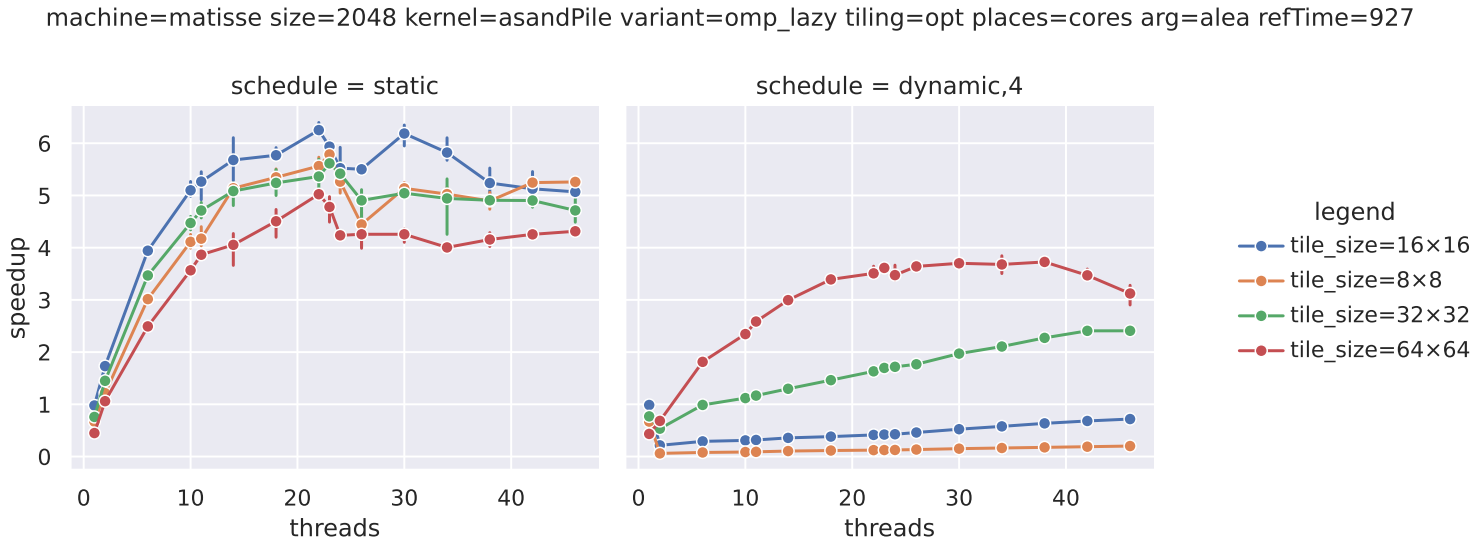

FIGURE 12 – Courbes de speedup du lazy OpenMP sur de la distribution *"static" et "dynamic,4"*

Suite à cette expériences nous allons essayer de trouver le nombre de threads et la répartition des tuiles qui présentent un meilleur speedup grâce à une carte de chaleur que nous présesentrons ci-dessous. Pour cela nous avons répérés quelques points présentant un meilleur speedup sur la courbe précedente(threads = 21, 22, 23 30) voir **??**. Nous constatons qu'avec 22 et 23 threads nous avons de meilleur résultat avec une taille de tuiles *tilew = 32 et tileh = 16.*

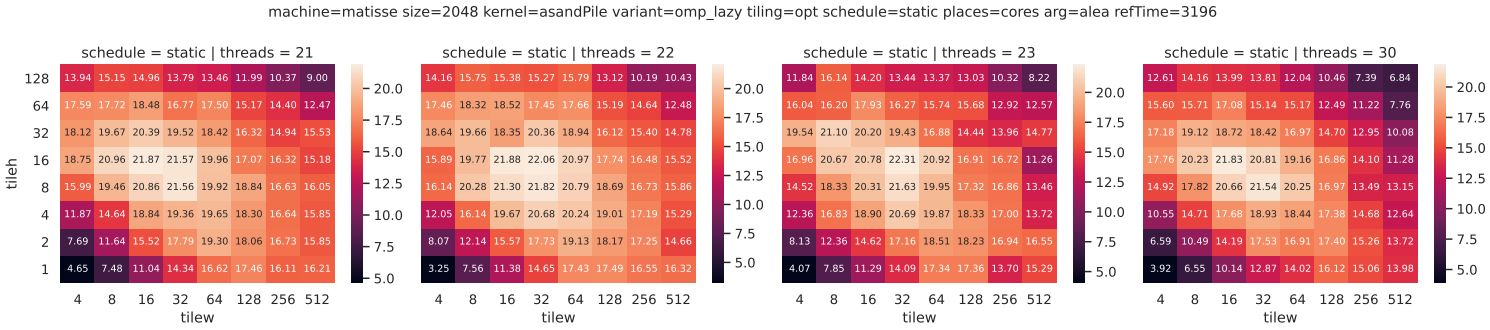

FIGURE 13 – Carte de chaleur de la distribution static avec 21, 22, 23 et 30 threads