pxpx

Rapport final du projet

Programmation des Architectures Parallèles

CAYROL Pierre
FURELAUD Thomas

13 Avril 2020

Résumé

Ce rapport présente les améliorations séquentielles et parallèles apportées au code de base de la gestion d'un tas de sable abélien.

Table des matières

Dans un premier temps, nous allons nous intéresser aux améliorations apportées au niveau séquentiel. Par une analyse des résultats du code de base ainsi qu'une compréhension de celui-ci, deux problèmes majeurs sont ressortis : la gestion de la variable changement, ainsi que le traitement de tuile déjà stable. Ces deux problèmes faisant perdre beaucoup de temps au programme, nous verrons par la suite pourquoi.

1 Non calcul des tuiles stables

Dans le code de base, les tuiles ne sont pas différenciées en fonction de leur stabilité. Ceci cause alors de nombreux calculs effectués pour rien. En effet, dans le cadre par exemple du programme aléa, seulement quelques tuiles sur la grille de base sont instables, il est donc inutile de calculer toutes ces tuiles/cases car l'ancienne et la nouvelle valeur seront les mêmes. Voici alors, comment nous avons procédé pour décider de la stabilité d'une case.

1.1 Code

1.1.1 Déclaration d'un tableau de booléen

```
...
static bool *tab_unstable;
...
#define unstable(i,j) tab_unstable[(i)*(GRAIN+2)+(j)]
...
```

Ce tableau sera utilisé pour connaître la stabilité d'une case. La valeur sera 0 si la case unstable(x,y) est stable et 1 sinon.

1.1.2 Allocation et initialisation du tableau

```
void sable_init ()
{
  ...
  tab_unstable = malloc((GRAIN+2) * (GRAIN+2) * sizeof(int));
  for (int i = 1; i < GRAIN + 1; i++){
    for (int j = 1; j < GRAIN + 1; j++){
      unstable(i,j) = 1;
    }
  }
```

```
  for (int i = 0; i < GRAIN + 2; i++){
    for (int j = 0; j < GRAIN + 2; j+=GRAIN+1){
      unstable(i,j) = 0;
    }
  }

  for (int i = 0; i < GRAIN + 2; i+=GRAIN+1){
    for (int j = 1; j < GRAIN + 1; j++){
      unstable(i,j) = 0;
    }
  }
}
```

Le tableau est initialisé avec des 1 partout sauf sur les bords de la grille où la valeur sera de 0. Les bords n'étant pas pris en compte, la stabilité de ceux la ne nous intéressera pas. On les considérera donc comme stables tout au long du processus.

1.1.3 Désallocation du tableau

```
void sable_finalize ()
{
  ...
  free(tab_unstable);
  ...
}
```

1.2 Nouvelle fonction compute

```
static inline int compute_new_state_stable (int y, int x)
{
  int change = 0;
  if (table (y, x) >= 4) {
    unsigned long int div4 = table (y, x) / 4;
    table (y, x - 1) += div4;
    table (y, x + 1) += div4;
    table (y - 1, x) += div4;
    table (y + 1, x) += div4;
    table (y, x) %= 4;
    change = 1;
  }
  return change;
}
```

1.2.1 Plusieurs *do_tile*

Ici, deux fonction *do_tile* sont créées, une pour les tuiles stables et l'autre pour les tuiles instables. La fonction pour les tuiles instables fait ce que l'ancienne fonction *do_tile* faisait déjà, c'est à dire traiter la tuile avec la fonction compute, case par case. C'est aussi dans ces fonctions que nous décidons de la stabilité de la tuile traitée, avec la variable change qui reste à 0 si aucun changement n'est effectué et qui passe à un sinon.

```
static bool do_tile_unstable(int x, int y, int width, int height, int
    who)
{
  int change = 0;
  monitoring_start_tile (who);
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      change += compute_new_state_stable (i, j);
  monitoring_end_tile (x, y, width, height, who);
  if(change == 0){
    return 0;
  }
  return 1;
}
```

Cependant, la fonction pour les tuiles stables est différente. En effet, celle-ci ne traite que les cases sur les bords de la tuile, car elles sont les seules à pouvoir être modifiées étant donné que tout le reste de la tuile est stable. Effectivement, les valeurs des cases sur les bords de la tuile peuvent évoluer, car elles peuvent être incrémentées par une tuile voisine.

```
static int do_tile_stable(int x, int y, int width, int height, int who)
{
  int change = 0;
  monitoring_start_tile (who);
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j += (width - 1))
      change += compute_new_state_stable (i, j);
  for (int i = y; i < y + height ; i += (height - 1))
    for (int j = x; j < x + width; j ++)
      change += compute_new_state_stable (i, j);
  monitoring_end_tile (x, y, width, height, who);
  if(change == 0){
    return 0;
  }
  return 1;
}
```

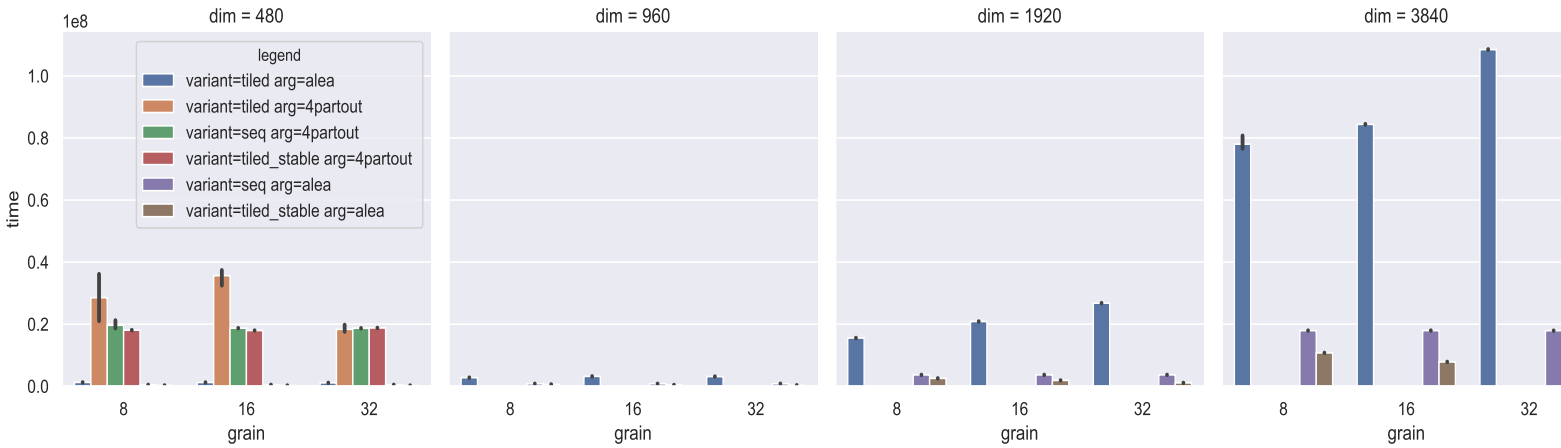
1.2.2 La nouvelle fonction tiled avec l'utilsation d'un tableau

```
unsigned sable_compute_tiled_stable (unsigned nb_iter)
{
  int pos_i, pos_j;
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
    int TS = TILE_SIZE;
    for (int i = 1; i < GRAIN + 1; i+=1)
        for (int j = 1; j < GRAIN + 1; j+=1){
            pos_i = i-1;
            pos_j = j-1;
            if(!unstable(i,j)){
              if(unstable(i-1,j) || unstable(i+1,j) || unstable(i,j-1)
                  || unstable(i,j+1))
                unstable(i,j) = do_tile_stable (pos_j*TS+(pos_j==0),
                                  pos_i*TS+(pos_i==0),
                                  TS-(((pos_j*TS)+TS==DIM)+(pos_j==0)),
                                  TS-(((pos_i*TS)+TS==DIM)+(pos_i==0)),
                                  0);
            }else{
              changement = 1;
              unstable(i,j) = do_tile_unstable (pos_j*TS+(pos_j==0),
                                pos_i*TS+(pos_i==0),
                                TS-(((pos_j*TS)+TS==DIM)+(pos_j== 0)),
                                TS-(((pos_i*TS)+TS==DIM)+(pos_i== 0)),
                                0);
            }
        }

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

Ainsi dans la nouvelle fonction tiled, nous différencions les tuiles de l'image en fonction de leur stabilité. Cela nous permet d'alléger les calculs sur les parties de l'image temporairement stable, où la vérification complète est inutile. Pour ce faire nous avons donc créé le tableau de booléen *unstable*, contenant les valeurs de stabilité correspondant à chaque tuile de notre image. Ensuite, dans le parcours des tuiles de notre fonction on teste si la tuile est stable ou non, afin d'appeler la fonction do-tile la plus adaptée. Comme dit précédemment, la fonction do-tile-unstable (appelée pour les tuiles instables), traite les valeurs de toutes les cases de la tuile et renvoi 0 ou 1 selon si des cases ont changé au cours du traitement (si elle est stable ou non). Et la fonction do-tile-stable (appeler pour les cases stables) traite elle uniquement les cases du bord de la tuile et vérifie si des tuiles voisines ont modifié des cases du bord dans un état instable. La fonction renvoie l'état de stabilité de la tuile. De plus on ne traite pas les tuiles stables, dont tous les voisins sont

stables, car il est logique qu'aucune case de ces tuiles ne soit susceptible de changer.

1.3 Résultats



pxpx

FIGURE 1 – Graphique présentant le temps mit pour une fonction de traiter une image de différente dimension.

Cet algorithme nous permet donc d'économiser le traitement de nombreuses tuiles stables, dont la vérification est inutile. Nous avons réussi à avoir de très bonnes performances sur l'alea (x4.2 par rapport à la tiled de base et seq sur 3840), mais pour 4 partout les tuiles étant tout le temps instable, cela ne crée aucune différence. En effet, il est remarquable sur le graphique, que pour le problème 4*partout* la différence en temps, avec la séquentielle de base est très faible voire inexistante. Cependant, sur le problème aléa, la différence en temps avec la séquentielle de base est de plus en plus grande, plus la dimension de l'image est grande. Sur ce problème, plus la dimension est importante, plus le nombre de tuile stable l'est aussi. Alors de nombreux calculs sont évités et donc le gain de temps est considérable.

2 Gestion de la variable *changement*

Ici le problème était que la variable était affectée de nouveau pour chaque case. Peu importe la stabilité de la case que l'on traitait cette variable était de nouveau affectée. Une affectation ne coûte pas forcément très cher, cependant, dès lors que nous allons vouloir paralléliser le code, un problème va se créer car certains threads voudront probablement modifier cette valeur en même temps ce qui créera de l'attente. Ainsi, moins la variable sera affectée, moins les threads devront la modifier et donc cela réduira l'attente et la concurrence. De plus, même si une affectation n'est pas chère, la différence entre le nombre d'affectations faites sur toutes les cases du tableau à chaque itération et le nombre d'affectations faites seulement sur les cases instables est énorme. Ainsi nous devrions par la suite, apercevoir une assez grande amélioration sur la version parallélisée et une différence faible mais notable sur la version séquentielle.

2.1 Code

2.1.1 Nouvelle fonction *compute_new_state_stable*

```
static inline int compute_new_state_stable (int y, int x)
{
  int change = 0;
  if (table (y, x) >= 4) {
    [...]
    // changement = 1;
    change = 1;
  }
  return change;
}
```

Avant le changement du code de cette fonction, la variable *changement* était passée à 1 dans le if. Ainsi, la variable changement était très souvent modifiée au cours de l'exécution d'un programme. L'affection de cette variable a donc été supprimée dans cette fonction, et nous verrons par la suite, la nouvelle gestion de celle-ci.

2.1.2 Changement dans la fonction *compute_tiled_stable*

```
unsigned sable_compute_tiled_stable (unsigned nb_iter)
{
  int pos_i, pos_j;
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
```

```
    for (int i = 1; i < GRAIN + 1; i+=1)
        for (int j = 1; j < GRAIN + 1; j+=1){
            pos_i = i-1;
            pos_j = j-1;
            if(tab_stable[i][j] == 0){
              if(tab_stable[i-1][j]==1 ||
                    tab_stable[i+1][j]==1 ||
                    tab_stable[i][j-1]==1 ||
                    tab_stable[i][j+1]== 1)
                        /*... Cas stable ...*/
            }else{
              changement = 1;
              /*... Cas instable ...*/
            }
        }
    if (changement == 0)
      return it;
  }
  return 0;
}
```

C'est ainsi dans cette fonction que la variable *changement* est modifiée. Cette variable est modifiée seulement lorsque l'algorithme traite une case instable. Ainsi cette variable n'est pas modifiée pour toutes les cases qui sont stables dans le tableau, ce qui permet donc de faire moins d'affection, mais le gain n'est pas significatif dans la version séquentielle.
C'est surtout dans la version parallèle que nous avons eu un gain considérable, car cette variable est partagée par les threads. Avec cette version, changement est accédé au plus GRAIN*GRAIN fois pour une itération, alors qu'elle pouvait être accédée jusqu'à DIM*DIM fois dans l'ancienne version. Ainsi, cette variable étant beaucoup moins modifiée, et nous avons multiplié notre speedup d'environ 6 sur 4partout, et presque 4 sur alea .

2.1.3 Changement dans le version parallélisée

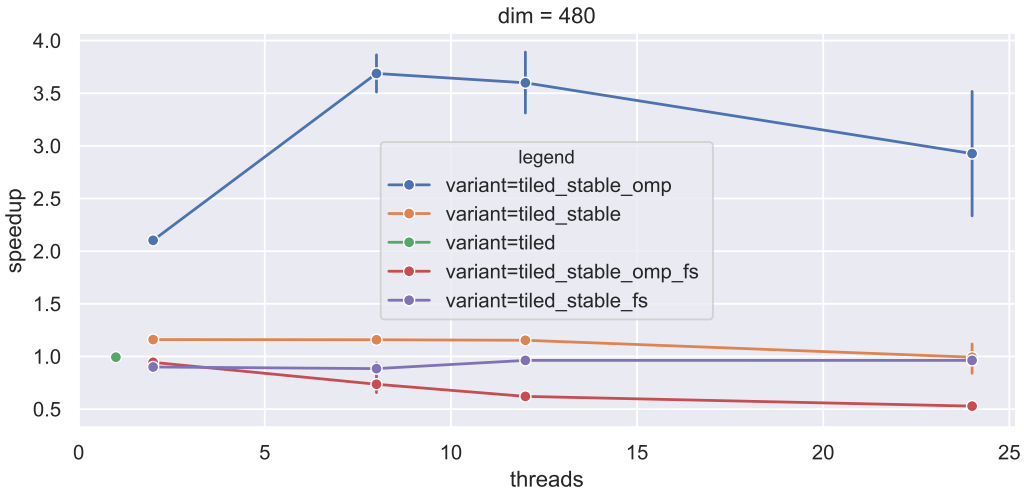
```
unsigned sable_compute_tiled_stable_omp (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
    #pragma omp parallel
    {
      #pragma omp for collapse(2) reduction(+:changement)
      for (int i = 1; i < GRAIN + 1; i+=2)
        for (int j = 1; j < GRAIN + 1; j+=2){
            int TS = TILE_SIZE
            if(!unstable(i,j)){
              if(unstable(i-1,j) ||
                unstable(i+1,j) ||
                unstable(i,j-1) ||
                unstable(i,j+1) ){
```

```
                    unstable[i][j] = do_tile_stable (
                      (j-1) * TS + ((j-1)==0),
                      (i-1) * TS + ((i-1)==0),
                      TS-((((j-1)*TS)+TS==DIM)+((j-1)==0)),
                      TS-((((i-1)*TS)+TS==DIM)+((i-1)==0)),
                      omp_get_thread_num()
                    );
                }
              }else{
                changement += 1;
                unstable(i,j) =
                  do_tile_unstable(
                    (j-1) * TS + ((j-1)==0),
                    (i-1) * TS + ((i-1)==0),
                    TS-((((j-1)*TS)+TS==DIM)+((j-1)==0)),
                    TS-((((i-1)*TS)+TS==DIM)+((i-1)==0)),
                    omp_get_thread_num()
                   );
              }
          }

        #pragma omp for collapse(2) reduction(+:changement)
        for (int i = 2; i <GRAIN+1; i+=2)
          for (int j = 1; j < GRAIN+1; j+=2){
              [...]
          }

        #pragma omp for collapse(2) reduction(+:changement)
        for (int i = 1; i <GRAIN+1; i+=2)
          for (int j = 2; j < GRAIN+1; j+=2){
              [...]
          }

        #pragma omp for collapse(2) reduction(+:changement)
        for (int i = 2; i <GRAIN+1; i+=2)
          for (int j = 2; j < GRAIN+1; j+=2){
              [...]
          }
      }
      if (changement == 0)
        return it;
    }
    return 0;
}
```

Ici, comme pour la fonction précédente, la variable *changement* n'est changée que lorsqu'une tuile instable est traitée. Cette variable étant une variable partagée, un *#pragma omp reduction* doit être placé avant pour s'assurer qu'il n'y ait pas de concurrence entre les différents threads travaillant sur cette variable au même moment.
On remarquera aussi dans le code le nouveau if : $if(unstable(i-1,j)||unstable(i+1,j)||unstable(i,j-1)||unstable(i,j+1))$. Ce if sert à gérer le cas où la tuile est stable et qu'aucun de ses voisins n'est instable. En effet, dans ce cas-là,

il ne sert à rien de s'attarder sur le cas de cette tuile, et nous passons à la tuile suivante.

2.2 Résultats



pxpx

FIGURE 2 – Graphique comparant les versions de différentes fonctions avec ou sans un traitement correcte de *changement*

Comme on peut le voir, 5 programmes sont testés ici. *tiled_stable_omp_fs* est un programme parallélisé contenant une variable partagée et *tiled_stable_omp* est le même programme qui lui n'en contient pas. Les programmes *tiled_stable_fs* et *tiled_stable* sont eux, séquentiel, celui avec l'extension fs contenant une variable partagée et le second non. Pour finir *tiled* est la version séquentielle tuilée de base.

On peut voir sur ce graphique, que les modifications apportées dans la partie **??** améliorent les résultats sur les versions parallélisées. En effet, on remarque que le speed-up est d'un peu moins de 4 pour 8 threads et monte a un peu plus de 3.5 pour 12 threads et plus. Ceci est assez normal car l'utilisation de plusieurs threads n'arrange pas le cas de la variable partagée car

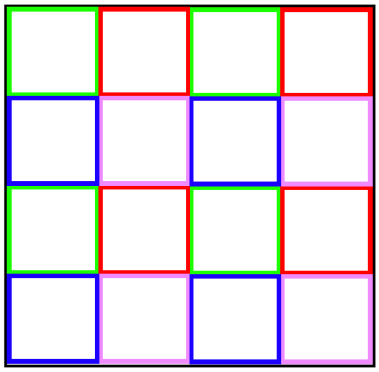
elle sera modifiée encore plus souvent, ce qui ralentira le programme. Or dès que cette variable est correctement traitée, alors les threads peuvent chacun travailler comme il le faut.
Cependant pour les versions séquentielles, on remarque que le speed-up n'est pas remarquable. Nous pouvons l'expliquer comme précédemment : les performances ne sont pas forcément bien plus mauvaise avec une variable partagée car ici seulement 1 thread travaille sur toute l'image, ainsi il ne serait pas surprenant que la version séquentielle contenant une variable partagée obtienne les mêmes performances que la version séquentielle n'en contenant pas.

3 Parallélisation du code

3.1 Code

```
unsigned sable_compute_tiled_stable_omp (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
    int TS = TILE_SIZE;
    #pragma omp parallel
    {
      #pragma omp for collapse(2) reduction(+:changement)
      for (int i = 1; i < GRAIN + 1; i+=2)
        for (int j = 1; j < GRAIN + 1; j+=2){
            if(!unstable(i,j)){
              if(unstable(i-1,j) || unstable(i+1,j) || unstable(i,j-1)
                  || unstable(i,j+1))
                unstable(i,j) = do_tile_stable ((j-1)*TS+((j-1)==0),
                                  (i-1)*TS+((i-1)==0),
                                  TS-((((j-1)*TS)+TS==DIM)+((j-1)==0)),
                                  TS-((((i-1)*TS)+TS==DIM)+((i-1)==0)),
                                  omp_get_thread_num());
            }else{
              changement += 1;
              unstable(i,j) = do_tile_unstable ((j-1)*TS+((j-1)==0),
                                  (i-1)*TS+((i-1)==0),
                                  TS-((((j-1)*TS)+TS==DIM)+((j-1)==0)),
                                  TS-((((i-1)*TS)+TS==DIM)+((i-1)==0)),
                                  omp_get_thread_num());
            }
        }

      #pragma omp for collapse(2) reduction(+:changement)
      for (int i = 2; i <GRAIN+1; i+=2)
        for (int j = 1; j < GRAIN+1; j+=2){
            [...]
        }

      #pragma omp for collapse(2) reduction(+:changement)
      for (int i = 1; i <GRAIN+1; i+=2)
        for (int j = 2; j < GRAIN+1; j+=2){
            [...]
        }

      #pragma omp for collapse(2) reduction(+:changement)
      for (int i = 2; i <GRAIN+1; i+=2)
        for (int j = 2; j < GRAIN+1; j+=2){
            [...]
        }
    }
    if (changement == 0)
      return it;
  }
  return 0;
}
```

Pour paralléliser cet algorithme, nous avons décidé d'utiliser la directive collapse pour paralléliser la double boucle for déjà présente dans *sable_compute_tiled*. Cependant un problème est apparu assez rapidement, celui des variables partagées. En effet, un thread peut modifier les valeurs d'une tuile en même temps qu'un autre thread. Ainsi ces modifications ne seront pas prises en compte. Il fallait donc qu'un seul thread travaille sur une tuile, qu'il finisse son travail et que seulement ensuite, un autre thread puisse travailler dessus. Nous avons donc résolu ce problème en traitant dans un premier temps toutes les rangées paires de tuiles, puis les impaires dans un second temps. Cependant, lorsque le nombre de threads n'était pas égal au nombre de tuiles présentes sur une rangée, certaines valeurs étaient encore partagées. Nous avons donc fini par traiter dans un premier temps les tuiles sur une rangée paire et colonne paire dans un premier temps, puis celle sur une rangée paire et une colonne impaire, ensuite celle sur une rangée impaire mais une colonne paire et pour finir celle sur une rangée impaire et une colonne impaire (illustré ci-dessous).



pxpx

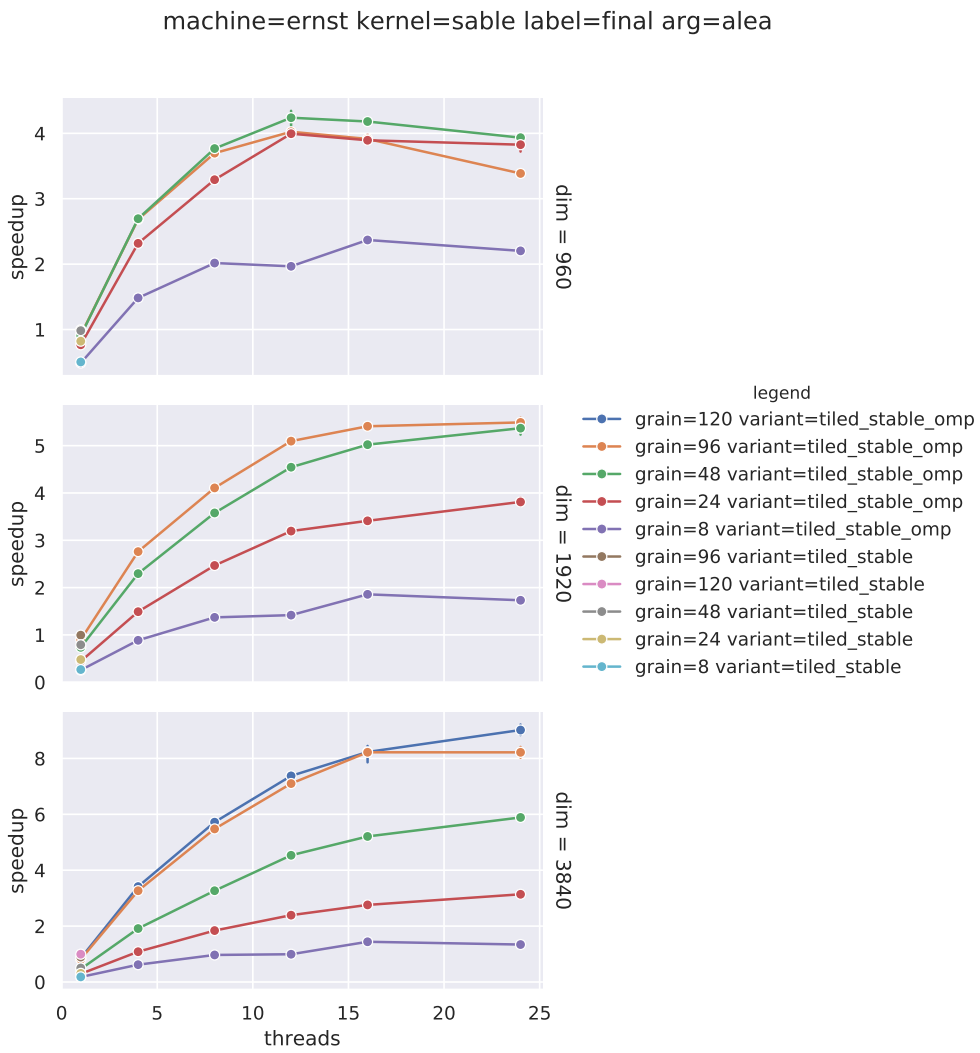
FIGURE 3 – Traitement des tuiles dans l'ordre : vert, rouge, bleu et enfin rose sur une image à 16 tuiles.

Ici les tuiles vertes sont traitées en première, puis ensuite les rouges, puis les bleues et pour finir les roses sont traitées.

Ainsi, les tuiles traitées en même temps par différents threads, ne seront ni les mêmes, ni voisines, ce qui supprime toutes valeurs partagées.
Pour faire ceci, comme on peur voir au-dessus, il faut donc 4 doubles boucles for que l'on parallélise avec collapse(2), sans oublier de protéger la variable *changement* dont on crée une copie locale grâce à la directive *reduction*.

3.2 Résultats de la parallélisation

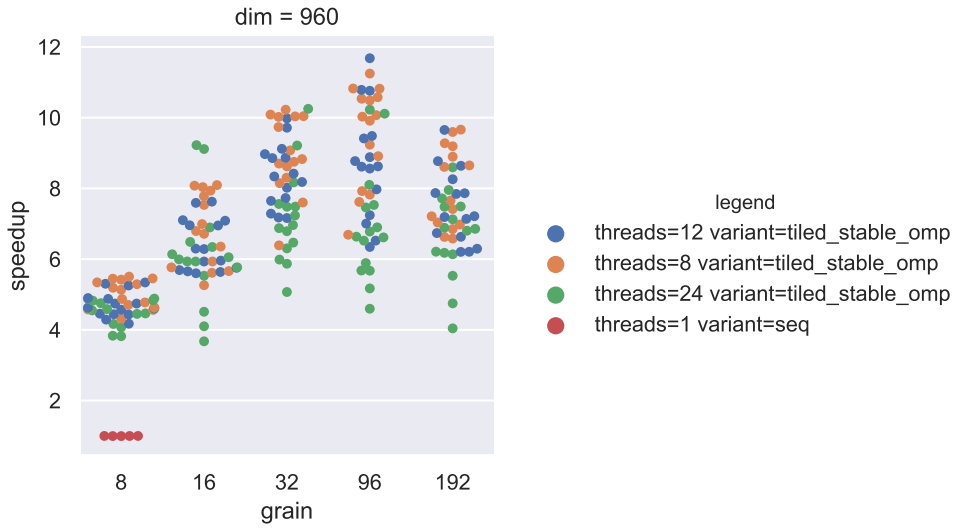
3.2.1 Sur le problème *alea*



pxpx

FIGURE 4 – Graphique représentant le speed-up sur l'étude de grain du programme *alea* d'une dimension de 480 avec différents nombres de threads

Sur ce graphique, on voit que pour une dimension d'image de 480, en moyenne, plus le nombre de threads est faible plus le speed-up est important. Aussi un grain trop élevé est inutile. En effet, plus le grain sera important, plus le nombre de tuile à traiter sera important. Cependant, si le grain est trop faible certains threads ne travailleront pas, de même si le nombre de threads est trop élevé. C'est pourquoi, pour une dimension assez faible (480), un petit nombre de threads est préférable.



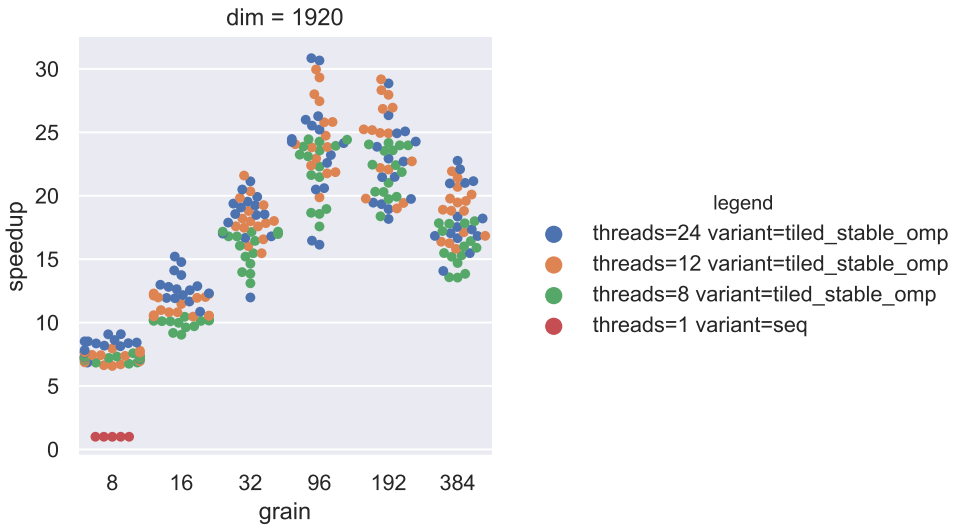
pxpx

FIGURE 5 – Graphique représentant le speed-up sur l'étude de grain du programme *alea* d'une dimension de 960 avec différents nombres de threads

Ici on voit que le speed-up maximal est atteint avec un grain de 96 et en utilisant 12 threads. On peut expliquer cela de la même façon que pour une dimension de 480.

Sur ces deux premières expériences, un compromis entre le nombre de threads et le grain semble intéressant. Comme dit juste avant, l'utilisation de trop de threads ou d'un grain trop grand ralentissent le programme . Ici 8 ou 12 threads et un grain d'un dixième de la dimension du problème est un bon compromis. Nous expliquerons plus bas, pourquoi l'utilisation de 24 threads ralentit le programme. Pour ce qui est du grain, une taille d'un dixième de la

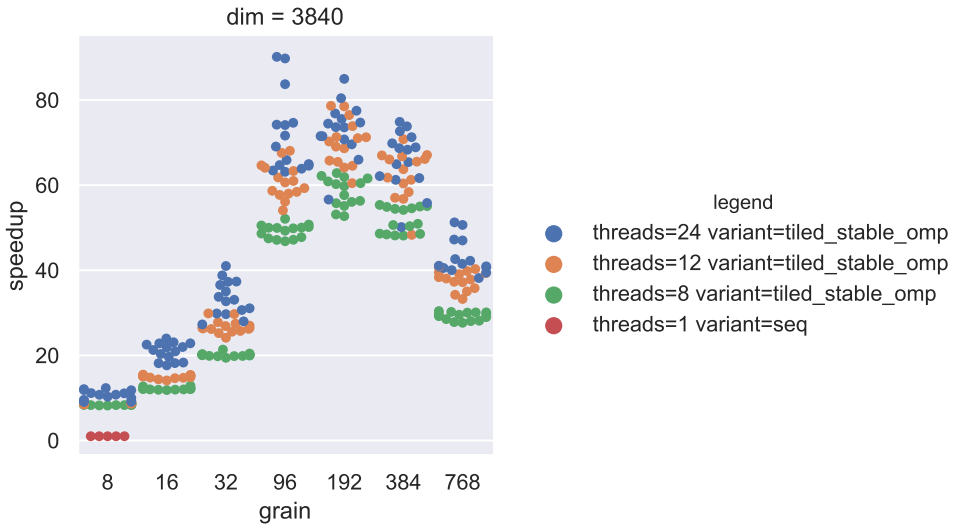
dimension du problème semble optimale. Si on prend un grain faisant donc un dixième de la taille du problème, 10 tuiles seront à traiter sur une ligne, et ceux sur 10 lignes. Or, comme on la vu ici **??**, toutes les deux lignes, une tuile sur deux est traitée. 5 tuiles seront traitées toutes les deux lignes et donc 25 tuiles seront traitées simultanément, ce qui semble optimal. En effet, on utilisera souvent 24 threads, 12 threads ou bien 8 threads. Ainsi dans tout les cas il ne restera qu'une seule tuile à traiter avant d'atteindre la barrière implicite.



pxpx

FIGURE 6 – Graphique représentant le speed-up sur l'étude de grain du programme *alea* d'une dimension de 1920 avec différents nombres de threads

Pour une dimension de 1920, ici encore, un grain de la taille d'un dixième de la dimension du problème est le plus rapide en moyenne. De plus, 12 threads sont suffisants pour cette dimension. Nous pouvons, ici aussi remarquer, que l'utilisation de trop nombreux threads ralentit le programme.

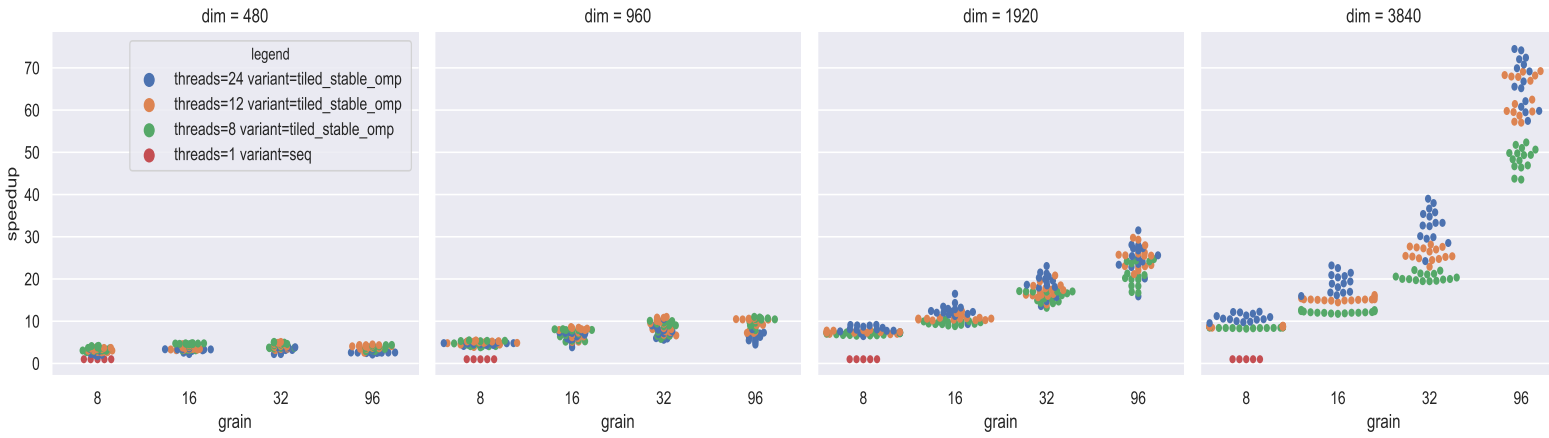


pxpx

FIGURE 7 – Graphique représentant le speed-up sur l'étude de grain du programme *alea* d'une dimension de 3840 avec différents nombres de threads

Ici, en moyenne, l'utilisation de 24 threads et d'un grain de taille 192 présente le meilleur speed-up.

On peut conclure de ces différentes études, que pour une dimension de problème assez faible, il est préférable d'utiliser 8 ou 12 threads et un grain d'une taille d'un dixième de la taille du problème. On verra juste après, pourquoi utiliser 24 threads sur un problème de taille assez faible ralentit le problème. Cependant, dès lors que la taille du problème devient importante, l'utilisation du maximum de threads réels est préférable, cependant le grain doit lui rester d'une taille raisonnable, toujours un dixième du problème voire moins si le problème à une dimension vraiment importante (une taille d'un vingtième est le mieux pour une dimension de 3840).



pxpx

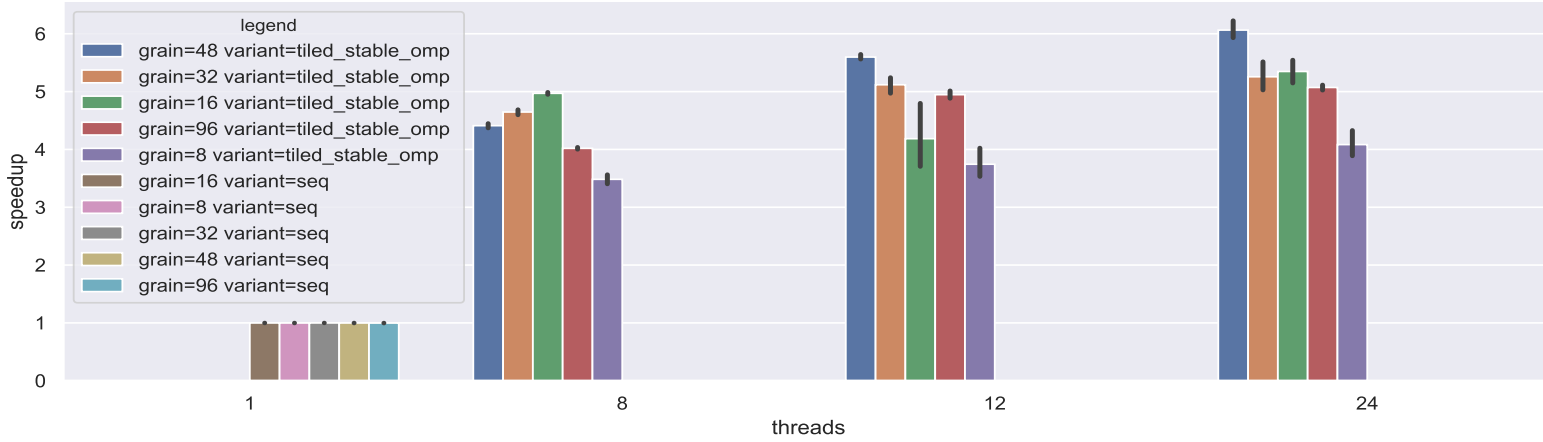
FIGURE 8 – Graphique représentant les speed-up de l'algorithme *tiled_stable_omp* avec différents nombre de threads et sur le programme *alea* avec différentes dimension

Enfin, ici on voit que plus la dimension du problème est grande, plus le speed-up est important. En effet, plus le problème est grand, plus la version séquentielle est lente, or plus la parallélisation est importante, et donc plus le speed-up augmentera.

Pour finir, comme on a pu le voir, pour des dimension de problème assez faible (480, 960 et 1920 sur les expériences), le fait d'utiliser un trop grand nombre de threads ralentit le programme. Ceci s'explique par le nombre d'accès à la variable *changement* simultanément. Comme on l'a dit précédemment, plus le nombre de threads est élevé plus la variable sera appelée à être modifiée et donc plus l'attente pour qu'un thread puisse modifier la variable sera longue. Ainsi, en utilisant 24 threads au lieu de 12, les threads, étant plus nombreux, sont plus souvent en concurrence ce qui ralentit le programme.
Cependant, pour une taille de problème assez grande, le temps perdu par cette attente devient plus faible que le temps gagné à utiliser 24 threads, car comme on a vu dans la partie **??**, le speed-up devient augmente, mais le temps à résoudre le problème lui aussi. Ceci explique donc pourquoi, pour un problème plus petit, le temps gagné a utiliser moins de thread est supérieur

à celui perdu à cause de l'attente dû à la concurrence.

3.2.2 Sur le problème *4partout*



pxpx

FIGURE 9 – Graphique représentant les speed-up de l'algorithme *tiled_stable_omp* avec différents nombre de threads et sur le programme *4partout* et une dimensions de 480

On voit sur le problème *4partout* que la solution optimale est d'utiliser 24 threads avec un grain de 48. Le grain est encore une fois et pour les mêmes raison, un dixième de la dimension du problème, cependant on remarque que l'utilisation de 24 threads est ici plus rapide (on s'attendrait ici, à utiliser 12 threads comme précédemment). Cependant, la différence entre 12 et 24 est assez faible lorsque l'utilisation d'un grain de taille 48 est faite. Ceci est sûrement dû au fait que les cases sont toutes instables, donc tout les threads, nombreux ou pas, vont chercher à modifier changement. Ainsi, de l'attente sera créée dans tout les cas. Ainsi, pour les raisons inverses a celles citées pour l'utilisation de moins de threads pour des petites dimension du problème *alea*, on préférera ici utiliser 24 threads.

4 Vectorisation

4.1 Création d'une nouvelle fonction séquentielle

La fonction *compute_new_state* que l'on utilise dans le code précédent n'est pas adaptée à la vectorisation, car on peut être amené à écrire plusieurs fois dans une case au cours d'une même itération. C'est pourquoi nous avons décidé de créer une nouvelle fonction *compute_new_state*, où chaque case de l'image sera modifiée au plus une fois, afin de simplifier la vectorisation. En effet dans la première version, une case pouvait être modifiée par ses voisins qui lui affectés des grains de sable. Mais dans la nouvelle fonction, c'est la case elle même qui va récupérer les grains de sables, en observant la valeur de ses voisins. Donc on passe de au plus cinq écritures par cases à une écriture max. Pour pouvoir implémenté cela, en plus de notre nouvelle variante de *compute_new_state* ci-dessous, nous avons du mettre en place un système de double tableau avec $NEXT\_TABLE$ en plus de $TABLE$. Le tableau $NEXT\_TABLE$ est, en quelques sortes, un tableau de sauvegarde, qui stocke les nouvelles valeurs des cases à chaque itération. La lecture se fait donc dans $TABLE$, et l'écriture dans $NEXT\_TABLE$. A la fin d'une itération on fait un swap entre $TABLE$ et $NEXT\_TABLE$, de manière à mettre à jour $TABLE$.

```
static inline int compute_new_state_vect (int x, int y)
{
  table_next(x,y) =  table(x,y)%4 + table(x-1,y)/4 + table(x+1,y)/4 +
      table(x,y-1)/4 + table(x,y+1)/4;
  return (table(x,y) != table_next(x,y));
}
```

Comme on le voit, ici à la place d'ajouter 1 à chaque voisin lorsque la case que l'on parcourt est égale ou supérieur à 4, nous ajoutons un à la case en question pour chaque voisin supérieur ou égal à 4. La case est donc modifié qu'une seul fois.

4.2 Vectorisation de la fonction *compute_new_state*

```
static inline __m256i compute_vect_long (int i, int j){

  __m256i top   = _mm256_loadu_si256((__m256i *)&table(i-1,j));
  __m256i curr  = _mm256_loadu_si256 ((__m256i *)&table(i,j));
  __m256i left  = _mm256_loadu_si256((__m256i *)&table(i,j-1));
  __m256i right = _mm256_loadu_si256((__m256i *)&table(i,j+1));
  __m256i bot   = _mm256_loadu_si256((__m256i *)&table(i+1,j));

  right = _mm256_srli_epi64(right,2);
  left  = _mm256_srli_epi64(left,2);
  bot   = _mm256_srli_epi64(bot,2);
  top   = _mm256_srli_epi64(top,2);

  __m256i res = curr;
  res = _mm256_slli_epi64(res,62);
  res = _mm256_srli_epi64(res,62);

  res = _mm256_add_epi64(res,right);
  res = _mm256_add_epi64(res,left);
  res = _mm256_add_epi64(res,bot);
  res = _mm256_add_epi64(res,top);
  _mm256_storeu_si256 ((__m256i *)&table_next(i, j), res);

  __m256i ret = _mm256_xor_si256(res,curr);
  return ret;
}
```

Pour vectoriser cette fonction, les données doivent tout d'abord être lues dans $TABLE$. Les données à récupérer sont les cases à traiter (que nous appellerons *a_traiter* par la suite), celle au dessus de celles-ci, celles en dessous, à leur droite, et à leur gauche. On charge respectivement dans l'ordre top, current, left, right, et bottom afin d'optimiser au mieux les performances du cache.
Ensuite, pour chaque case de *a_traiter*, les valeurs autour de celle-ci sont divisées par 4, en déplaçant la valeur actuelle de deux bits vers la droite. Ensuite, les 4 résultats des divisions sont additionnés entre eux, puis additionnés au modulo 4 de la case *a_traiter*. L'opération modulo 4 étant égal aux deux derniers bits d'un élément, les bits de la valeur sont déplacé de 62 bits vers la gauche, puis déplacé de 62 bits vers la droite, ce qui au final remplace tout les 62 premiers bits par 0.
Ceci est fait pour chaque éléments du vecteur *a_traiter*,puis ce vecteur est stocké dans $NEXT\_TABLE$.
Pour finir, afin de vérifier si il y a eu des changements dans les cases, on retourne le vecteur résultant du ou exclusif entre res et curr. Si ce vecteur contient au moins un bit à 1, cela signifie que res et curr sont différents, et

donc qu'au moins une des cases traitées a changée.

4.3 Fonction *do_tile*

```
static int do_tile_vect_long (int x, int y, int width, int height, int
    who)
{
  __m256i zero = _mm256_setzero_si256();
  __m256i change_vect = zero;
  __m256i res;
  int change = 0;
  int reste = width%4;
  for (int i = y; i < y + height; i++){
    for (int j = x; j < x + width - reste; j+=4) {
      res = compute_vect_long (i, j);
      change_vect = _mm256_or_si256(res, change_vect);
    }
    int j = x + width - reste;
    for(int k = 0;k<reste;k++)
      change += compute_new_state_vect(i,j+k);
  }
  int ret = _mm256_testc_si256(zero, change_vect);
  if (change > 0 || ret == 0)
    return 1;
  return 0;
}
```

Comme nous traitons des long, un vecteur pourra posséder au plus 4 éléments. On incrémente donc de 4 la position horizontale à chaque tour de boucle. Ensuite, lors de la première itération, pour savoir si un changement a eu lieu, le OU logique est fait entre le vecteur retourné par compute_vect_long (qui contient que des zéros si il n'y a eu aucun changement sur les cases traitées) et un vecteur contenant que des 0. Lors des itérations suivantes, ce Ainsi le résultats de ce OU sera 0 si et seulement si il n'y a pas eu de changement. Pour finir, la fonction _mm256_testc_si256 retourne 0 si le ET logique entre le NON zero et change_vect ne renvoie que des 1 bit par bit. Ainsi si change_vect contient un 1 alors ret sera égal à 0 et donc la do_tile_vect_long renverra 1 pour indiquer qu'un changement a eu lieu.

4.3.1 Fonction *compute* pour la vectorisation séquentielle

```
unsigned sable_compute_seq_vect (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
    do_tile_vect_long(1, 1, DIM - 2, DIM - 2, 0);
    swap_tab();
    if (changement == 0)
      return it;
  }
  return 0;
```

```
}
```

4.3.2 Fonction *swap_tab*

```
static inline void swap_tab (void)
{
  long unsigned int *tmp = TABLE;
  TABLE      = TABLE_NEXT;
  TABLE_NEXT = tmp;
}
```

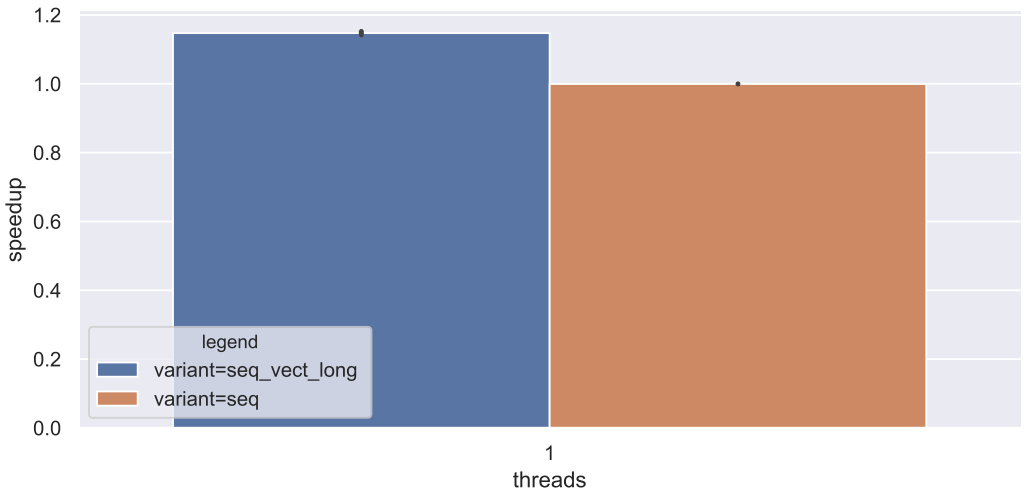
4.4 Parallélisation de la vectorisation

```
unsigned sable_compute_vect_omp (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {
    changement = 0;
    int TS = TILE_SIZE;
    #pragma omp parallel
    {
      #pragma omp for
      for (int y = 0; y < DIM; y += 2*TS)
          do_tile_vect_int(1, y + (y == 0),DIM - 2, TS - ((y + TS == DIM
              ) + (y == 0)),
                  omp_get_thread_num());
      #pragma omp barrier
      #pragma omp for
      for (int y = TS; y < DIM; y += 2*TS)
          do_tile_vect_int(1, y, DIM - 2,TS - (y + TS == DIM),
                omp_get_thread_num());
    }
    swap_tab();
    if (changement == 0)
      return it;
  }
  return 0;
}
```

La parallélisation est très semblable à celle faite dans le premier rapport.

Cependant, on peut voir l'appel à swap_tab() en fin de fonction.

4.4.1 Résultat



pxpx

FIGURE 10 – Speed-up de la vectorisation en AVX2 avec des long par rapport à la séquentielle sur le problème 4*partout* d'une dimension de 480 et un grain de 8.

On voit sur ce graphique que le speed-up de la version vectorisée est de moins d'1.2 avec la version séquentielle de base comme référence sur le problème 4*partout*.

4.5 Utilisation de l'AVX-512

Cependant, nous avons pensé qu'il serait sûrement plus rapide d'utiliser des vecteurs de 8 éléments. Nous avons donc dans un premier temps utilisé l'AVX-512 pour pouvoir travailler avec des vecteurs de 8 long.

4.5.1 Code

```
static inline __m512i compute_vect512_long (int i, int j){
```

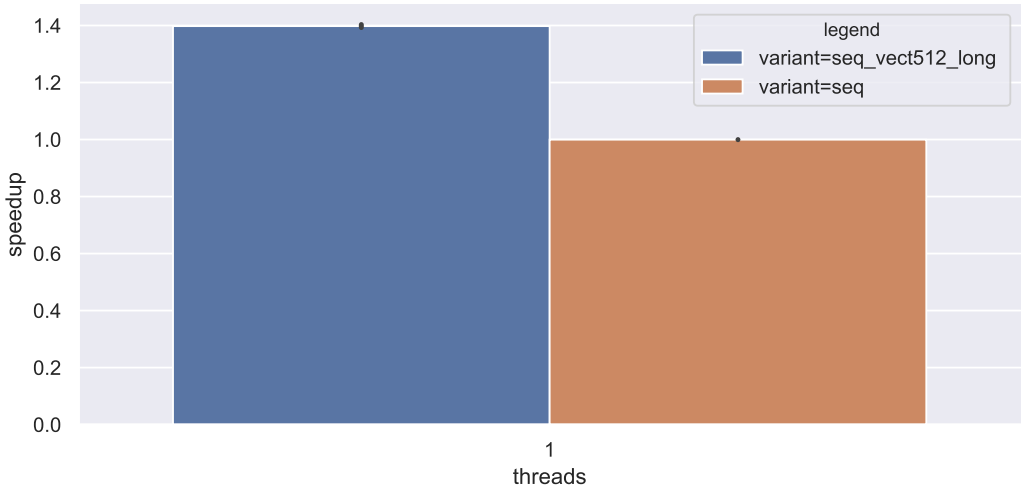
```
  __m512i bot   = _mm512_loadu_si512((__m512i *)&table(i+1,j));
  __m512i curr  = _mm512_loadu_si512 ((__m512i *)&table(i,j));
  __m512i right = _mm512_loadu_si512((__m512i *)&table(i,j+1));
  __m512i left  = _mm512_loadu_si512((__m512i *)&table(i,j-1));
  __m512i top   = _mm512_loadu_si512((__m512i *)&table(i-1,j));

  right = _mm512_srli_epi64(right,2);
  left  = _mm512_srli_epi64(left,2);
  bot   = _mm512_srli_epi64(bot,2);
  top   = _mm512_srli_epi64(top,2);

  __m512i res = curr;
  res = _mm512_slli_epi64(res,62);
  res = _mm512_srli_epi64(res,62);

  res = _mm512_add_epi64(res,right);
  res = _mm512_add_epi64(res,left);
  res = _mm512_add_epi64(res,bot);
  res = _mm512_add_epi64(res,top);

  _mm512_storeu_si512 ((__m512i *)&table_next(i, j), res);
  __m512i ret = _mm512_xor_si512(curr,res);

  return ret;
}
```

Le code de cette fonction est très semblable au code de celle utilisant AVX2. En effet, rien ne change mis à part l'utilisation de vecteur de taille de 512 bits et plus de 256.

```
static int do_tile_vect512_long (int x, int y, int width, int height,
    int who)
{
  int change = 0;
  int reste = width%8;
  __m256i zero   = _mm256_setzero_si256 ();
  __m512i change_vect = _mm512_setzero_si512 ();
  __m512i res;
  for (int i = y; i < y + height; i++){
    for (int j = x; j < x + width - reste; j+=8) {
      res = compute_vect512_long (i, j);
      change_vect = _mm512_or_si512 (change_vect, res);
    }
    int j = x + width - reste;
    for(int k = 0;k<reste;k++)
      change += compute_new_state_vect(i,j+k);
  }
  __m256i res1 = _mm512_cvtsepi64_epi32 (change_vect);
  int ret  = _mm256_testc_si256(zero,res1);//+ _mm256_testc_si256(zero,
      res2);
  if (change > 0 || ret == 0)
    return 1;
  return 0;
}
```

Pour cette fonction, comme on peut le voir, la seule chose qui change est le nombre d'éléments traités à chaque tour de boucle (8), puis le reste qui devient la largeur modulo 8 logiquement. On remarquera aussi, le passage en AVX2 sur la fin du problème, pour pouvoir convertir le vecteur en un int et ainsi savoir si la le processus à procédé à des changements. Ici, des solutions sont possibles pour pouvoir retourner un int en utilisant l'AVX-512, cependant pour des soucis de compilation sur les masks et donc ne pouvant pas les utiliser, nous avons trouvé cette solution.

4.5.2 Résultat



pxpx

FIGURE 11 – Speed-up de la vectorisation en AVX-512 avec des long par rapport à la séquentielle sur le problème 4*partout* d'une dimension 480 et un grain de 8.

On voit sur ce graphique que le speed-up de la version vectorisée en utilisant des vecteurs 8 long est de 1.4 avec la version séquentielle de base comme référence sur le problème 4*partout*. On a donc un speed-up supérieur en utilisant des vecteurs de 8 long plutot qu'en utilisant des vecteurs de 4 long. Ceci s'explique par la rapidité de traitement d'une ligne. En effet, avec une dimension de 480, simplement 60 itérations seront nécessaire à traiter

une ligne avec des vecteurs de taille 8 contre 120 avec des vecteurs de taille 4.

4.6 Utilisation de int dans les vecteurs

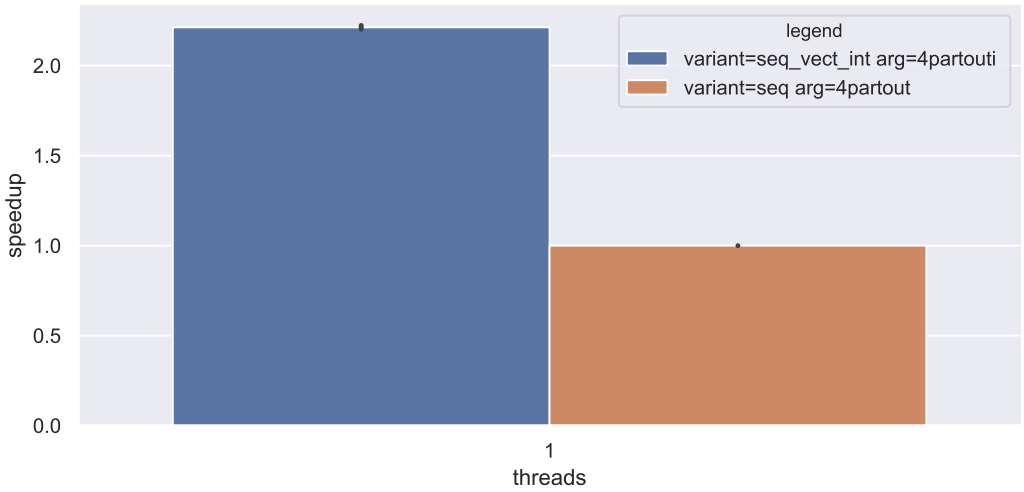
Pour finir, nous avons aussi testé en utilisant des int. En effet, plus les vecteurs sont grands, plus le temps gagné doit être important. En utilisant des int codé sur 32 bits au lieu des long qui eux sont codés sur 64 bits, nous doublons alors le nombre d'éléments dans les vecteurs. Pour ce faire, à la place d'initialiser $TABLE$ avec des long, nous l'initialisons exactement de la même manière mais avec des int. Pour finir, un changement a aussi été effectué dans la fonction do_tile, ou la boucle traitera 8 int à la fois pour AVX2 et 16 éléments à la fois pour AVX-512. Par exemple pour l'AVX-512 :

```
static int do_tile_vect512_int (int x, int y, int width, int height, int
    who)
{
  int change = 0;
  int reste = width%16; //ici le modulo devient logiquement 16
  __m256i zero   = _mm256_setzero_si256 ();
  __m512i change_vect = _mm512_setzero_si512 ();
  __m512i res;

  for (int i = y; i < y + height; i++){
    for (int j = x; j < x + width - reste; j+=16) { //Ici 16 elements
        sont traitees a la fois on passe alors a 16 elements plus loin
      res = compute_vect512_int (i, j);
      change_vect = _mm512_or_si512 (change_vect, res);
    }
    int j = x + width - reste;
    for(int k = 0;k<reste;k++)
      change += compute_new_state_int(i,j+k);
  }
  __m256i res1 = _mm512_extracti32x8_epi32 (change_vect, 0);
  __m256i res2 = _mm512_extracti32x8_epi32 (change_vect, 1);
  int ret  = _mm256_testc_si256(zero,res1) + _mm256_testc_si256(zero,
      res2);
  ...
  if (change > 0 || (ret == 0))
    return 1;
  return 0;
}
```

En AVX-512 on remarquera aussi la facon d'extraire les deux vecteurs de 256 pour encore une fois pouvoir les convertir en int pour savoir si un changement a eu lieu. Nous sommes ici aussi, conscient que cette façon de faire nous font perdre des performances par la latence de cette fonction ($\_mm512\_extracti32x8\_epi3$), cependant nous n'avons pas trouvé de ma-

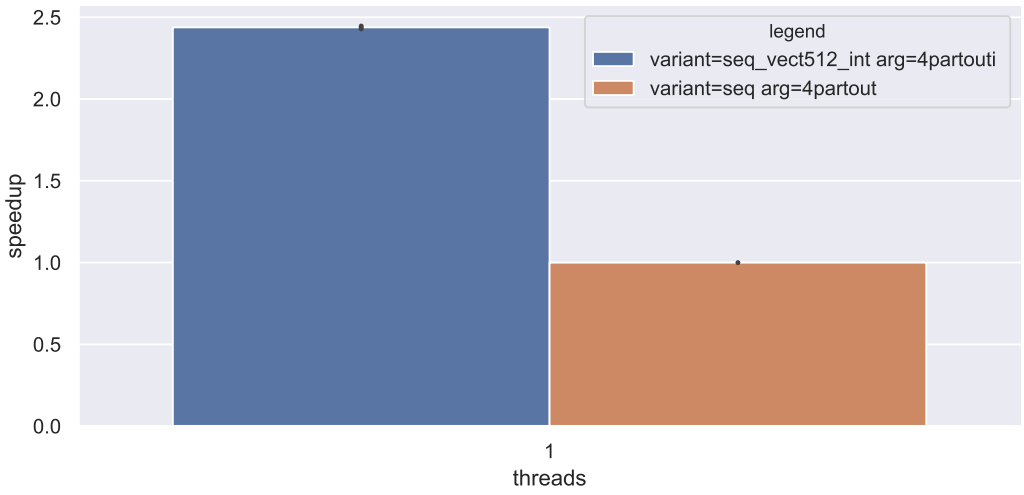
nière plus rapide de faire.

4.6.1 Résultat



pxpx

FIGURE 12 – Speed-up de la vectorisation en AVX2 avec des int par rapport à la séquentielle.



pxpx

FIGURE 13 – Speed-up de la vectorisation en AVX-512 avec des int par rapport à la séquentielle.

On voit sur ces graphes que le speed-up gagné avec des int est de 1. En effet pour AVX-2, un speed-up de 2.3 est remarquable en utilisant des int, avec la séquentielle de base en référence contre 1.2 en utilsant des long. Pour AVX-512, on note un speed-up de pratiquement 2.5 en utilisant des int contre 1.4 en utilisant des long. Un speed-up 2 fois plus élevé est attendu ici, car un vecteurs traite deux fois plus d'éléments qu'en utilisant des long. Pour ce qui est de l'AVX2 ce résultat est atteint avec un speed-up de 2.3 contre un peu moins de 1.2 en utilisant des long (1.15 * 2 = 2.3). Cependant pour l'AVX-512 le speed-up est de 2.45 en utilisant des int contre 1.4 en utilisant des long. Ce résultat n'est pas forcément étonnant car on utilise deux fois la fonction $\_mm512\_extracti32x8\_epi3$ dont la latence est de 3 et le débit de 1. Ceci ralentit alors notre programme.

4.7 Ananlyse du speed-up de la vectorisation

4.8 Présentation générale des résulats