

- code bony à factoriser
  - tenir compte de l'état de la tuile à l'étape précédente (le code ne fonctionne pas si il n'y a qu'une seule tuile par exemple)
- Projet : rapport2

4TIN804U

BERASATEGUY Tanguy, GOEDEFROIT Charles

- Simplifier le code sur un p-tiled (buf?)

2021-2022

## Table des matières

1 ILP optimization (4.1)	2
2 OpenMP implementation of the synchronous version (4.2)	5
3 OpenMP implementation of the asynchronous version (4.3)	7
4 Lazy OpenMP implementations (4.4)	9
5 Résultats produit a la fin	11

## 1 ILP optimization (4.1)

On a fait les modifications :

Pour `ssandPile_do_tile_opt()` on a retiré les appels à `table(out, i, j)` pour passer par une variable intermédiaire `result`. Cette modification permet au compilateur de vectoriser car il peut maintenant facilement voir que les différentes lignes peuvent être calculées en parallèle.

Nous avons modifié ces lignes :

---

```
1 int ssandPile_do_tile_opt(int x, int y, int width, int height)
2 {
3     int diff = 0;
4
5     for (int i = y; i < y + height; i++)
6         for (int j = x; j < x + width; j++)
7             {
8                 -     table(out, i, j) = table(in, i, j) % 4;
9                 +     int result = table(in, i, j) % 4;
10                -     table(out, i, j) += table(in, i + 1, j) / 4;
11                +     result += table(in, i + 1, j) / 4;
12                -     table(out, i, j) += table(in, i - 1, j) / 4;
13                +     result += table(in, i - 1, j) / 4;
14                -     table(out, i, j) += table(in, i, j + 1) / 4;
15                +     result += table(in, i, j + 1) / 4;
16                -     table(out, i, j) += table(in, i, j - 1) / 4;
17                +     result += table(in, i, j - 1) / 4;
18                -     table(out, i, j) = result;
19                -     if (table(out, i, j) >= 4)
20                    diff = 1;
21                +     diff /= result >= 4;
22            }
23
24     return diff;
25 }
```

---

OK

Le code de la fonction final :

---

```
27 int ssandPile_do_tile_opt(int x, int y, int width, int height)
28 {
29     int diff = 0;
30
31     for (int i = y; i < y + height; i++)
32         for (int j = x; j < x + width; j++)
33             {
34                 int result = table(in, i, j) % 4;
35                 result += table(in, i + 1, j) / 4;
36                 result += table(in, i - 1, j) / 4;
37                 result += table(in, i, j + 1) / 4;
38                 result += table(in, i, j - 1) / 4;
```

---

---

```

39         table(out, i, j) = result;
40         diff /= result >= 4;
41     }
42
43     return diff;
44 }
```

---

Nous avons vérifié et on obtient le même résultat et le même nombre d'iterations (69190) avec la version par défaut et la version opt.

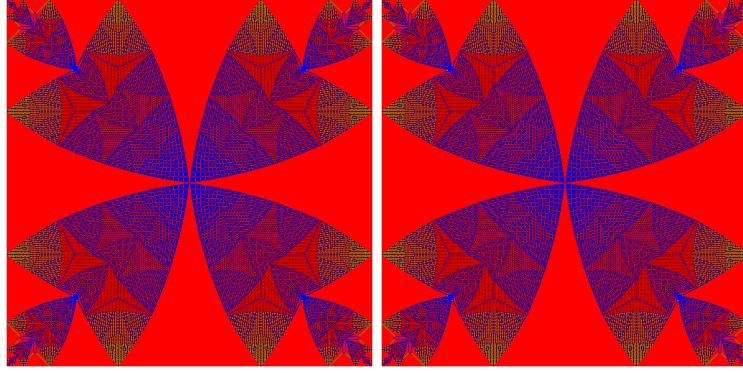


FIGURE 1 – Vérification du résultats pour ssandPile

Le gain de performance est de 2.37 car  $\frac{178812}{75408}$

Pour asandPile\_do\_tile\_opt() on a retiré les appels à atable(i, j) pour passer par une variable intermédiaire result. Cette modification permet au compilateur de vectoriser car il peut maintenant facilement voir que les différentes lignes peuvent être calculés en parallèle.

Nous avons modifié ces lignes :

---

```

1 int asandPile_do_tile_default(int x, int y, int width, int height)
2 {
3     int change = 0;
4
5     for (int i = y; i < y + height; i++)
6         for (int j = x; j < x + width; j++)
7             if (atable(i, j) >= 4)
8                 int result = atable(i, j);
9                 if (result >= 4)
10                 {
11                     result/=4;
12                     atable(i, j - 1) += atable(i, j) / 4;
13                     atable(i, j - 1) += result;
14                     atable(i, j + 1) += atable(i, j) / 4;
```

---

```

15     +      atable(i, j + 1) += result;
16     -      atable(i - 1, j) += atable(i, j) / 4;
17     +      atable(i - 1, j) += result;
18     -      atable(i + 1, j) += atable(i, j) / 4;
19     +      atable(i + 1, j) += result;
20         atable(i, j) %= 4;
21         change = 1;
22     }
23     return change;
24 }
```

---

Le code de la fonction final :

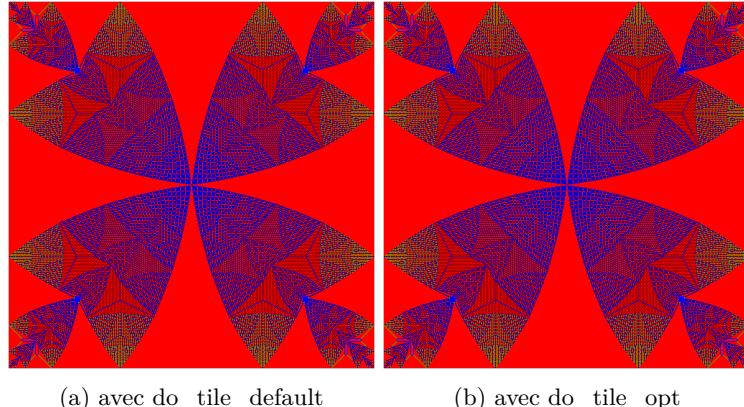
```

26 int asandPile_do_tile_opt(int x, int y, int width, int height)
27 {
28     int change = 0;
29
30     for (int i = y; i < y + height; i++)
31         for (int j = x; j < x + width; j++)
32     {
33         int result = atable(i, j);
34         if (result >= 4)
35         {
36             result /= 4;
37             atable(i, j - 1) += result;
38             atable(i, j + 1) += result;
39             atable(i - 1, j) += result;
40             atable(i + 1, j) += result;
41             atable(i, j) %= 4;
42             change = 1;
43         }
44     }
45     return change;
46 }
```

---

OK

Nous avons vérifié et on obtient le même résultat et le même nombre d'iterations (34938) avec la version par défaut et la version opt.



(a) avec do\_tile\_default

(b) avec do\_tile\_opt

FIGURE 2 – Verification du résultats pour asandPile

Le gain de performance est de 1.2 car  $\frac{37990}{31405}$

machine ?

## 2 OpenMP implementation of the synchronous version (4.2)

Pour *ssandPile\_compute\_omp()* on a fait en sorte de parcourir chaque grain de sable puis on a ajouté un *pragam omp for* pour que chaque grain de sable soit calculé par un thread.

Le code de la fonction :

---

```

1  unsigned ssandPile_compute_omp(unsigned nb_iter)
2  {
3      for (unsigned it = 1; it <= nb_iter; it++)
4      {
5          int change = 0;
6          #pragma omp parallel for schedule(runtime) reduction(+: change)
7          for (int y = 1; y < DIM-1; y += 1)
8              for (int x = 1; x < DIM-1; x += 1)
9                  change += do_tile(x, y, 1, 1, omp_get_thread_num());
10         swap_tables();
11         if (change == 0)
12             return it;
13     }
14 }
15
16 return 0;
17 }
```

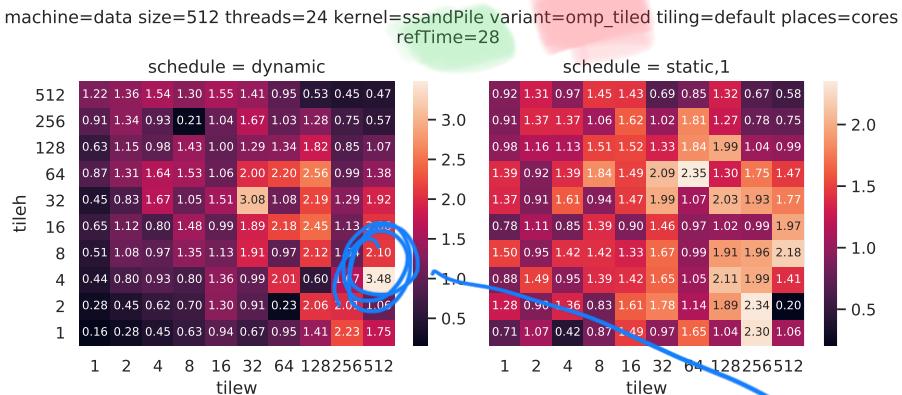
---

Pour *ssandPile\_compute\_omp\_tiled()* on a dupliqué la version tile pour y ajouter un pragam *omp for* avec un *collapse(2)* pour que les tuiles soient calculés par des threads.

durée trop petite  
inexploitable

La heatmap montre que en moyenne, le schedule static est meilleur que le dynamique, cependant on cherche à résoudre un problème précis, donc on va chercher l'accélération maximale. Ici ce sera avec des tuiles de width 512 et de height 4 en schedule dynamique.

cool ?



Pour ssandPile\_compute\_omp\_taskloop() on a dupliqué la version tile pour y ajouter un pragam omp single, pour qu'un thread definisse les taches, et un pragam omp task devant l'appelle de la fonction do\_tile() pour que le calcule des tuiles se trouve dans un thread.

On voit que taskloop avec des tuile de taille 64 et 32 est presque aussi bonne que omp\_tile avec des tuiles de taille 64, 32 et 16. Et que taskloop n'est pas très efficace avec des tuiles de taille 8 ou 16.

expliquer pourquoi

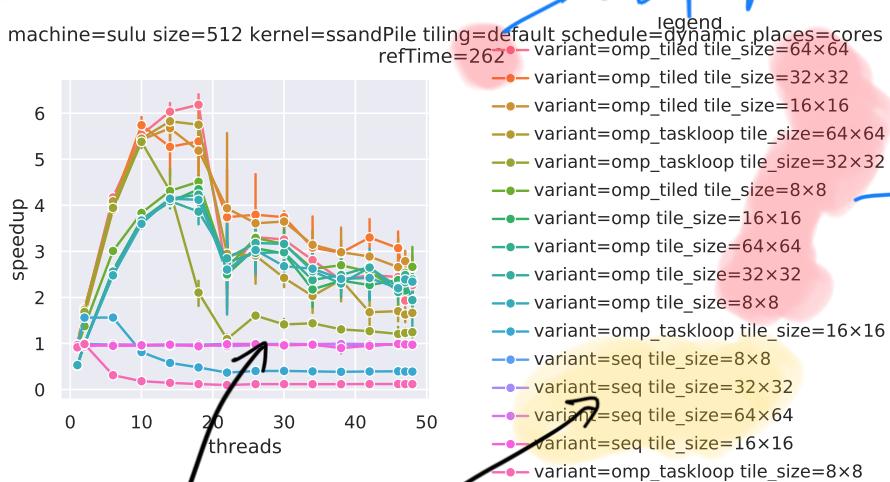


FIGURE 3 – ssandPile comparaison de taskloop avec toutes les variantes

cas inutiles

6

### 3 OpenMP implementation of the asynchronous version (4.3)

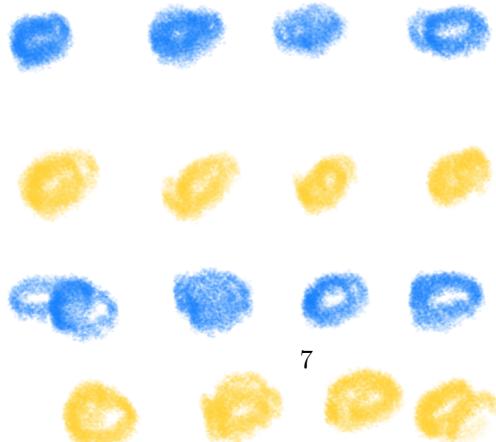
Pour paralléliser avec asandPile\_compute\_omp\_tiled, il a fallu créer 4 nids de boucles afin d'éviter les lectures/écritures concurrentes. On a donc un nid de boucle pour les lignes et colonnes impaires, un pour les lignes impaires et colonnes paires, un pour les lignes paires et colonnes impaires, et un pour les lignes et colonnes paires.

Le code de la fonction final :

```
1  unsigned asandPile_compute_omp_tiled(unsigned nb_iter)
2  {
3      for (unsigned it = 1; it <= nb_iter; it++)
4      {
5          int change = 0;
6
7          //BLEU
8          #pragma omp parallel for schedule(runtime) shared(change)
9          for(int y=0; y<DIM; y+=2*TILE_H)
10         {
11             for (int x = y%(TILE_H*2); x < DIM; x += TILE_W*2)
12             {
13                 int localChange =
14                     do_tile(x + (x == 0), y + (y == 0),
15                             TILE_W - ((x + TILE_W == DIM) + (x == 0)),
16                             TILE_H - ((y + TILE_H == DIM) + (y == 0)), omp_get_thread_num());
17                 if (change == 0 && localChange != 0)
18                 {
19                     #pragma omp critical
20                     change /= localChange;
21                 }
22             }
23         }
24     }
```

Le nid BLEU est répété 4 fois avec des initialisations de x et y qui diffèrent

*pas de collapse  $\Rightarrow$  on peut écouvoyer 2 coeurs*



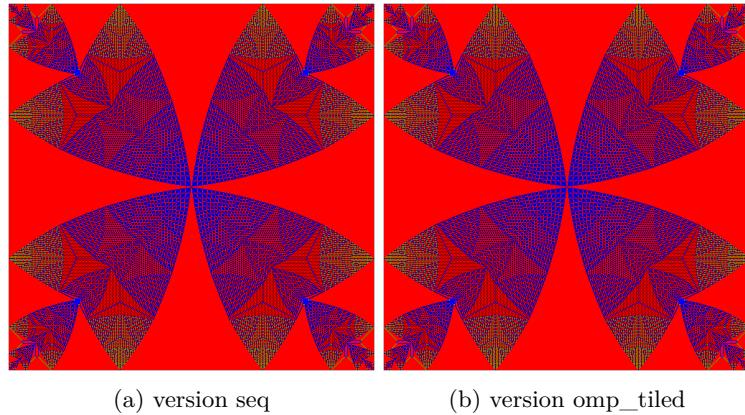


FIGURE 4 – Vérification du résultats pour asandPile

```
machine=kira size=512 threads=24 kernel=asandPile variant=omp_tiled tiling=default iterations=100
places=cores refTime=144
```

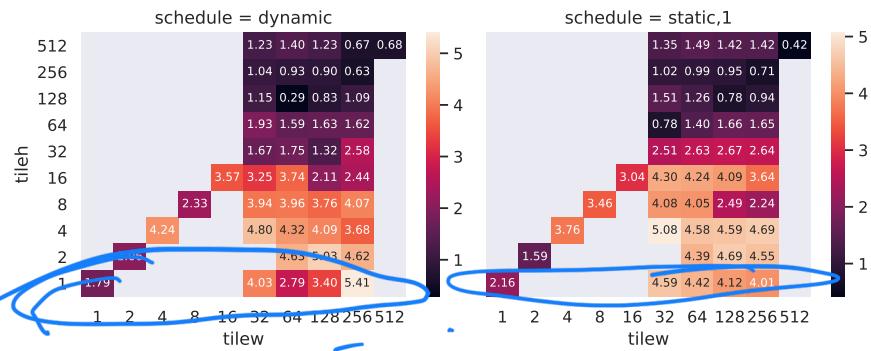


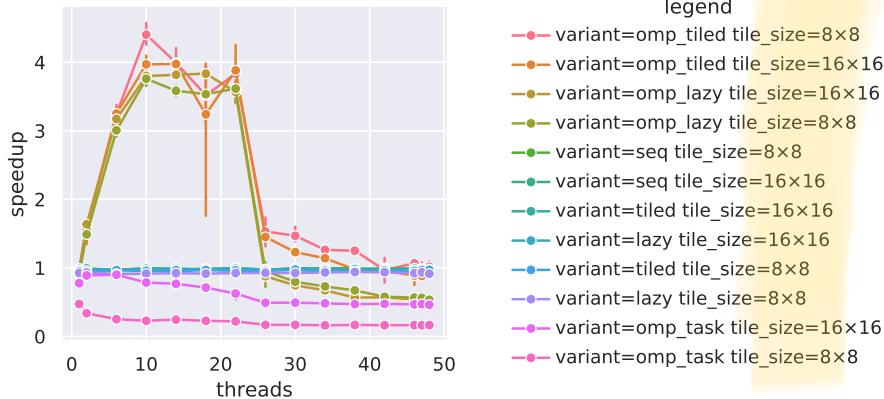
FIGURE 5 – heatmap comparaison omp\_tiled & tiled

Les expériences montrent que pour certaines tailles de tuiles, il n'y a pas de résultat d'accélération, et après vérification, le programme ne fonctionne pas sur ces tailles là. Néanmoins, avec ces résultats, on en déduit que le schedule static,1 avec une width de 32 et une height de 4 est la meilleure option.

Aller à la suite  
n'est pas prévu pour  
utiliser des tuiles de  
hauteur 1 -

où est le cos 9 x 32?

machine=kira size=512 kernel=asandPile tiling=default schedule=dynamic places=cores refTime=143



machine=kira size=512 kernel=asandPile tiling=default schedule=dynamic places=cores refTime=142



FIGURE 6 – asandPile comparaison de toutes les variantes

D'après ces schémas, l'accélération maximale est toujours donnée pour 10 threads peu importe la taille de tuile donnée ( $8 \times 8$ ,  $16 \times 16$ ,  $32 \times 32$ ,  $64 \times 64$ ). On voit également une chute de performances vers 25 threads, c'est parce qu'on était sur kira qui est une machine 24 coeurs.

#### 4 Lazy OpenMP implementations (4.4)

Pour la version asynchrone, nous avons repris la version précédente et ajouté deux tableaux servant successivement de lecture et d'écriture pour tester si les tuiles autour ont été modifiés à l'itération précédente.

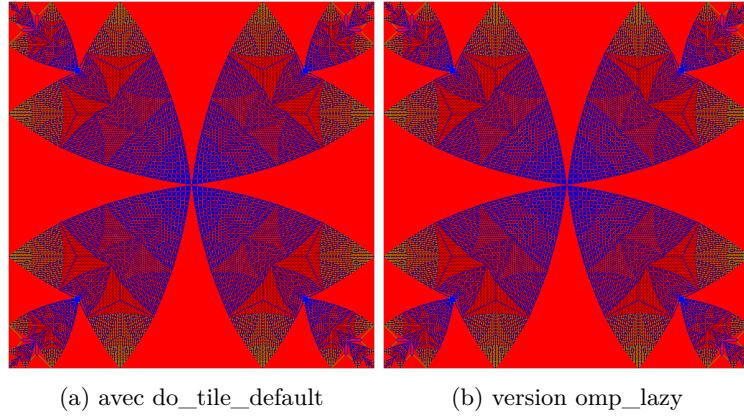


FIGURE 7 – Vérification du résultats pour asandPile

```
machine=kira size=512 threads=24 kernel=asandPile variant=omp_lazy tiling=default iterations=100
places=cores refTime=144
```

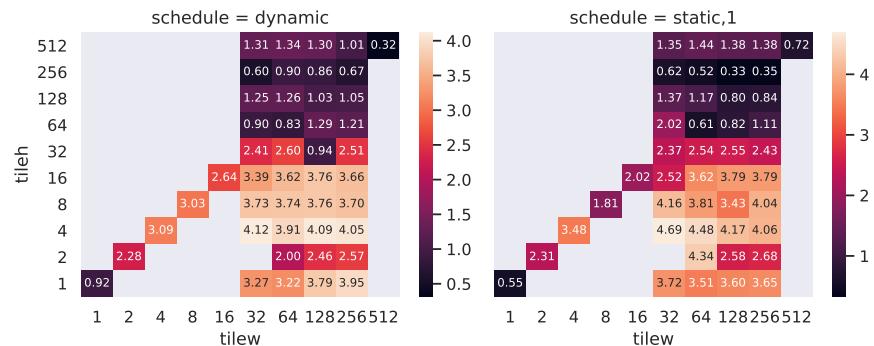


FIGURE 8 – heatmap comparaison omp\_lazy & lazy

*Les expériences montrent que pour certaines tailles de tuiles, il n'y a pas de résultat d'accélération, et après vérification, le programme ne fonctionne pas sur ces tailles là. Néanmoins, avec ces résultats, ont en déduit que le schedule static,1 avec une width de 32 et une height de 4 est la meilleure option.*

## 5 Résultats produit à la fin

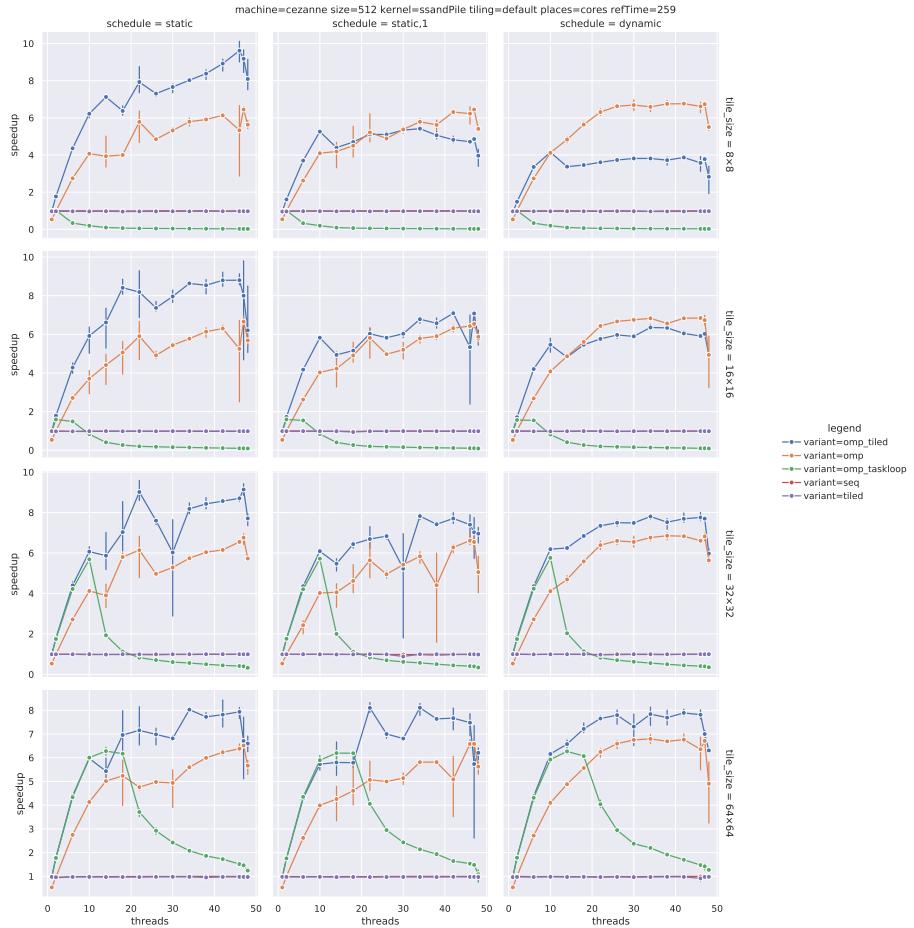


FIGURE 9 – ssand all variants experiments

*Bien dans l'esprit!*

mais il faut :

- sélectionner les expériences
- expliquer les performances

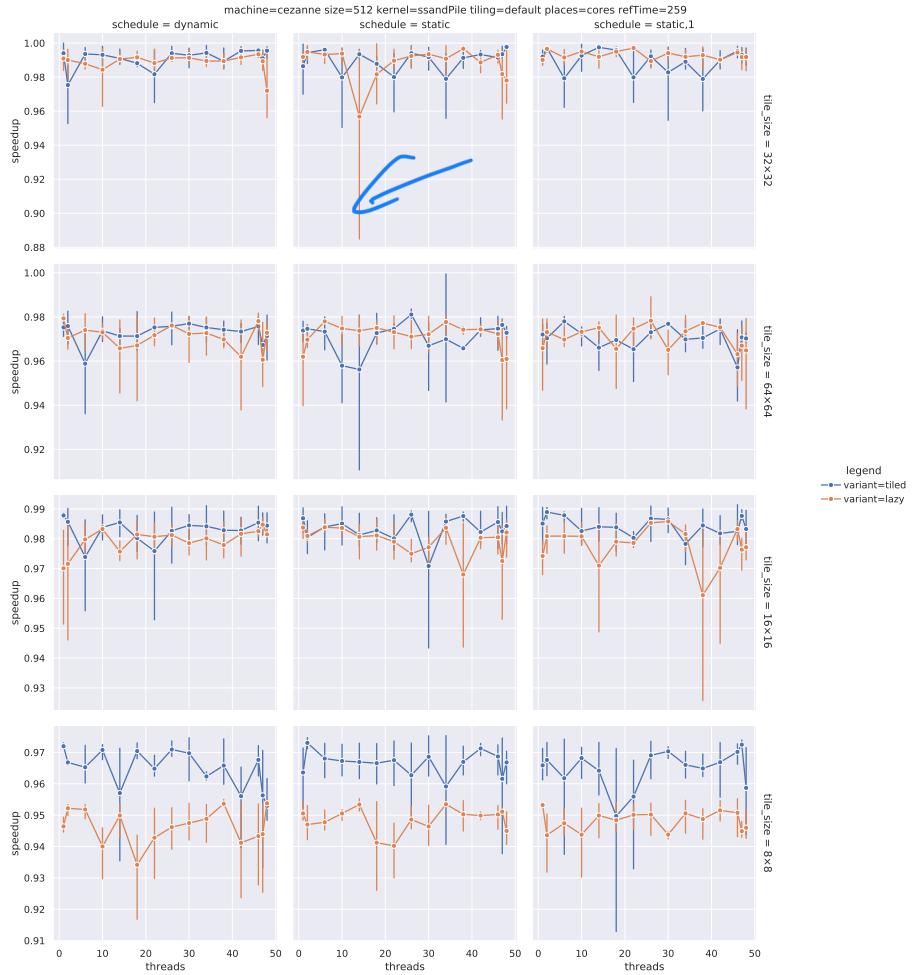


FIGURE 10 – ssand tiled vs lazy experiments

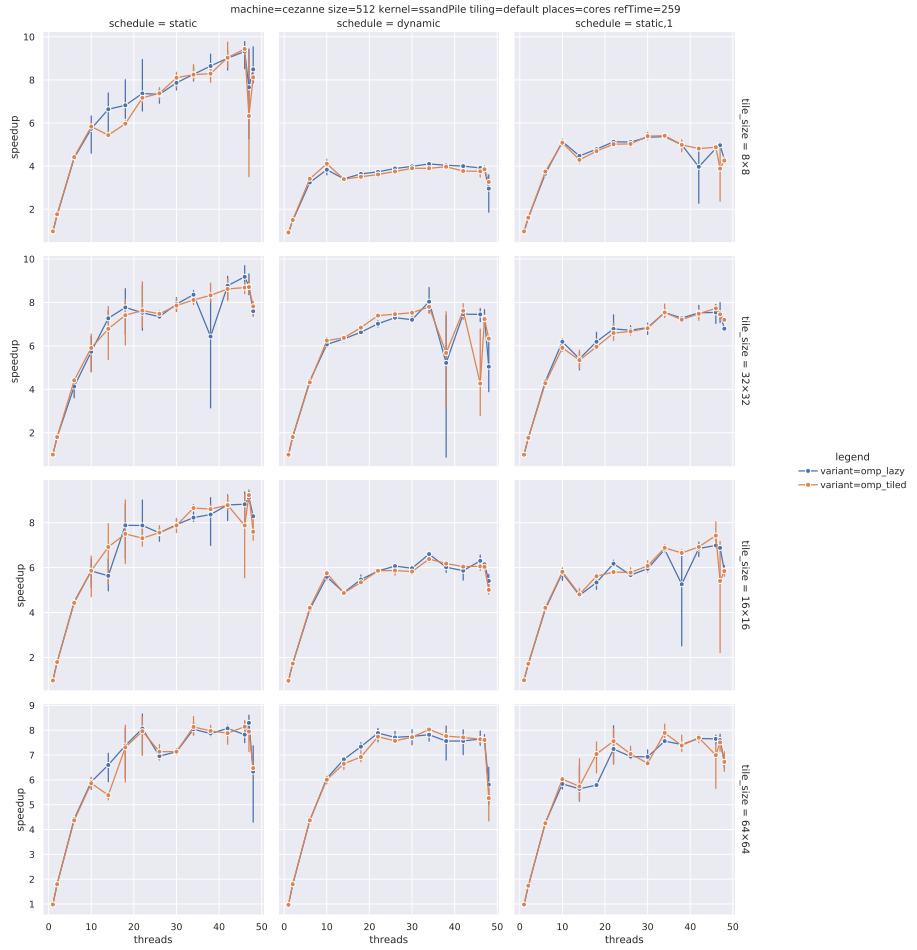


FIGURE 11 – ssand omp\_tiled vs omp\_lazy experiments

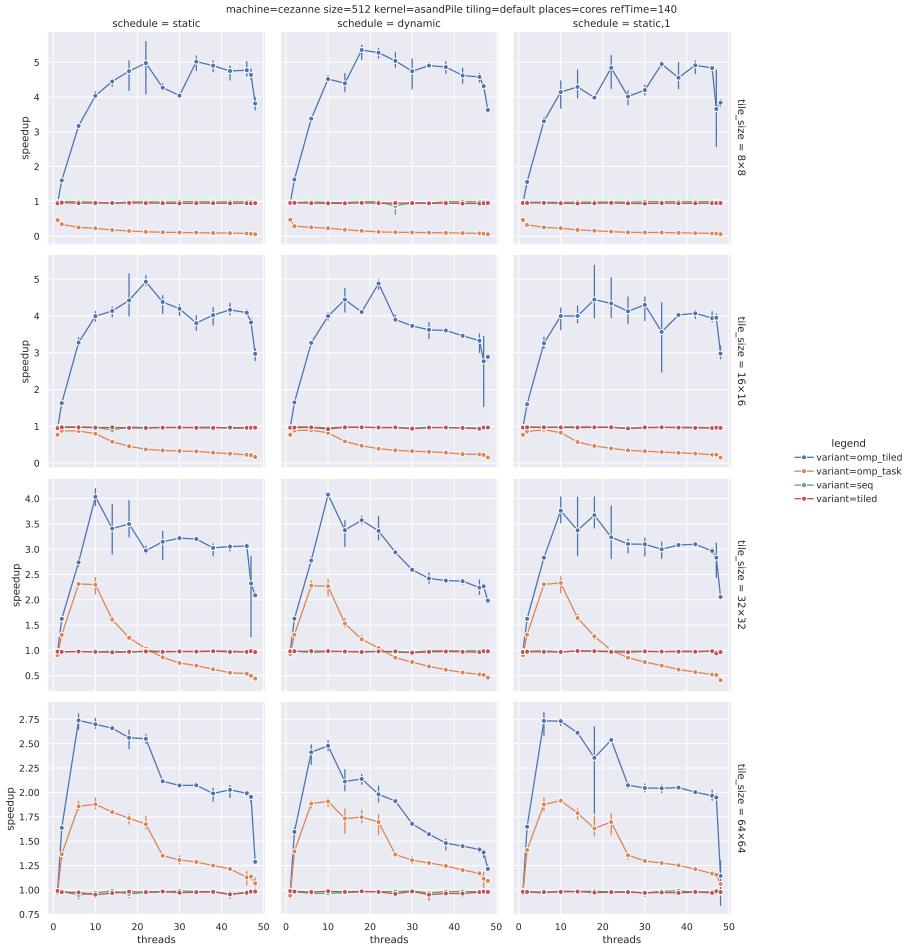


FIGURE 12 – asand all variants experiments

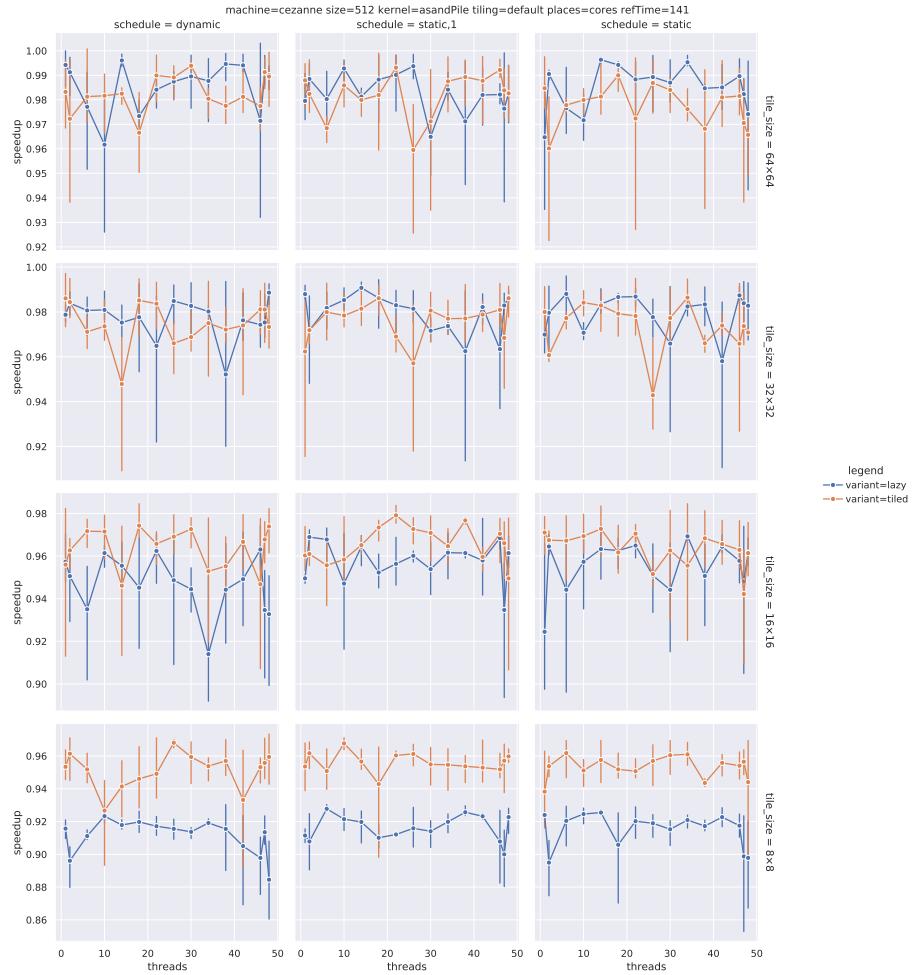


FIGURE 13 – asand tiled vs lazy experiments

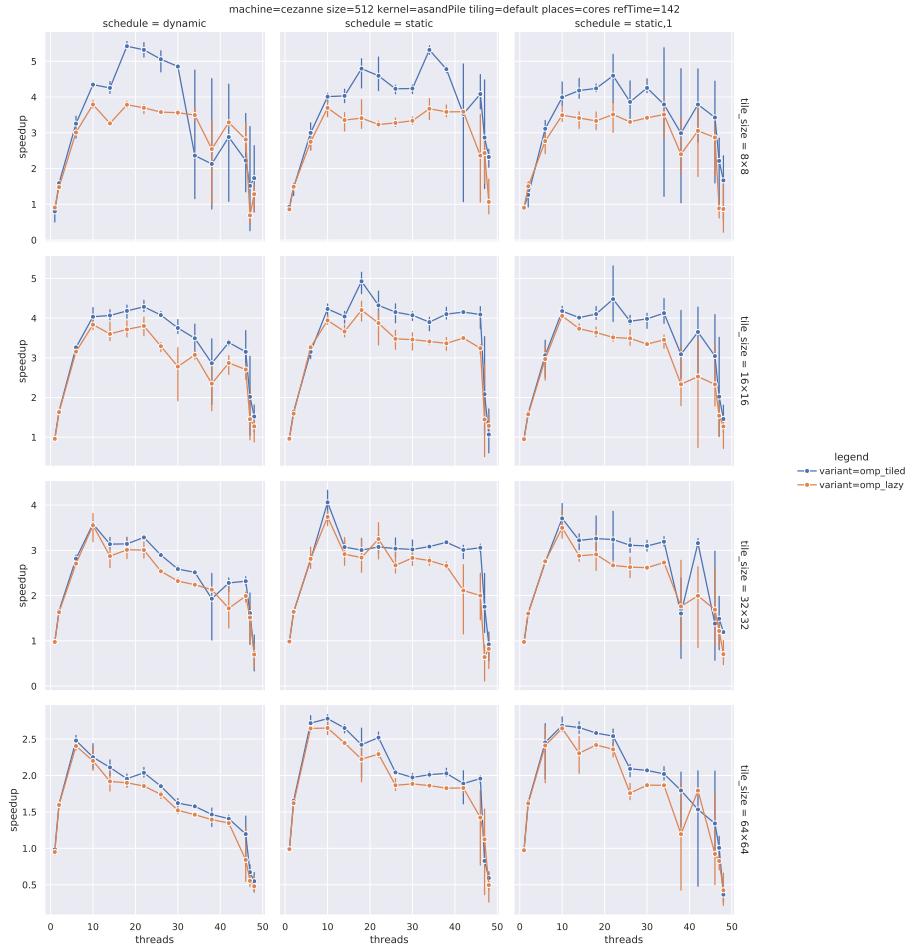


FIGURE 14 – asand omp\_tiled vs omp\_lazy experiments