Projet de Programmation Parallèle

Erwan DUPLAND Guillaume FLORET

Sommaire

Travail effectué

Pour ce projet, nous avons réalisé les 3 versions séquentielles demandées, les versions ‘OpenMP for’ correspondantes, avec à chaque fois les variantes : ‘static’, ‘cyclic’, ‘dynamic’ et ‘collapse’. Les versions ‘task’ pour les versions tuilées et optimisées sont présentes, ainsi que la version OpenCL de la version tuilée. Nous avons également une version MPI, qui se contente de faire calculer la moitié de l’image à un « esclave », et, pour la fonctionnalité avancée, une version MPI + OpenMP, où on se contente de faire traiter les tuiles par des fonctions parallélisées. Cette dernière version n’est, dans l’état actuel, pas très performante.

Machines utilisées

Charge des machines

Nous avons effectué nos mesures et tests sur des machines non chargées. Nous avons vérifié avec la commande 'htop' que nos machines n'étaient pas utilisées par quelqu'un d'autre et qu'il n'y avait pas de tâche en arrière-plan susceptibles d'altérer les performances.

Configuration des machines

Les mesures de temps ont été effectuées sur la machine « Kirk » de la salle 203 du CREMI. Cette machine de 12 cœurs est dotée de processeurs Xeon Gold 5118 64 Go et de carte graphique RTX 2070 (8Go). Les programmes ont été exécutés sur Linux.

Conditions des expériences

Toutes les mesures de temps ont été effectuées sur 100 itérations, sans affichage. Les accélérations des versions OpenMP ont été tracées graphiquement pour évaluer l'impact du nombre de threads utilisés. Pour les autres versions (séquentielles, OpenCL et MPI), il s'agira de tableaux de temps.

Versions séquentielles

Nous avons implémenté les trois versions séquentielles demandées : ‘seq_base’, ‘seq_tiled’ et ‘seq_tiled_opt’. Ces fonctions (ainsi que toutes les autres fonctions qui suivront) utilisent une version modifiée de ‘compute_new_state’ avec un nombre réduit de sauts conditionnels.

```
static int compute_new_state(int y, int x)
{
    unsigned n = 0;
    unsigned change = 0;

    n += (cur_img(y - 1, x - 1) != 0);
    n += (cur_img(y - 1, x) != 0);
    n += (cur_img(y - 1, x + 1) != 0);
    n += (cur_img(y, x - 1) != 0);
    n += (cur_img(y, x + 1) != 0);
    n += (cur_img(y + 1, x - 1) != 0);
    n += (cur_img(y + 1, x) != 0);
    n += (cur_img(y + 1, x + 1) != 0);

    // On laisse un 'if' pour éviter 2 appels à cur_img
    if (cur_img(y, x) != 0){
        n = (n == 2 || n == 3) * 0xFFFF00FF;
    }else{
        n = (n == 3) * 0xFFFF00FF;
    }

    change = (cur_img(y, x) != n);

    next_img(y, x) = n;

    return change;
}
```

La version optimisée fonctionne avec un tableau booléen mémorisant le changement (ou non) des tuiles lors de la dernière itération. On commence par calculer une première itération, puis, pour toutes les suivantes, avant de recalculer une tuile, on vérifiera qu'elle ou ses voisins ont subi des changements récents. Les bords de l'image sont traités séparément.

```
// true si la tuile a été modifiée
    bool tab[GRAIN][GRAIN];
    for (int i = 0; i < GRAIN; i++)
        for (int j = 0 ; j < GRAIN ; j++)
            tab[i][j] = false;

    tranche = DIM / GRAIN;
    unsigned change = 0;


    // 1er tour pour initialiser les tuiles modifiées dans le tableau

    change |= traiter_tuile_seq_tiled_opt(1, 1, tranche, DIM-2);
    change |= traiter_tuile_seq_tiled_opt(DIM-tranche, 1, DIM-2, DIM -2);
    change |= traiter_tuile_seq_tiled_opt(tranche+1, 1, DIM-tranche-1,
tranche);
    change |= traiter_tuile_seq_tiled_opt(tranche+1, DIM-tranche, DIM-tranche-
1, DIM -2);

    for (int i = 1 ; i < GRAIN-1; i++){
        for (int j = 1 ; j < GRAIN-1; j++){
            change |= traiter_tuile_seq_tiled_opt(
                i * tranche,
                j * tranche,
                (i + 1) * tranche - 1,
                (j + 1) * tranche - 1
            );

            tab[i][j] = change;
        }
    }
```

Mesures

Pour ces versions séquentielles, nous avons effectué des mesures pour les configurations ‘random’ et ‘guns’, en faisant varier le grain entre 16 et 32 (pour les 2 versions tuilées), ainsi que la taille du monde : 512 ou 4096.

Résultats & Interprétations

Configuration	Grain	Taille	Version	Temps
guns	16	512	seq	159
			seq_base	133
			seq_tiled	208
			seq_tiled_opt	52
		4096	seq	10060
			seq_base	8485
			seq_tiled	14500
			seq_tiled_opt	3785
	32	512	seq	159
			seq_base	133
			seq_tiled	213
			seq_tiled_opt	37
		4096	seq	10060
			seq_base	8485
			seq_tiled	15100
			seq_tiled_opt	1995
random	16	512	seq	198
			seq_base	174
			seq_tiled	252
			seq_tiled_opt	70
		4096	seq	12388
			seq_base	10780
			seq_tiled	16500
			seq_tiled_opt	4100
	32	512	seq	198
			seq_base	174
			seq_tiled	258
			seq_tiled_opt	41
		4096	seq	12388
			seq_base	10780
			seq_tiled	17500
			seq_tiled_opt	2360

La première remarque que l'on puisse faire concernant ces résultats est que les versions de base et optimisée sont plus rapides que la version d'origine « naïve », mais que la version tuilée est plus lente. On peut expliquer ce phénomène par le surcoût entraîné par le partage du monde en tuiles. La version tuilée optimisée fonctionne sur le même principe, mais le gain de temps dû à l'optimisation compense cette perte.

Ensuite, on peut voir que les temps pris par les différentes fonctions sont systématiquement plus longs lorsque la configuration du jeu est 'random'. Comparé à la version 'guns', il y a plus de calculs à effectuer, et qui ne sont pas aussi prévisibles.

Enfin, pour les versions tuilées, on peut voir qu'un grain plus grand améliore les performances de la version optimisée, mais diminue celles de la version tuilée. Cela est dû au fait que la version optimisée va trouver les tuiles qui n'ont pas besoin d'être recalculées. Un grain plus élevé divise l'image en un plus grand nombre de tuiles. De cette manière, plus de tuiles seront trouvées comme inchangées, et ne seront pas recalculées.

Versions de base

Nous avons parallélisé toutes les versions de bases suivant le même schéma :

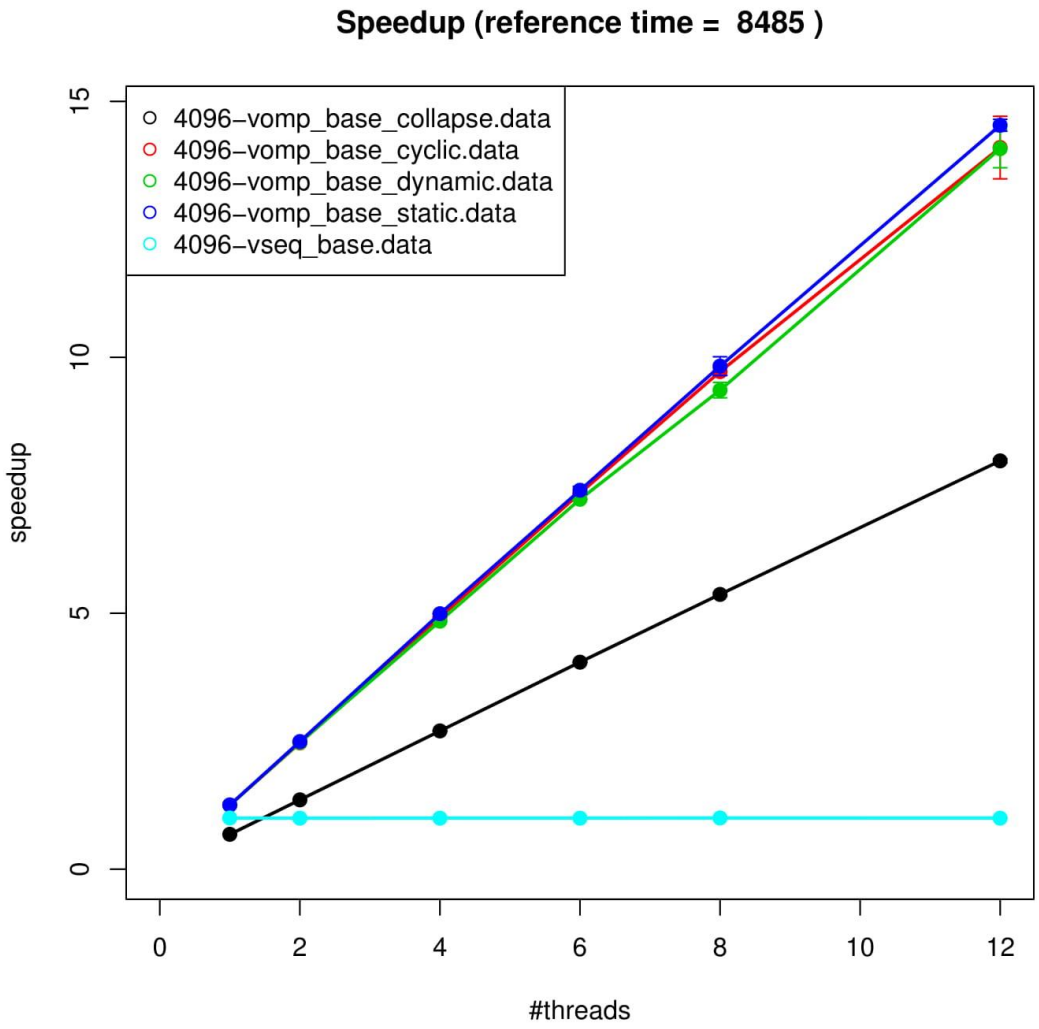
```
#pragma omp parallel for schedule (static) reduction(|:change)
for (int i = 1; i < DIM-1; i++){
       for (int j = 1; j < DIM-1; j++){
            change |= compute_new_state(i, j);
       }
}
```

On parallélise la première boucle avec un ‘for schedule()’ et une réduction. Nous avons implémenté des versions statique, cyclique, et dynamique, ainsi qu’une version collapse pour la comparaison.

Mesures

Nous avons comparé toutes les variantes de la version de base (excepté la version MPI). Notre temps de référence est donc celui pris par la version ‘seq_base’, et est comparé aux versions ‘OpenMP for’. Les speedups obtenus sont relativement proches pour des mondes de taille 512 ou 4096. Nous avons donc réalisé 2 graphes comparant nos différentes versions : dans une configuration ‘guns’ et dans une configuration ‘random’, avec des tailles de 4096.

Résultats & Interprétations

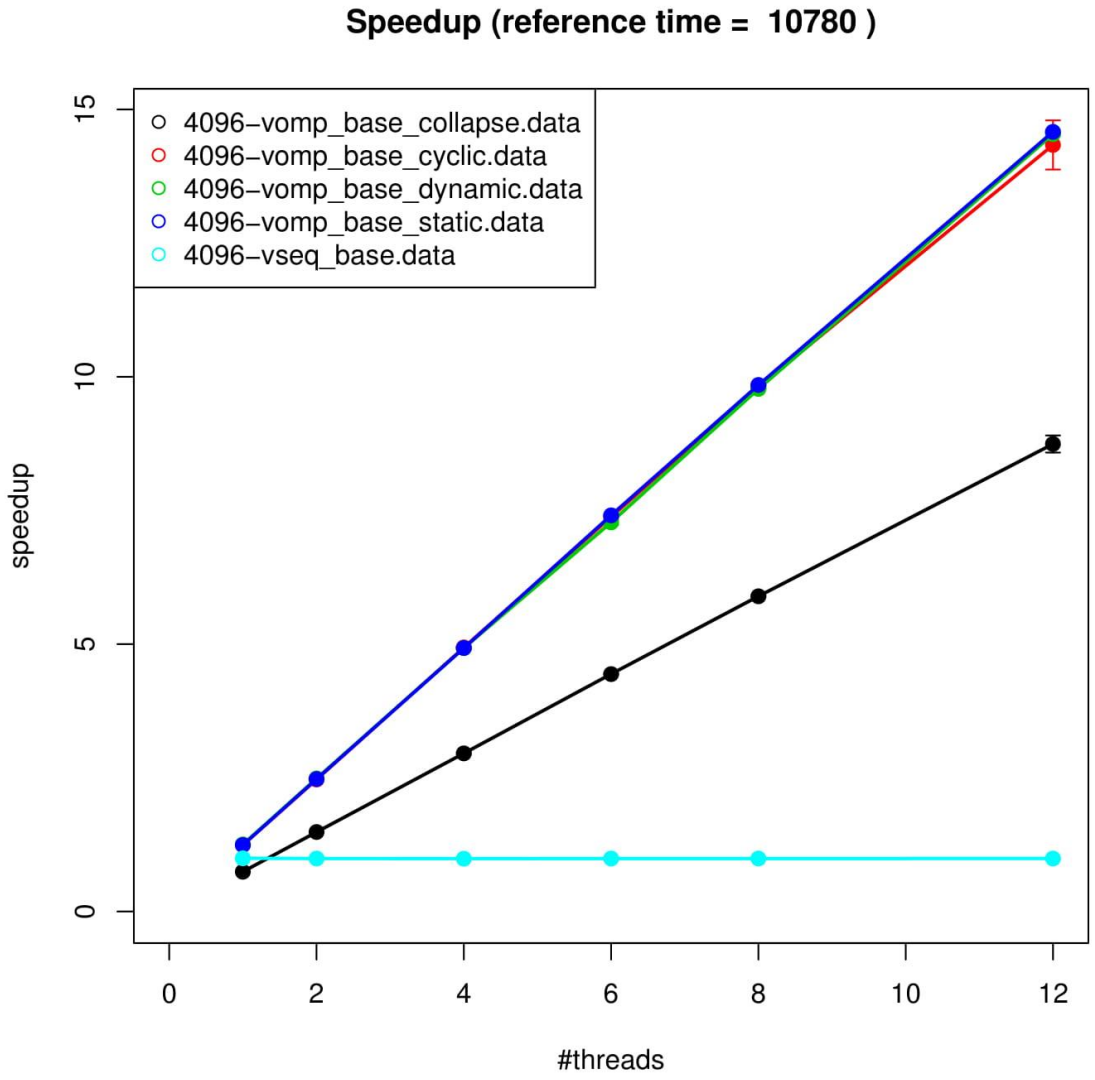


La première observation que l'on puisse faire est que le speedup est supérieur au nombre de cœurs. Nous pensons que cela est dû à certains effets de caches favorables, en effet, nous utilisons dans notre code une réduction, sur la variable 'change', qui sert à stocker le résultat de la fonction 'compute_new_state'. De ce fait, cette variable est mise dans le cache lorsqu'elle est accédée pour la première fois par un thread, puis elle y reste, rendant son accès plus rapide. Comme la « mise à jour » de sa valeur (qui est un OU binaire) est faite à la fin, on n'a pas de problèmes d'accès et de barrières.

On peut constater sur ce graphe que les trois versions 'static', 'cyclic' et 'dynamic' présentent des accélérations très proches. La version dynamique est légèrement inférieure. Cela peut s'expliquer par le fait que le calcul effectué par notre fonction est régulier, et que donc le surcoût de la répartition des tâches ne permet pas de compenser totalement le temps gagné.

On remarque également que la version collapse est beaucoup plus lente que les autres versions. Ceci peut s'expliquer par le fonctionnement de la directive collapse, qui, en « transformant »

nos 2 boucles for en une seule, augmente le nombre d'itérations à partager entre chaque thread. La répartition entraîne alors un surcoût trop important, qui diminue alors l'accélération.



Pour une configuration de jeu aléatoire, on remarque que les accélérations des différentes méthodes sont très similaires à celle obtenues pour une configuration 'guns'. Nos fonctions de base parcourent le monde suivant 2 boucles for et traitent donc indifféremment les différentes parties de l'image. On peut donc tirer les mêmes conclusions que précédemment.

Versions tuilées

Les versions tuilées ont été parallélisées selon le même schéma que pour les versions séquentielles, la seule différence étant que les boucles parallélisées sont celles des fonctions auxiliaires 'traiter_tuile' :

```
#pragma omp parallel for schedule(static) reduction(|:change)
    for (int i = i_d; i <= i_f; i++)
        for (int j = j_d; j <= j_f; j++)
            change |= compute_new_state(i, j);
```

La version 'task' est gérée différemment, on définit comme tâche chaque appel à la fonction 'traiter_tuile', à l'intérieur de la seconde boucle 'for' :

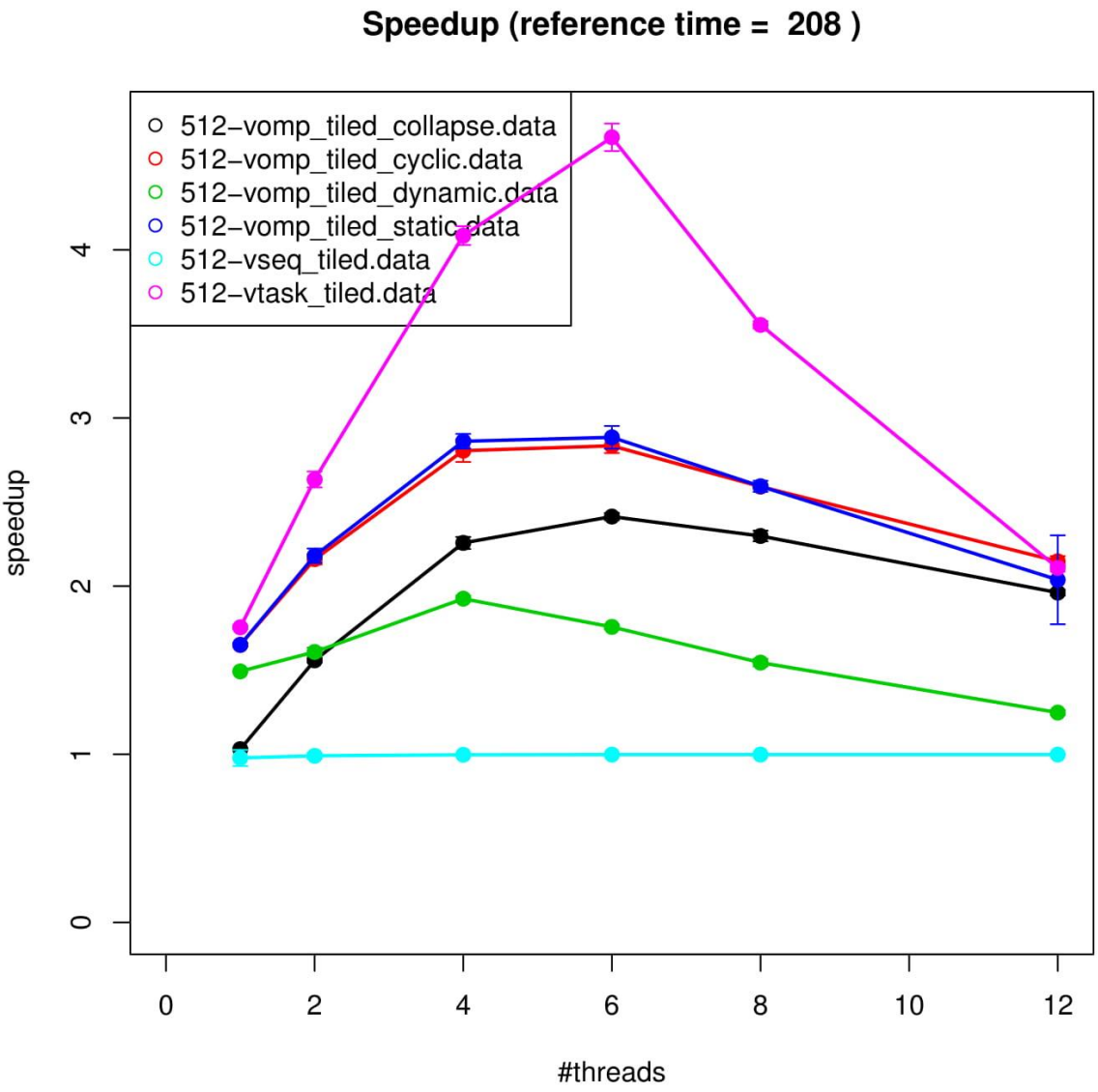
```
for (int i = 0; i < GRAIN; i++){
      #pragma omp parallel
      #pragma omp single
      for (int j = 0; j < GRAIN; j++){
           #pragma omp task
           change |= traiter_tuile_seq_tiled(
                    (i == 0) + i * tranche,
                    (j == 0) + j * tranche,
                    (i + 1) * tranche - 1 - (i == GRAIN-1),
                    (j + 1) * tranche - 1 - (j == GRAIN-1));
      }
      #pragma omp taskwait
 }
```

Mesures

Nous avons comparé les variantes de la version tuilée (excepté la version OpenCL). Les graphes d'accélération produits sont quasiment identiques pour des configurations 'guns' et 'random'. Nous avons donc choisi de faire varier la taille et le grain sur une configuration 'guns'.

Résultats & Interprétations

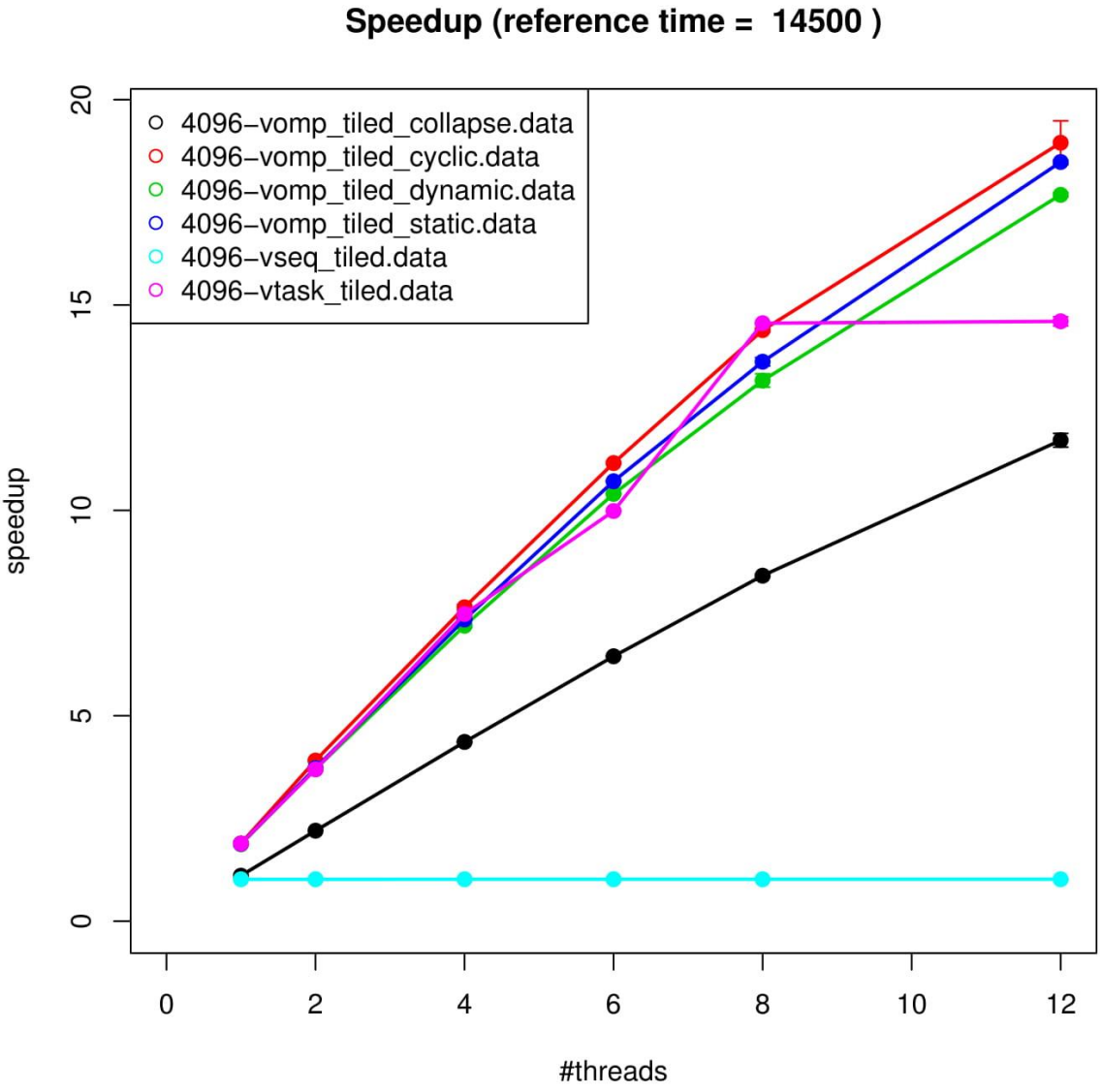
Grain 16



La première constatation que l'on peut faire est que les courbes de speedup cessent d'augmenter avant d'arriver au maximum de threads utilisés. Ceci signifie que sur une image de petite taille (comme 512), et en divisant le travail en tuiles, il n'y a pas assez de travail à répartir pour compenser le surcoût de la parallélisation.

Ensuite, on peut voir que la version 'task' est beaucoup plus efficace que les autres versions. Cela est dû au fait que l'utilisation des tâches s'accorde bien avec le traitement d'une tuile : chaque tuile correspond à une tâche, et, statistiquement, chaque tuile représente la même quantité de travail, ce qui assure une bonne répartition et donc un meilleur gain de temps.

Enfin, on remarque que la version dynamique est plus lente que les autres. Ceci est justement dû au fait que le travail à effectuer est similaire pour chaque tuile, le temps pris pour la répartition dynamique « retarde » donc cette version par rapport aux autres, car il n'y a pas de réel gain de temps.

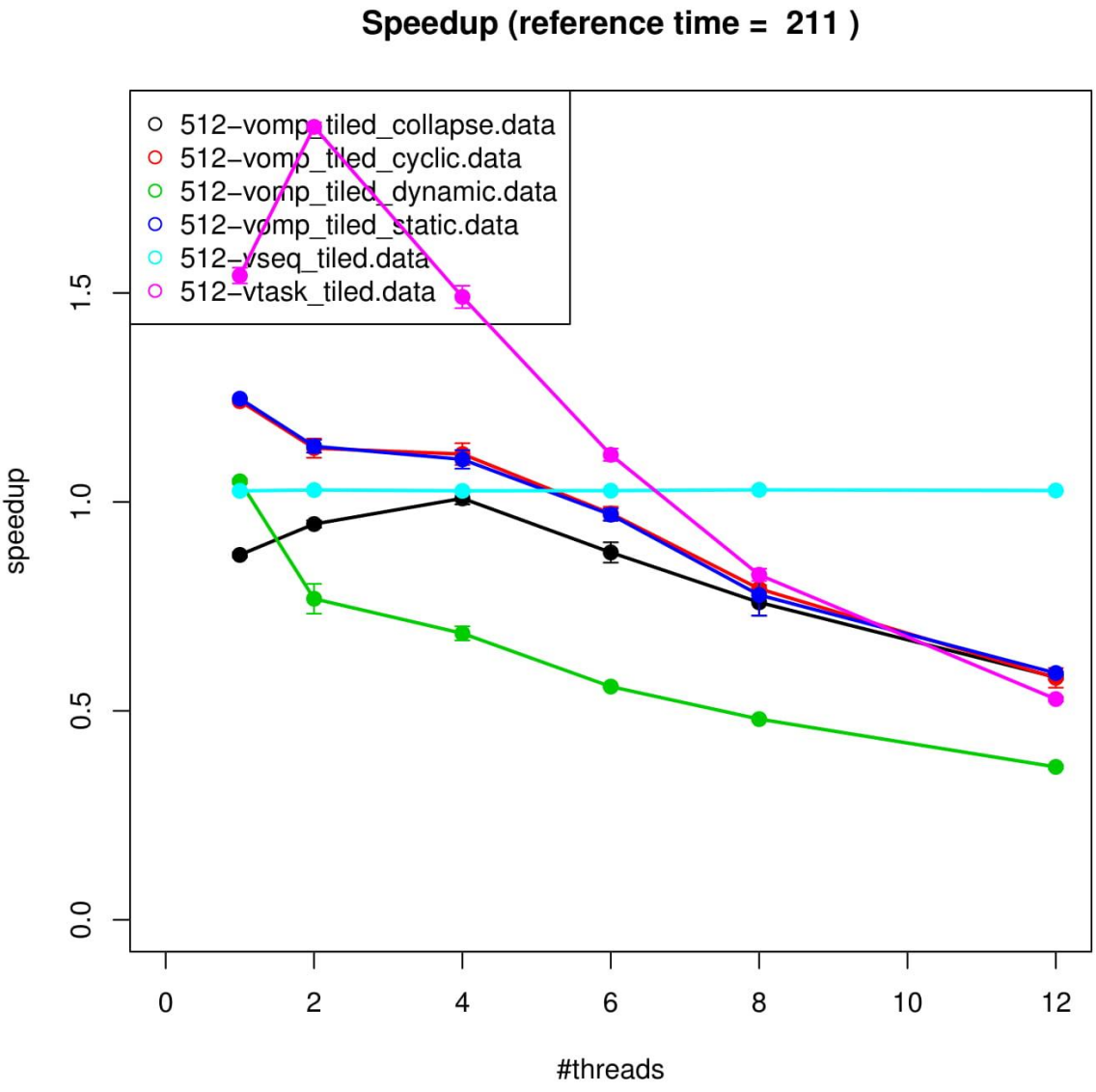


Pour un monde de taille 4096, les différences entre les versions parallèles sont moins nettes, car la quantité de travail à effectuer est suffisante pour justifier la parallélisation. Le surcoût de cette parallélisation est négligeable en comparaison du temps mis par l'exécution de la fonction.

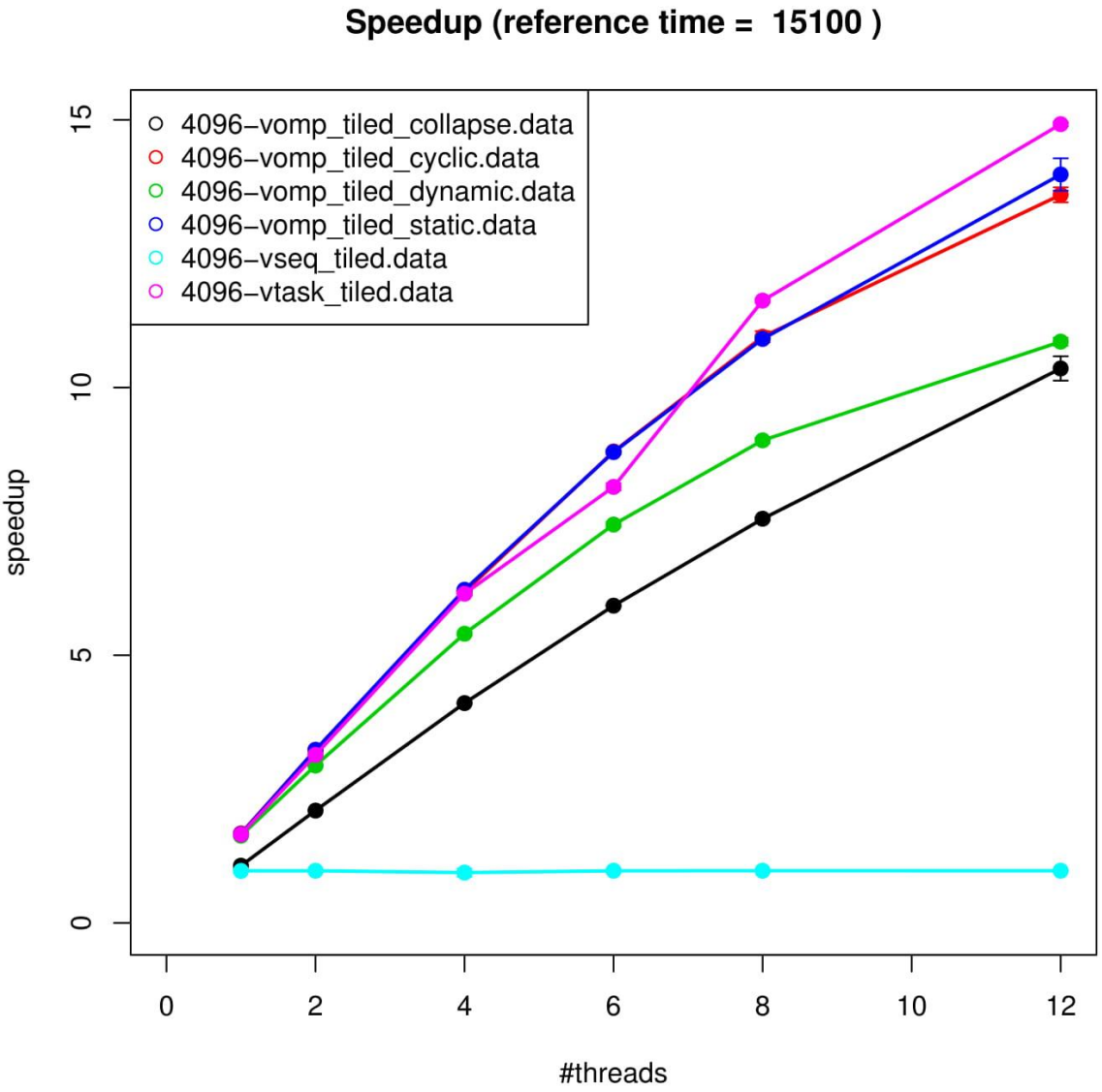
La seconde observation qui puisse être faite est la stagnation de l'accélération de la version 'task' entre 8 et 12 threads. On pourrait expliquer ceci par un problème de cache. Sur une image de taille 4096x4096, et un grain de 16, une ligne a 16 tuiles. Chaque thread exécute 2 tuiles sur cette ligne.

Mais si l'on tente de calculer des tuiles supplémentaires, ces tuiles seront sur une autre ligne, pouvant ainsi poser des problèmes de défauts de cache, expliquant le ralentissement du speedup.

Grain 32



On peut voir sur ce graphe que le phénomène précédent (image de taille 512 et grain = 16) est accentué lorsque le grain augmente. On ajoute un surcoût en créant plus de tuiles, de plus petite taille, mais le travail à effectuer n'en est pas plus important sur une image de cette taille. De ce fait ces versions en viennent à mettre plus de temps que la version séquentielle tuilée.



Avec un grain de 32 et un monde de taille 4096, on a plus les inconvénients précédents, il y a suffisamment de travail parallélisable à effectuer pour justifier la parallélisation. Les versions ‘collapse’ et dynamique restent plus lentes, toujours pour les mêmes raisons.

Versions optimisées

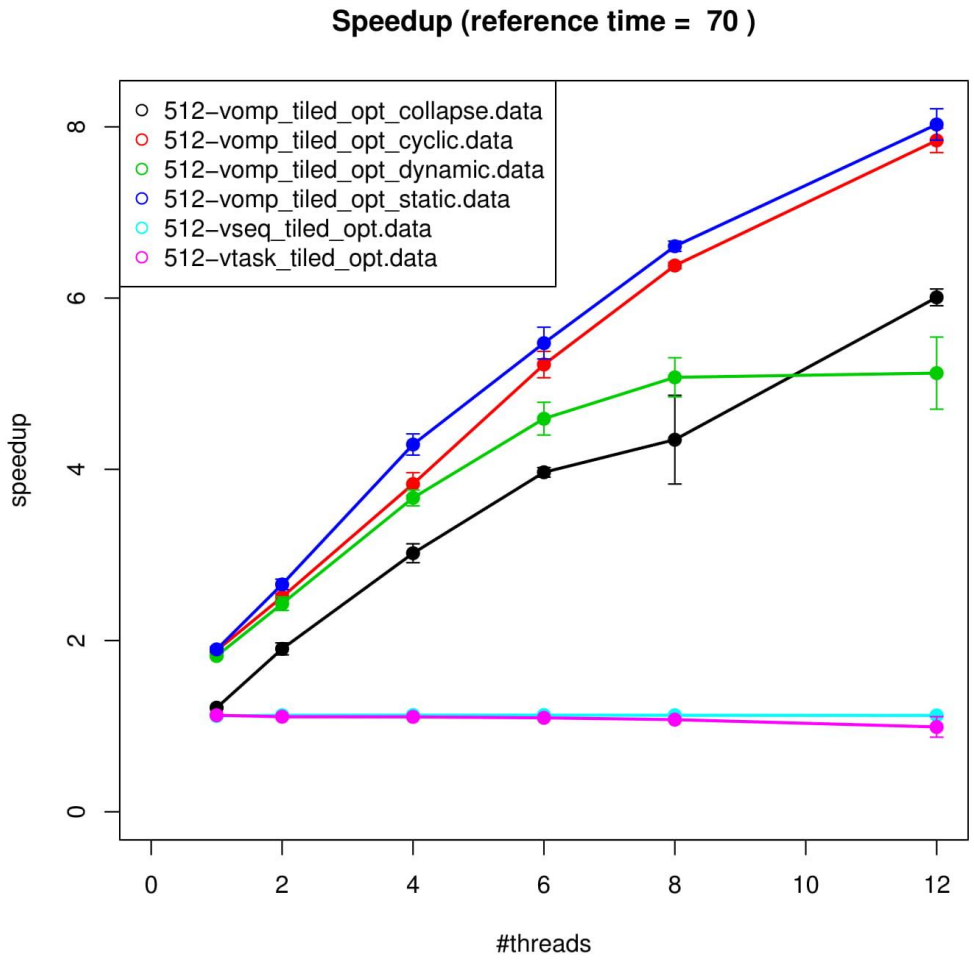
Les versions optimisées ont été parallélisées de manière similaire aux versions tuilées.

Mesures

Nous avons mesuré les différentes variantes des versions tuilées optimisées. Encore une fois, les graphes produits sont similaires pour les configurations ‘random’ et ‘guns’ (les temps sont légèrement plus lents pour ‘random’ mais l’accélération est identique). Nous allons donc nous baser sur une configuration ‘random’ pour examiner les différences avec des grains de 16 et 32, et des tailles de 512 et 4096.

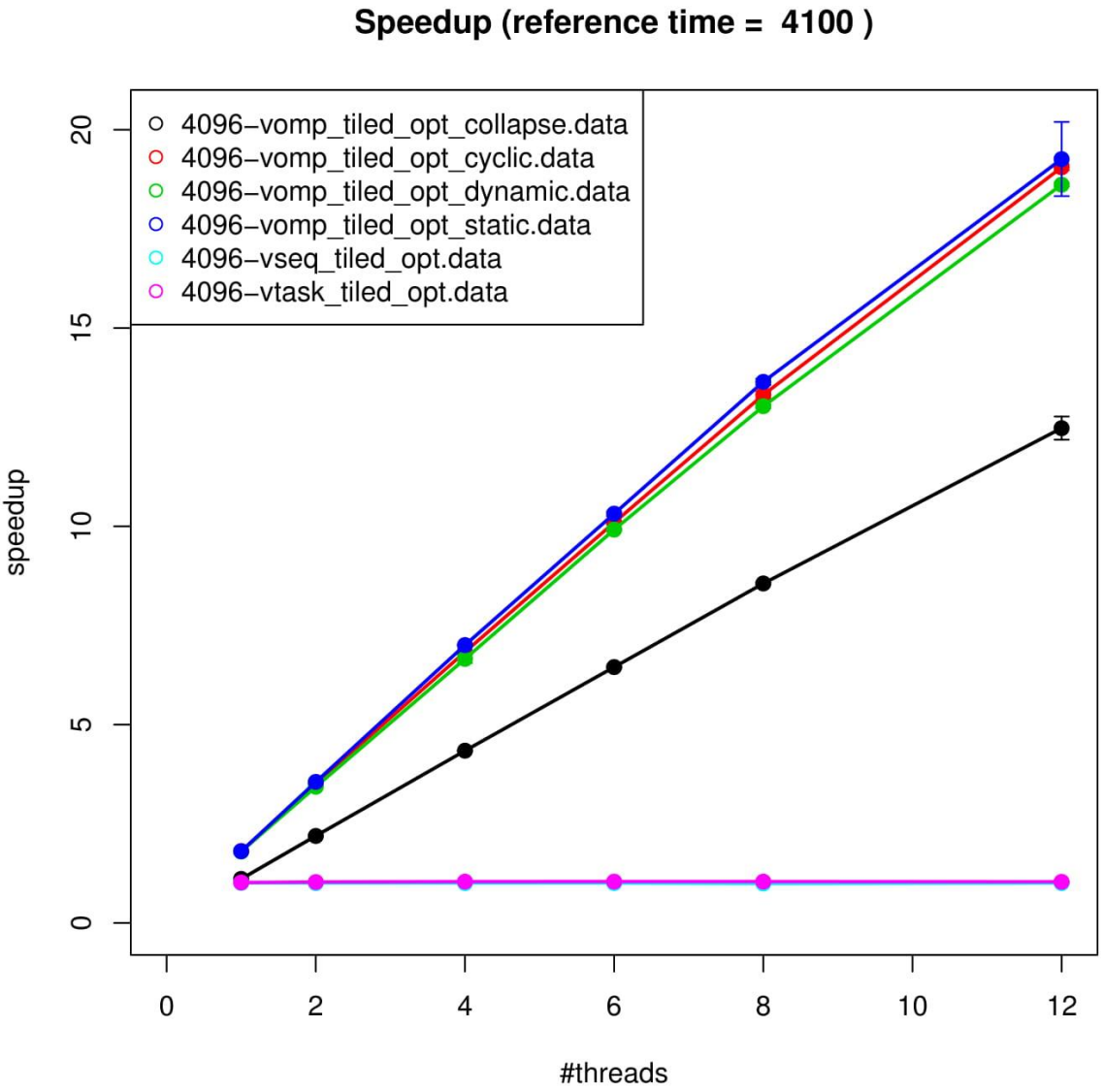
Résultats & Interprétations

Grain 16

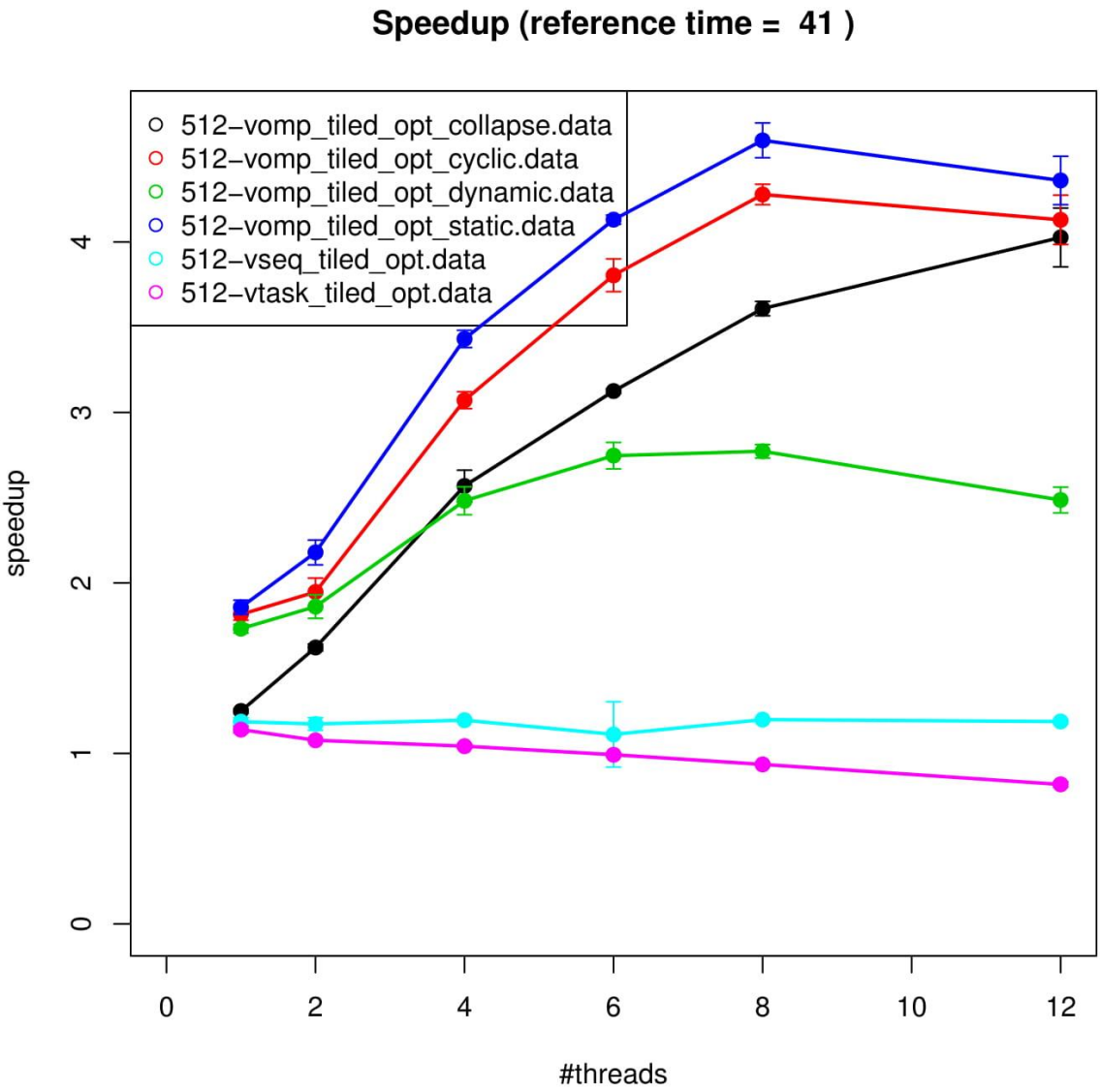


Le premier constat que l'on puisse faire est que la version tuilée affiche de très mauvais temps, identiques voire inférieurs à la version séquentielle. Cela est probablement dû à notre implémentation qui doit comporter des erreurs de conception.

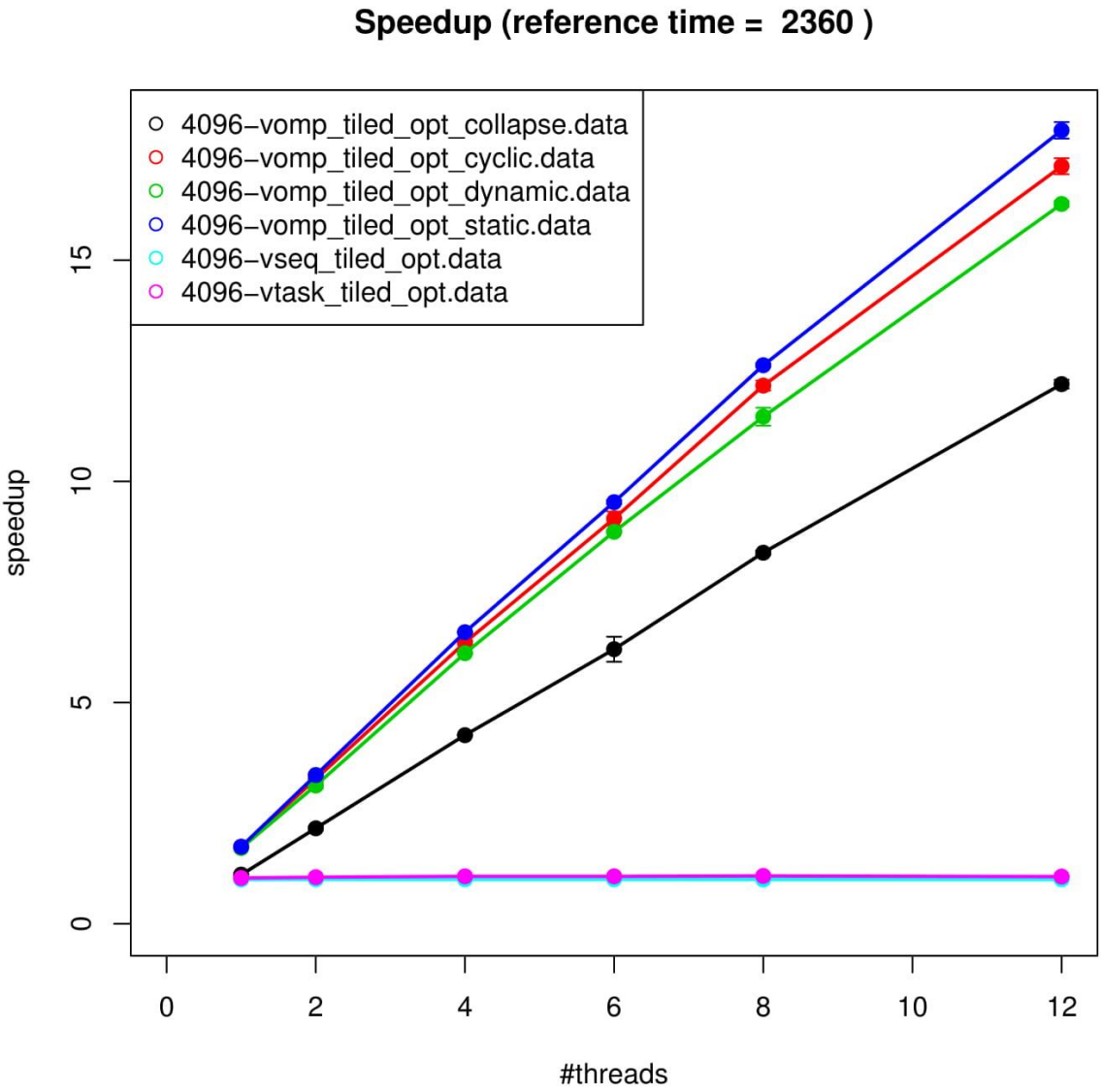
Ensuite, la version cyclique est légèrement plus lente que la version statique. Cela est dû au fait que le travail à faire est régulier, et que donc le surcoût de la répartition cyclique n'est pas justifié. De même, la version dynamique est ici aussi plus lente, et la répartition sur un plus grand nombre de threads n'offre pas d'accélération plus importante.



Pour un monde de taille 4096, les courbes d'accélération s'uniformisent car le coût de la parallélisation est négligeable en comparaison du temps total mis par l'exécution des fonctions.



On observe ici des résultats similaires à ceux obtenus pour un grain de 16 (et une image de même taille = 512), si ce n'est que le speedup est moins important (il atteignait pratiquement 8 avant, et dépasse à peine les 4 ici, pour 12 threads). On peut expliquer cela par les mêmes justifications que pour les versions tuilées non-optimisées : le travail à effectuer n'est pas assez important pour compenser la parallélisation, et la création de tuiles plus petites et plus nombreuses ajoute un surcoût en plus.



De la même manière que précédemment, passer sur un monde de taille 4096 résout la majorité des problèmes énoncés précédemment. Le travail à effectuer est régulier, la répartition statique offre donc une meilleure accélération que la répartition cyclique, qui va plus vite que la version dynamique. La version collapse reste plus lente.

Version OpenCL

L'initialisation du noyau se fait dans le fichier 'vie.c'. Ensuite, dans le fichier 'vie.cl', on « réécrit » notre code des fonctions 'compute_new_state' et de la fonction tuilée, adapté pour OpenCL :

```
if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1){

        if (haut || bas)
            tile [ yloc + 1][ xloc + 1 - haut + bas ] =
                   in [ y * DIM + x - haut + bas ];

        if (gauche || droite)
            tile [ yloc + 1 - gauche + droite ][ xloc + 1] =
                   in [( y - gauche + droite ) * DIM + x ];

        if ((haut || bas) && (gauche || droite))
            tile [ yloc + 1 - gauche + droite ][ xloc + 1 - haut + bas ] =
                   in [( y - gauche + droite ) * DIM + x - haut + bas ];

        barrier (CLK_LOCAL_MEM_FENCE);

        unsigned n = 0;
        n += (tile[yloc][xloc] != 0);
        n += (tile[yloc][xloc+1] != 0);
        n += (tile[yloc][xloc+2] != 0);
        n += (tile[yloc+1][xloc] != 0);
        n += (tile[yloc+1][xloc+2] != 0);
        n += (tile[yloc+2][xloc] != 0);
        n += (tile[yloc+2][xloc+1] != 0);
        n += (tile[yloc+2][xloc+2] != 0);

        if (tile[yloc+1][xloc+1] != 0)
            result = (n == 2 || n == 3) * 0xFFFF00FF;
        else
            result = (n == 3) * 0xFFFF00FF;

    }

    out [y * DIM + x] = result;
```

Mesures

La version OpenCL ne dépendant pas du nombre de cœurs, nous n'avons pas tracé de courbe mais effectué des mesures de temps. En guise de comparaison, nous avons effectué des mesures sur la version tuilée séquentielle, la version tuilée parallélisée avec 'OpenMP for schedule(cyclic)', et la version tuilée parallélisée avec 'OpenMP task'.

Résultats & Interprétations

Configuration	Grain	Taille	Version	Temps
guns	16	512	tiled	215
			omp_tiled	170
			task_tiled	209
			opencl	1,3
		4096	tiled	14040
			omp_tiled	766
			task_tiled	1146
			opencl	84,5
	32	512	tiled	212
			omp_tiled	600
			task_tiled	775
			opencl	1,3
		4096	tiled	14600
			omp_tiled	1100
			task_tiled	1084
			opencl	84,8
random	16	512	tiled	253
			omp_tiled	169
			task_tiled	208
			opencl	1,3
		4096	tiled	16500
			omp_tiled	1016
			task_tiled	1263
			opencl	84,6
	32	512	tiled	257
			omp_tiled	596
			task_tiled	758
			opencl	1,3
		4096	tiled	16980
			omp_tiled	1333
			task_tiled	1361
			opencl	84,8

On remarque que quelque soit la configuration, la taille, ou le grain, la version OpenCL s'exécutant sur la carte graphique est beaucoup plus rapide que toutes les autres. On remarque une très légère différence lorsque l'on fait varier le grain de 16 à 32 : on met alors un peu plus de temps. Il y a également une très légère augmentation du temps pris par l'exécution lorsque l'on est sur une configuration 'random', mais trop réduite pour avoir une réelle importance à cette échelle.

Version MPI & MPI + OpenMP

Notre version MPI de base fait calculer à un « esclave » la moitié de l'image avant de la récupérer. La version MPI + OpenMP suit le même principe, sauf que l'on effectue le calcul de cette tuile (grosse tuile prenant la moitié de l'image) par une de nos fonctions de calcul de tuile parallélisée avec 'OpenMP for'.

```
for (unsigned it = 1; it <= nb_iter; it++){

        MPI_Scatter (
            &cur_img(0,0),
            tranche * DIM,
            MPI_UNSIGNED,
            &cur_img(0,0),
            tranche * DIM,
            MPI_UNSIGNED,
            0,
            MPI_COMM_WORLD
        );

        traiter_tuile_seq_tiled(1,1, tranche-2, DIM-2);

        MPI_Gather (
            &next_img(0,0),
            tranche * DIM,
            MPI_UNSIGNED,
            &next_img(0,0),
            tranche * DIM,
            MPI_UNSIGNED,
            0,
            MPI_COMM_WORLD
        );

        swap_images();
}
```

Mesures

Nous avons effectué des mesures de temps de nos versions MPI en lançant l'exécution avec 'mpirun -np 2'. Nous avons effectué la comparaison de ces deux versions avec la version séquentielle de base et une version parallèle avec 'OpenMP for schedule(cyclic)'.

Résultats & Interprétations

Configuration	Taille	Version	Temps
guns	512	seq_base	159
		omp_base	17,9
		mpi_base	121
		mpi_omp	3678
	4096	seq_base	8485
		omp_base	586
		mpi_base	5056
		mpi_omp	6034
random	512	seq_base	133
		omp_base	174
		mpi_base	122
		mpi_omp	3729
	4096	seq_base	10780
		omp_base	753
		mpi_base	5919
		mpi_omp	6179

On peut constater que notre version MPI de base est un peu plus rapide que la version séquentielle mais beaucoup plus lente que la version 'OpenMP for'. Cela peut s'expliquer par le fait que notre implémentation est assez rudimentaire et ne comporte pas d'optimisations particulière. De même, notre version OpenMP + MPI est encore plus lente (bien que toujours un peu plus rapide que la version séquentielle), notamment sur des mondes de taille 512. Cela peut se justifier par le temps pris pour les transferts de données, qui prennent du temps, et ajoutent donc un surcoût à l'exécution, surcoût plus visible sur de petites images.

Effort de programmation *vs* Performances obtenues

Les versions séquentielles montrent qu'un effort d'optimisation supplémentaire peut très vite être bénéfique et offrir un gain de temps non négligeable. En effet, notre version optimisée repose sur un principe trivial, qui est de noter les parties de l'image qui changent à chaque fois et de ne les traiter que si elles ou leurs voisins ont été changées à l'itération précédente. Le gain de temps obtenu est alors très important.

Pour les versions OpenMP, il est assez facile d'obtenir de bons gains de temps, le tout étant de déterminer laquelle des versions OpenMP offre la meilleure accélération. La mise en œuvre n'est pas extrêmement compliquée et justifie donc la parallélisation d'un programme.

Pour la version OpenCL, cela demande un peu plus de réflexion et de travail pour implémenter un noyau s'exécutant sur une carte graphique, mais le gain de temps est incomparable avec celui des versions OpenMP. De ce fait, pour de gros calcul, cela peut être intéressant d'investir du temps de programmation en OpenCL pour gagner beaucoup de temps d'exécution. Pour de petits programmes, ce choix est discutable cependant.

Pour les versions MPI, leur implémentation demande plus de travail que les versions OpenMP, et sans optimisation un peu poussée, le gain de temps n'est pas suffisamment intéressant. Son utilisation se justifie donc lors de très gros calculs, qui ne peuvent être effectués sur une seule machine.