Projet jeu de la vie

Auteurs
Maxime Mondot
Adrien Rodrigues

Responsables du projet
Raymond Namyst

5 juin 2017

Table des matières

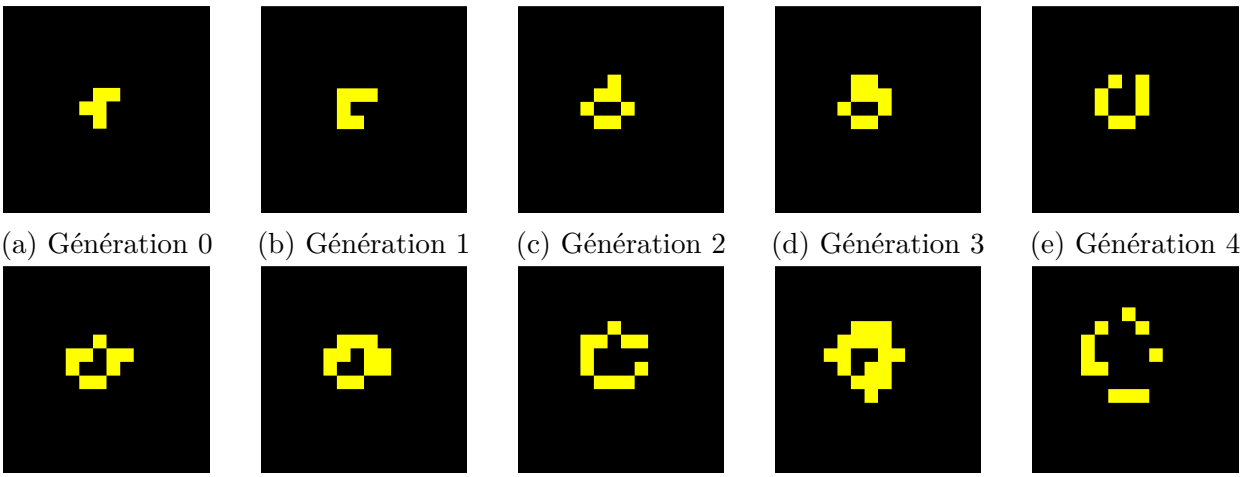
1 Introduction

Le jeu de la vie est un automate cellulaire simulant une forme basique de "vie" dans une grille orthogonale composée de cellules vivantes et de cellules mortes. Le temps est découpé en générations et à chaque génération, chaque cellule vit, meurt, naît ou subsiste selon des règles très précises. Ainsi dans le jeu qui est étudié dans ce projet, les règles sont les suivantes :

— Une cellule vivante survit à la génération suivante si et seulement si elle est entourée de deux ou trois cellules vivantes sinon elle meurt.

— Une cellule morte naît à la génération suivante si et seulement si elle est entourée de trois cellules vivantes.

Ce jeu étant prouvé Turing complet, il est possible de simuler n'importe quel programme informatique grâce à ce dernier.



(a) Génération 0 (b) Génération 1 (c) Génération 2 (d) Génération 3 (e) Génération 4

(f) Génération 5 (g) Génération 6 (h) Génération 7 (i) Génération 8 (j) Génération 9

FIGURE 1 – Premières générations de l'évolution du r-pentomino dans le jeu de la vie

Le but de ce projet est de comparer l'efficacité de l'implémentation de ce jeu à l'aide de différents outils.

On discutera, dans un premier temps, des différentes possibilités d'implémentation de ce jeu dans le cadre d'une exécution séquentielle ainsi que les différences de performances qui en découlent.

On s'attachera ensuite aux différentes possibilités d'exécution parallèle de ces algorithmes ainsi qu'à leurs performances. La parallélisation s'effectuant tout d'abord sur les différents cœurs d'un processeur grâce à OpenMP puis sur carte graphique à l'aide d'OpenCL.

Enfin nous essaierons de créer une version hybride où le processeur et la carte graphique travaillent de concert afin d'accélérer le plus possible la vitesse de calcul des différentes générations.

L'ensemble des tests ayant été effectué sur un ordinateur de la salle 201 du CREMI pour les tests OpenCL (carte graphique GTX 1070) et sur un ordinateur de la salle 203 pour les tests OpenMP (disposant de deux processeurs possédant au total 12 cœurs physiques et 24 cœurs hyperthreadés).

2 Implémentation séquentielle

Le but de cette partie est d'écrire différents algorithmes permettant de calculer les différentes générations du jeu de la vie.

2.1 Implémentation naïve

Cette première implémentation est la plus basique possible et servira donc de référence pour observer le gain de temps des autres implémentations par la suite.

Le principe de base de cette implémentation, et de toutes celles qui suivent, est de travailler sur deux grilles, l'une décrivant l'état du jeu à la génération actuelle (on peut la voir comme une grille d'entrée) et l'autre devant décrire l'état du jeu à la génération suivante (on peut la voir comme une grille de sortie). Les grille étant interchangées à chaque calcul d'une nouvelle génération. Ce dernier ce contentant d'échanger la valeur de deux pointeur s'effectue ainsi en temps constant. Ce système permet de ne pas avoir à ce soucier de l'ordre dans lequel les cellules de la nouvelle générations doivent être calculées à chaque étape car le calcul ne dépend que de la grille de la génération actuelle qui n'est pas changée lors de cette opération.

Ainsi l'implémentation naïve se contente, pour chaque cellule de la grille, de calculer le nombre de voisins vivant pour cette cellule et de renseigner dans la grille correspondante à la génération suivante l'état que va avoir cette cellule suivant les règles du jeu. Cependant les cellules se trouvant sur les bords de la grille de jeu sont ignorées car elles ne possèdent pas huit voisins.

```
for (int y = 1; y < DIM-1; y++)
    for (int x = 1; x < DIM-1; x++)
        compute_case(x,y);
```

FIGURE 2 – Code simplifié de calcul d'une génération de manière séquentielle et naïve

On peut ainsi voir sur la figure 2 comment ces deux idées sont implémentées. Le tableau est représenté en mémoire dans un tableau à une seule dimension où chaque ligne de la grille de jeu est mise bout à bout. Ainsi la case (x, y) de la grille de jeu correspond à la case $(y * DIM + x)$ de ce tableau. On a donc fait varier rapidement les x et lentement les y pour le calcul des différentes cellules dans le but que les accès en mémoire du tableau se fassent de manière contigus et que la mise en cache des valeurs proches par le processeur permettent d'accélérer le calcul des valeurs suivantes. Ainsi si les deux boucles sont inversées les calculs sont plus de deux fois plus long.

2.2 Implémentation tuilée

Cette version ressemble à la version précédente mais ici la grille de jeu est découpée en tuiles plus on moins grandes (32x32, 64x64 ou 128x128 cellules). Les calculs n'étant alors plus effectués en parcourant la grille entière ligne par ligne mais en en parcourant chaque tuile de la grille et en calculant, pour chaque tuile, l'état de cette tuile pour la génération suivante.

```
for (int tuiley = 0; tuiley < nb_tuiles; tuiley++)
    for (int tuilex = 0; tuilex < nb_tuiles; tuilex++)

        for (int yloc = 0; yloc < GRAIN; yloc++) {
            for (int xloc = 0; xloc < GRAIN; xloc++) {
                unsigned y=tuiley*GRAIN+yloc;
                unsigned x=tuilex*GRAIN+xloc;
                if (x>0 && x<DIM-1 && y>0 && y<DIM-1)
                    compute_case(x,y);
            }
        }
```

FIGURE 3 – Code simplifié de calcul d'une génération de manière séquentielle et tuilé

On peut voir sur la figure 3 comment est implémenté cet algorithme. Il s'avère en comparant la vitesse d'exécution de cette version par rapport à celle de la version précédente que cette version est en moyenne 25% plus lente quelque soit la taille des tuiles. On peut expliquer cette lenteur par deux éléments :

— Tout d'abord l'utilisation des tuiles ne nous permet plus d'ignirer facilement les cellules se trouvant au bords de la grille de jeu. Il devient alors nécessaire de tester, pour chaque cellule, si elle n'est pas aux bords de la grille de jeu. Ces tests supplémentaires augmentant la charge de calcul du CPU.

— Le découpage en tuile ne permet plus de lire les données ligne par ligne et de manière quasiment contiguë dans les tableaux représentant les générations en mémoire. Il est ainsi probable que le processeur fasse plus d'erreur de caches quand on accède aux différentes lignes d'une tuile ce qui va ralentir l'exécution du programme en nous forçant d'aller chercher en mémoire les valeurs que nous ne possédons pas.

2.3 Implémentation séquentielle optimisée

Il est possible lorsque l'on calcule plusieurs générations d'une configuration de départ particulière du jeu de la vie qu'une grande partie des tuiles calculées n'évoluent pas pour deux raisons principales : elles sont vides (la majeure partie des cas) ou elles possèdent une configuration de cellules n'évoluant pas.

Afin d'améliorer la vitesse d'exécution du programme, l'idée est de ne calculer une tuile que si celle-ci peut changer de configuration à la génération suivante. Afin de savoir quelles tuiles traiter, nous avons ajouté deux tableaux de $nb\_tuiles \times nb\_tuiles$ cases chacun et composés de booléens indiquant si la tuile n'a pas changé de configuration à la génération actuelle et si elle ne changera pas de configuration à la génération d'après.

```
for (int tuiley = 0; tuiley < nb_tuiles; tuiley++) {
    for (int tuilex = 0; tuilex < nb_tuiles; tuilex++) {

        if (compute_tile_required(tuilex, tuiley))
            next_unchanged[coord(tuiley,tuilex)] =
               compute_tile_changement(tuilex, tuiley);
        else
            next_unchanged[coord(tuiley,tuilex)] = true;

    }
}
```

FIGURE 4 – Code simplifié de calcul d'une génération de manière optimisé séquentiellement

On peut ainsi voir sur la figure 4 le fonctionnement simplifié de cet algorithme qui va pour chaque tuile :

- Déterminer si la tuile doit être calculée ou non (fonction *compute_tile_required*), pour cela on regarde si la tuile en question a été changée ou pas à la génération actuelle mais on regarde également si une des tuiles voisines a été changée à la génération actuelle. En effet il est possible qu'une tuile voit sa configuration changer à cause d'une des tuiles qui l'entoure.
- Si elle doit être calculée, on calcule son évolution en notant si oui ou non la tuile a bien été modifiée et on renseigne cette information dans le tableau de la génération suivante.
- Si non, on peut directement dire que la tuile n'a pas été modifiée lors de la génération en cours.

Il est a noter que les tuiles qui ne sont pas calculées ne sont même pas copiées pour la génération suivante. En effet, une tuile est détectée comme stagnante lors du calcul d'une génération et pendant ce calcul la tuile du tableau de la génération suivante est alors identique à celui de la génération actuelle (car la tuile est stagnante). Il ne devient alors plus nécessaire de copier le contenu de cette tuile à chaque génération.

Il est ainsi évident, avec cet algorithme de génération que la vitesse de calcul ne dépend plus que des seuls facteurs que sont la taille de la grille et le nombre d'itérations calculées mais dépend alors également de la taille des tuiles ainsi que du remplissage de la grille. En effet une grille quasiment vide, verra la plupart de ses tuiles considérées comme inactives rapidement ce qui diminuera grandement la quantité de calculs à effectuer pour calculer les générations suivantes du jeu. De plus, l'implémentation séquentielle fonctionne mieux avec des tuiles relativement petites (mais qui restent suffisamment grandes pour limiter les problèmes liées à la mise en cache des différentes tuiles). En effet, avec des tuiles plus petites, la détection des tuiles inactives est plus précise ce qui diminue le nombre de calculs dans des configuration creuses.

— Version naïve : 42 s
— Version optimisé, configuration gun (tuiles 32x32) : 2.3 s
— Version optimisé, configuration aléatoire (tuiles 32x32) : 50 s
— Version optimisé, configuration gun (tuiles 64x64) : 4.4 s
— Version optimisé, configuration aléatoire (tuiles 64x64) : 50.6 s
— Version optimisé, configuration gun (tuiles 128x128) : 8.8 s
— Version optimisé, configuration aléatoire (tuiles 128x128) : 50.2 s

FIGURE 5 – Vitesses d'exécutions du programme pour différentes tailles de tuiles et remplissages de grille (grille 2048x2048, 800 générations)

On peut ainsi observer sur la figure 5 l'influence des facteurs évoqués précédemment. On voit bien que le temps de calcul, pour une grille creuse est très impacté par la taille des tuiles et que celui-ci double en même temps que la largeur des tuiles. Il est intéressant de noter que dans ce type de configuration la version optimisée arrive tout de même à tourner 20 fois plus vite que l'implémentation naïve. Finalement, comme prévu, une grille aléatoire, c'est à dire dont toutes les tuiles sont tout le temps en train d'évoluer, présente des mauvaises performances car toutes les tuiles sont calculées à chaque génération. Cette version est même plus lente que la version naïve à cause des différents tests sur les tuiles s'effectuant à chaque génération et tentant de savoir s'il faut calculer la tuile ou pas. On peut donc en conclure que cette version est très intéressante quand on sait à l'avance que la grille de jeu sera majoritairement stagnante mais est à éviter pour des grilles donc l'ensemble des tuiles évolue en permanence.

3 Implémentation par boucles parallélisées

3.1 Implémentation naïve parallélisée

Cette première implémentation parallélisée reprends l'implémentation séquentielle naïve. La seule variation est l'utilisation de la bibliothèque OpenMP pour paralléliser les deux boucles principales du calcul. Les calculs effectués sont les mêmes que précédemment, mais ils sont cette fois ci effectués sur plusieurs cœurs des différents processeurs de la machine.

```
#pragma omp parallel for schedule(static) collapse(2)
for (int y = 1; y < DIM-1; y++)
    for (int x = 1; x < DIM-1; x++)
            compute_case(x,y);
```

FIGURE 6 – Code simplifié de calcul d'une génération de manière parallèle

On peut voir sur la figure 6 une version possible de code simplifié pour paralléliser la version séquentielle précédemment présentée. Dans le cas présenté, chaque processeur effectue une partie fixée des $(DIM-1)^2$ calculs à effectuer.

La figure 7 présente le speedup obtenu par rapport à la version séquentielle naïve. Le nommage de l'ensemble des courbes de speedup suivront le même formalisme qui est le suivant :
— La version utilisé est le nombre suivant le v (ici il s'agit de la version 3).
— La taille de la grille est le nombre suivant le n (ici la grille a une taille de 1024x1024 cellules).
— Le nombre suivant le i est le nombre d'itérations (ici 200) et le nombre suivant le t correspond à la taille des tuiles utilisé (ici absent car les tuiles ne sont pas utilisés).

— Il est enfin indiqué le type de scheduling dans le cas de parallélisation avec le pragma for (static ou dynamic).

On note ainsi que la courbe n'est pas linéaire. Ceci peut s'expliquer par la présence de 12 cœurs physiques hyperthreadés, on note en effet un maximum local, relativement proche du maximum global atteint pour un nombre de threads égal à 12, suivi d'un baisse de performances. En effet, chaque cœur physique est associé à deux logiques, ainsi un noeud physique peut être occupé par deux threads, et il effectuera donc deux fois plus de calculs que les autres, en supposant que chaque thread est bindé à un unique cœur logique. D'où la baisse de performances. On remarque que le speedup recroît ensuite vers un maximum lorsque la répartition de calcul se rééquilibre entre les processeurs.

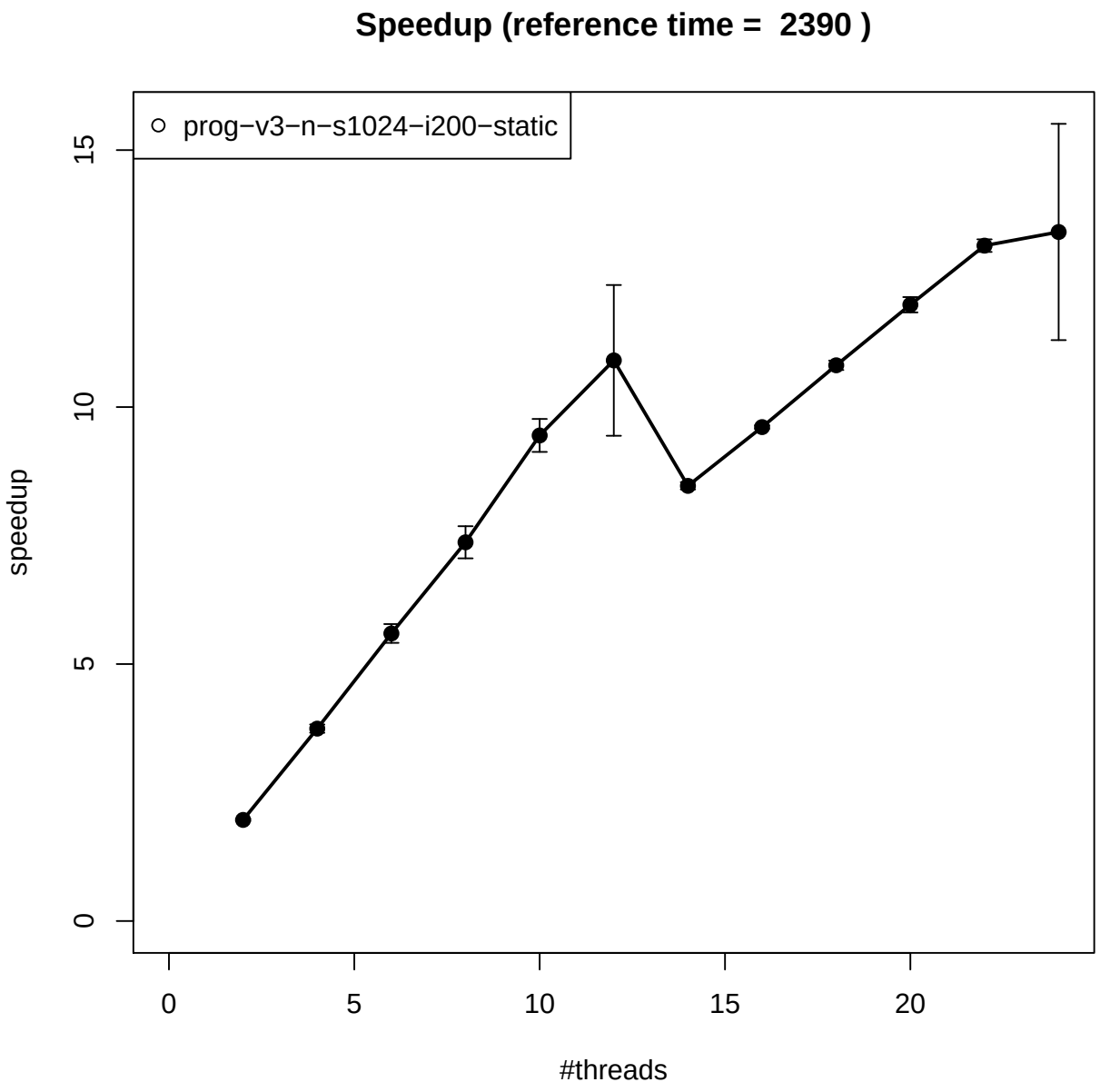


Figure 7 – Speedup de la version OpenMP for naïve par rapport au nombre de cœurs

Les pertes de performances pour les valeurs 12 et 24 que montrent la courbe peuvent s'expliquer par la présence d'un autre thread qui doit s'exécuter sur l'un des cœurs, on peut supposer qu'il s'agit du système d'exploitation. Ce dernier explique sans doute également l'importante variance observé pour ces deux valeurs. En effet le système d'exploitation peut effectuer une quantité de travail différente en fonction du temps et ainsi faire varier de manière plus importante le temps d'exécution de notre programme.

Plusieurs tests ont été réalisés sur la méthode de parallélisation, en particulier, si on utilise une méthode de scheduling dynamique, on s'aperçoit que les temps de calculs sont nettement plus long :

— schedule(dynamic, 1) : 8488.246 ms
— schedule(dynamic, 2) : 4669.070 ms
— schedule(dynamic, 4) : 2504.147 ms
— schedule(dynamic, 16) : 635.305 ms
— schedule(dynamic, 32) : 431.335 ms
— schedule (static) : 187.674 ms

On note que le temps de calcul est inversement proportionnel à la taille des blocs pour de petites tailles. Ce comportement peut s'expliquer par les problèmes de mise en cache qu'impliquent le scheduling dynamique. Chaque calcul étant affecté au premier processeur disponible, la répartition des calculs s'approche du schéma suivant : Un processeur effectue les calculs sur les cases $i * nb\_core + core\_id$ les accès mémoires ne sont alors plus contigus, ce qui cause un plus grand nombre d'erreurs de cache. À cela s'ajoute le temps de répartition des calculs. L'augmentation de la taille des blocs permet de réduire à la fois le nombre de calculs à répartir, et le nombre d'erreur de cache d'où l'amélioration des performances. Cependant il n'est à priori pas possible de dépasser les performances du scheduling statique dans ce cas, car le calcul d'une case s'effectue en temps constant.

3.2 Implémentation tuilée parallélisée

Cette implémentation reprends la version séquentielle tuilée, on utilise une nouvelle fois OpenMP pour paralléliser les deux boucles principales du programme, chaque fragment de calcul correspond alors au calcul d'une tuile complète et non d'une seule case.

```
#pragma omp parallel for schedule(static) collapse(2)
for (int tuiley = 0; tuiley < nb_tuiles; tuiley++)
    for (int tuilex = 0; tuilex < nb_tuiles; tuilex++)

        for (int yloc = 0; yloc < GRAIN; yloc++) {
            for (int xloc = 0; xloc < GRAIN; xloc++) {
                unsigned y=tuiley*GRAIN+yloc;
                unsigned x=tuilex*GRAIN+xloc;
                if (x>0 && x<DIM-1 && y>0 && y<DIM-1)
                    compute_case(x,y);
            }
        }
```

FIGURE 8 – Code simplifié de calcul d'une génération de manière séquentielle et tuilé

On peut voir sur la figure 8 une méthode de parallélisation similaire à la précédente, où chaque processeur va exécuter une portion fixe des nb_tuiles^2 calculs de fragments à effectuer.

Le speedup obtenu pour cette implémentation est représenté en figure 9. On note un comportement relativement similaire au speedup de la version naive, en particulier pour les tuiles les plus petites. Pour ce qui est de la version avec tuiles de 128×128, il est important de noter qu'un tableau de 1024×1024 ne comporte que 64 tuiles au total, la répartition ne peut donc pas être équitable d'où les baisses de performances pour 18 et 20 threads.

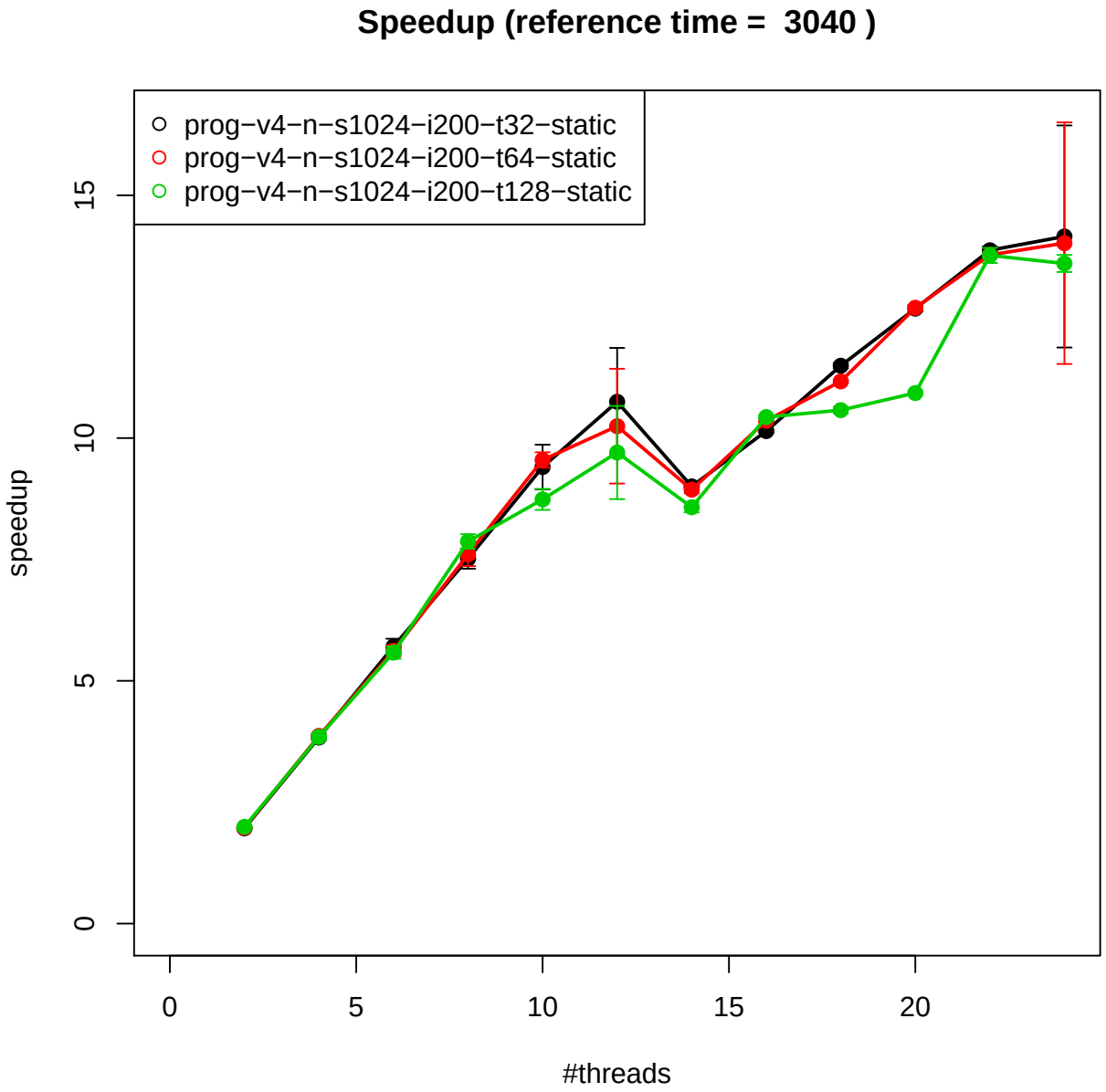


FIGURE 9 – Speedup de la version OpenMP for tuilé par rapport au nombre de cœurs (scheduling statique)

Cette fois-ci les tests sur un scheduling dynamique apportent des résultats beaucoup plus intéressants. On remarque premièrement l'absence de cassure dans la courbe de speedup. De plus on observe des performances équivalentes au scheduling statique. L'absence de pertes de performances dans cette implémentation est due à l'utilisation du tuiles, ce qui d'une part réduit le nombre de calculs à répartir, mais surtout permet à chaque processeur une meilleure mise en cache des données lors de l'exécution d'un calcul. Pour ce qui est du comportement général du speedup en fonction du nombre de threads, ceci est directement causé par la méthode de

scheduling : chaque calcul étant affecté au premier cœur disponible, ceci permet notamment d'assurer une répartition équitable du temps de calcul, en plus de compenser les éventuelles différences de temps d'exécution sur chaque tuile.

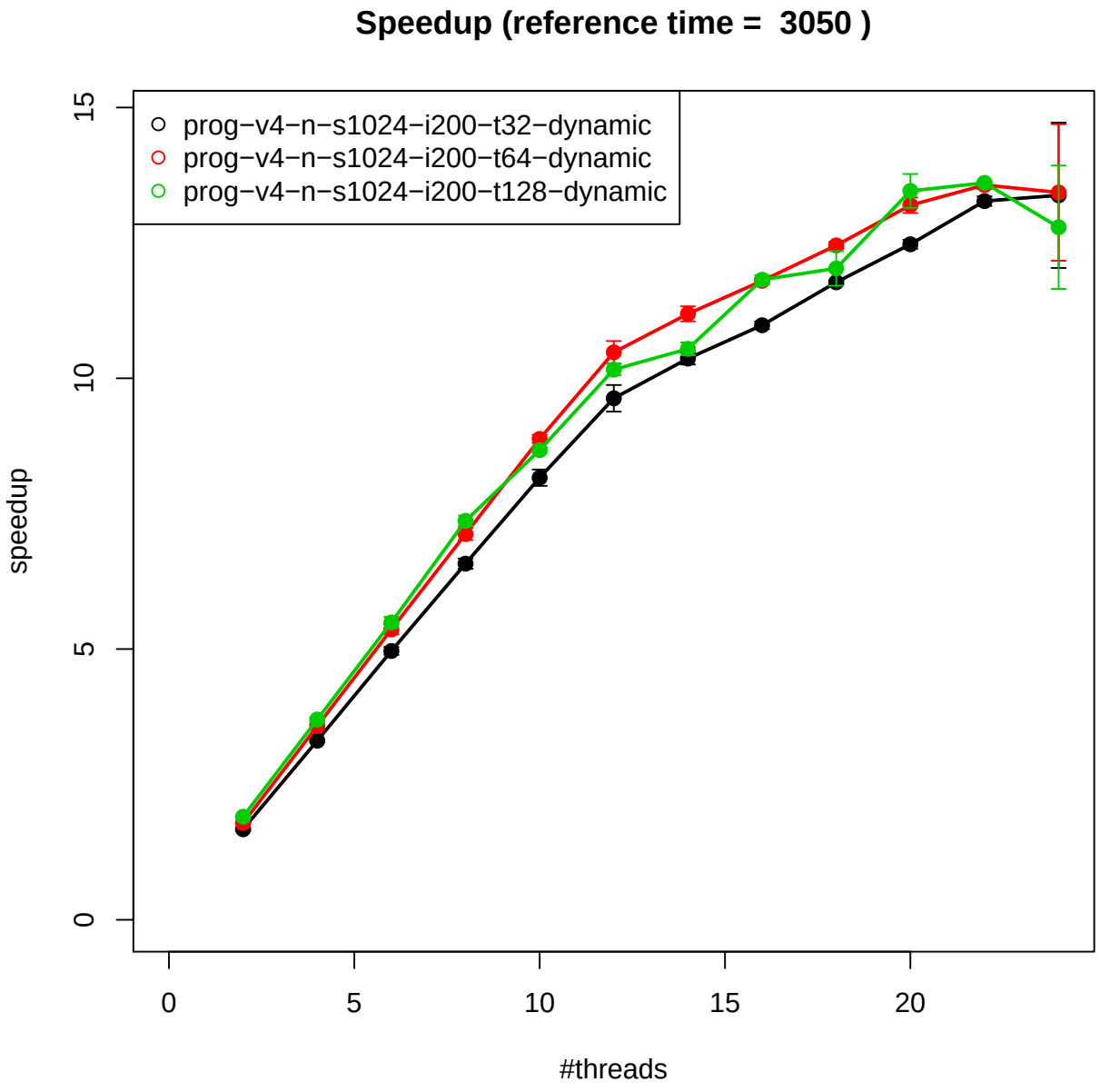


FIGURE 10 – Speedup de la version OpenMP for tuilé par rapport au nombre de cœurs (scheduling dynamique)

3.3 Implémentation Optimisée parallélisée

Cette implémentation applique le même raisonnement que précédemment : On détermine si la tuile doit être calculée ou non avant de lancer le calcul si nécessaire. Cela se fait toujours en utilisant deux tableaux de booléens indiquant si chaque tuile a été changée ou non à l'itération courante et à l'itération suivante.

```
#pragma omp parallel for collapse(2) schedule(static)
for (int tuiley = 0; tuiley < nb_tuiles; tuiley++) {
    for (int tuilex = 0; tuilex < nb_tuiles; tuilex++) {
        if (compute_tile_required(tuilex, tuiley))
            next_unchanged[coord(tuiley,tuilex)] =
               compute_tile_changement(tuilex, tuiley);
        else
            next_unchanged[coord(tuiley,tuilex)] = true;
    }
}
```

FIGURE 11 – Code simplifié de calcul d'une génération de manière optimisé parallèlement

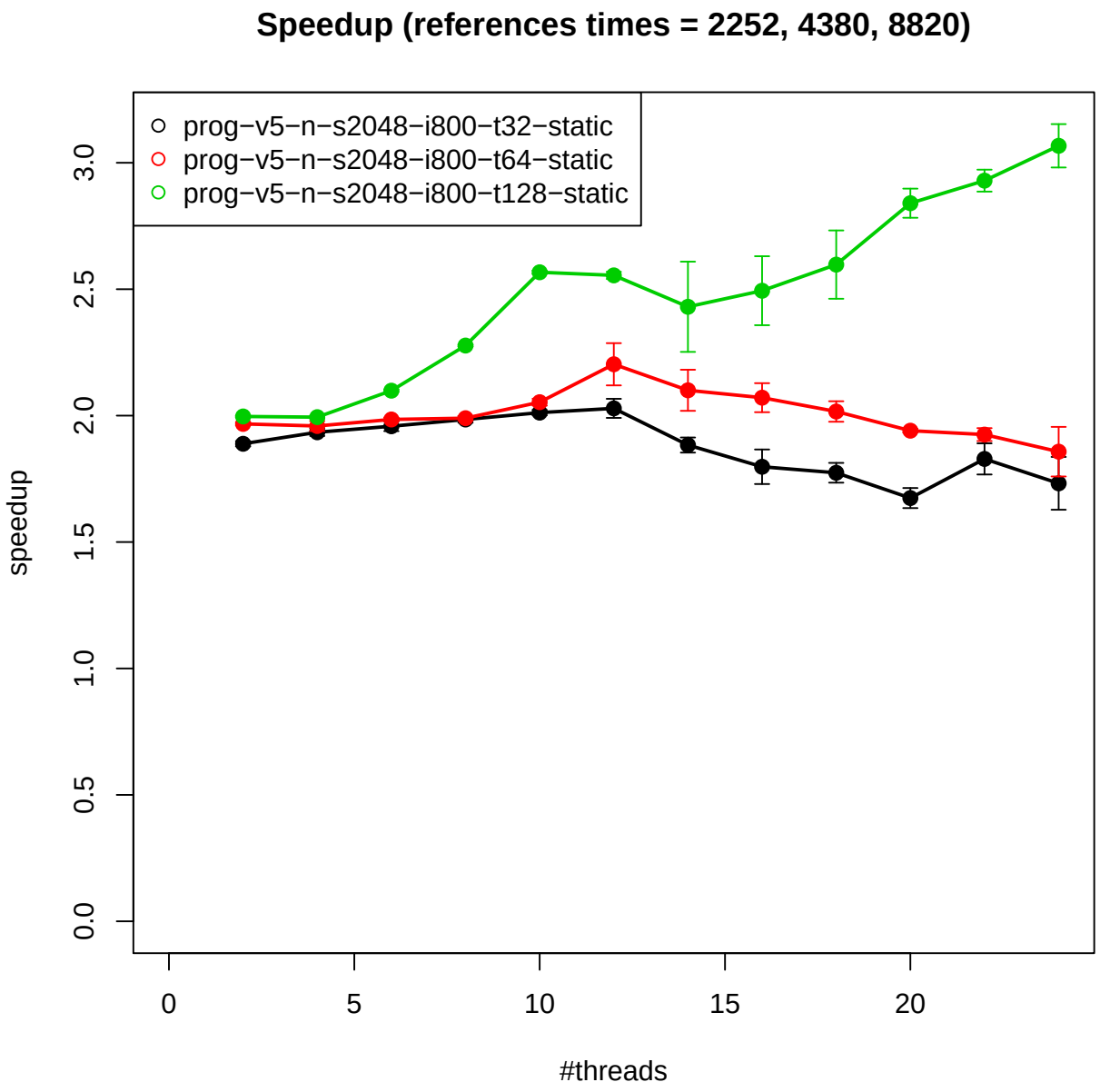


FIGURE 12 – Speedup de la version OpenMP for optimisé par rapport au nombre de cœurs (scheduling statique)

On peut voir sur la figure 11 un exemple de parallélisation, utilisant une fois encore un scheduling statique. Tous les processeurs effectuent donc les calculs sur une partie prédéfinie de la grille. Les résultats obtenus pour ce scheduling sont présentés en figure 12. On note que le speedup est stagnant et qu'il ne dépasse pas 3 dans le meilleur des cas. Cet absence de gain s'explique par le fait que, étant donné que la répartition des calculs est faite à l'avance, et que le calcul sur une tuile ne s'effectue plus en temps constant, il est probable qu'un thread ait à effectuer un plus grand nombre de calculs effectif et ralentisse donc l'exécution globale du programme.

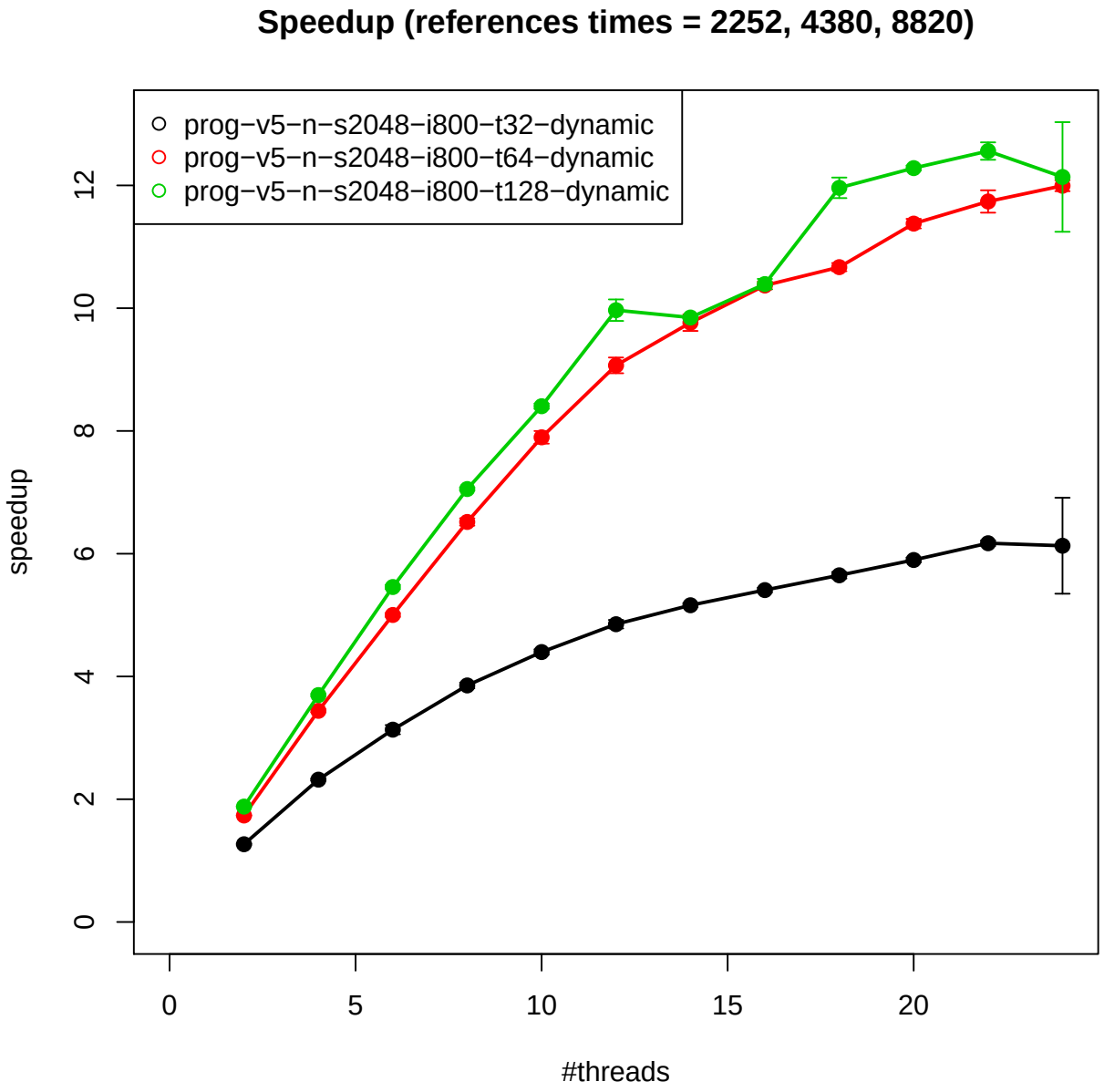


FIGURE 13 – Speedup de la version OpenMP for optimisé par rapport au nombre de cœurs (scheduling dynamique)

Dans le cas d'un scheduling dynamique, dont les résultats sont présentés en figure 13 on remarque un gain de performance nettement plus important et également plus dépendant du nombre de threads. En effet dans le cas du scheduling dynamique, le temps d'exécution sur

chaque tuile est pris en compte puisque les tâches sont réparties durant l'exécution du programme. Ainsi le temps de calcul global est mieux réparti entre les différents noyaux, et on observe en général de meilleurs performance pour ce scheduling.

On note également l'influence de la taille des tuiles, dans ce cas, on observe des speedups réduits pour les petites tailles. Ceci pourrait être dû au fait que des tuiles plus petites impliquent un plus grand nombre de tâches, et que le coût d'ordonnemecement de ces tâches devienne non négligeable.

4 Implémentation par tâches

4.1 Implémentation tuilée par tâches

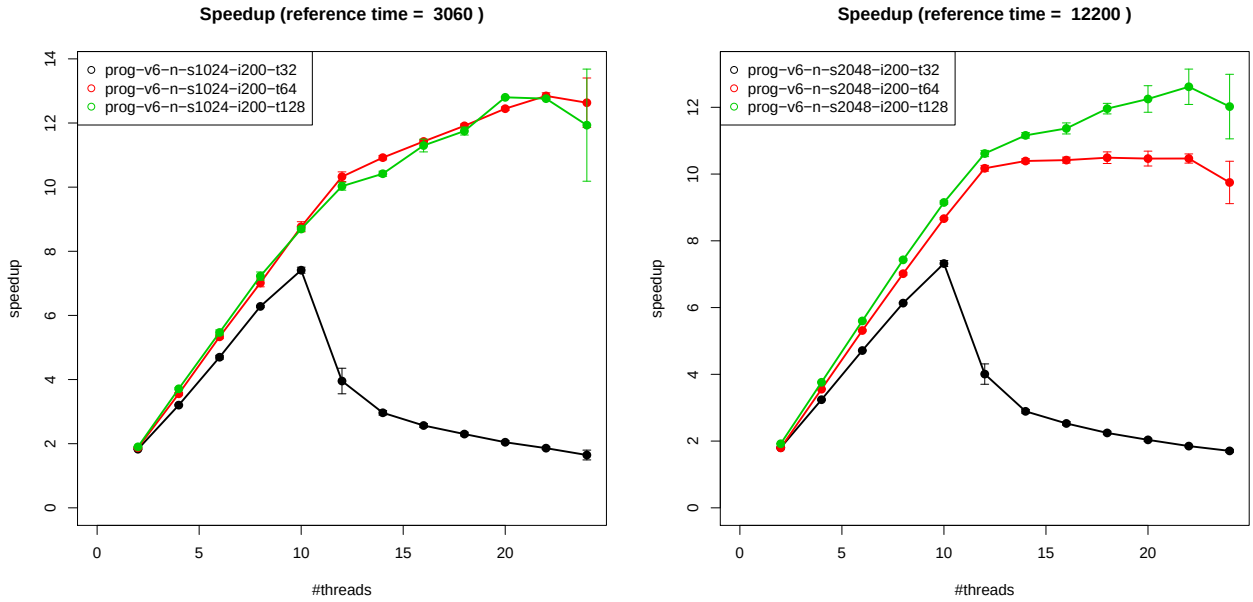
Cette implémentation a un comportement assez similaire à celle de la version tuilée parallélisée avec un scheduling dynamique. En effet l'un des processus va créer une tâche pour chaque tuile à calculer (une tâche étant alors le calcul d'une tuile pour la génération suiante), et les autres processus exécuteront ensuite ces tâches dans un ordre arbitraire avant d'être rejoint par le premier processeur une fois que ce dernier a fini de créer l'ensemble des tâches à calculer. Le code correspondant à cette implémentation est présenté en figure 14

```
#pragma omp parallel
{
#pragma omp single
    for (int tuiley = 0; tuiley < nb_tuiles; tuiley++)
        for (int tuilex = 0; tuilex < nb_tuiles; tuilex++)
#pragma omp task firstprivate(tuilex, tuiley)
            compute_tile(tuilex, tuiley);
}
```

FIGURE 14 – Code simplifié de calcul d'une génération de manière séquentielle et tuilé

La figure 15 présente deux ensembles de résultats, sur des grilles de tailles 1024×1024 et 2048×2048. On remarque des performances équivalentes à celles obtenues jusqu'ici pour des tailles de tuiles suffisamment grandes, en revanche on note une limite à cette implémentation, particulièrement mise en valeur par la courbe rouge du graphe *b*. Ceci s'explique par le fait que le nombre de taches à générer est inversement proportionnel à la taille des tuiles, et le temps d'exécution d'une de ces taches est lui proportionnel à cette taille. Il arrive donc pour un nombre de threads suffisamment grand que les taches soient traitées plus rapidement qu'elles ne sont créées, d'où la limite observée dans ce cas.

Pour des tuiles de 32×32 cellules, le speedup diminuant pour atteindre 2 sur 24 cœurs. Le phénomène semble similaire à celui qui limite le speedup dans la figue 15b mais nous n'avons pas d'explications quand au fait que celui-ci diminue autant pour atteindre une valeur aussi faible.



(a) Grille de taille 1024×1024

(b) Grille de taille 2048×2048

FIGURE 15 – Speedup de la version OpenMP task tuilé par rapport au nombre de cœurs

4.2 Implémentation optimisée par tâches

Cette implémentation par tâches s'approche de la version optimisée parallélisée avec un scheduling dynamique. Un des processus va créer toutes les taches, et les autres processus vont commencer à les exécuter avant d'être rejoints par le premier. Le code correspondant est présenté en figure 16

```
#pragma omp parallel
{

#pragma omp single
    for (int tuiley = 0; tuiley < nb_tuiles; tuiley++) {
        for (int tuilex = 0; tuilex < nb_tuiles; tuilex++) {
            if (compute_tile_required(tuilex, tuiley))
#pragma omp task firstprivate(tuilex, tuiley)
                next_unchanged[coord(tuiley,tuilex)] =
                   compute_tile_changement(tuilex, tuiley);
            else
                next_unchanged[coord(tuiley,tuilex)] = true;
        }
    }

}
```

FIGURE 16 – Code simplifié de calcul d'une génération de manière séquentielle et tuilé

Les résultats associés à cette implémentation sont présentés en figure 17 on remarque que les résultats restent meilleurs pour des tuiles de plus grande taille. Il a également été remarqué que cette implémentation est plus efficace lorsque le test est effectué avant la création de la

tâche (on ne crée la tâche que si nécessaire) et non si ce test fait partie de la tâche. La forte divergence des résultats peut s'expliquer une fois encore par des problèmes de mise en cache : lorsque le thread qui crée les taches effectue le test, les tableaux de booléens ne sont mis en cache que sur un seul processeur, ce qui laisse plus d'espace en cache sur les autres processeur pour stocker l'état de la grille.

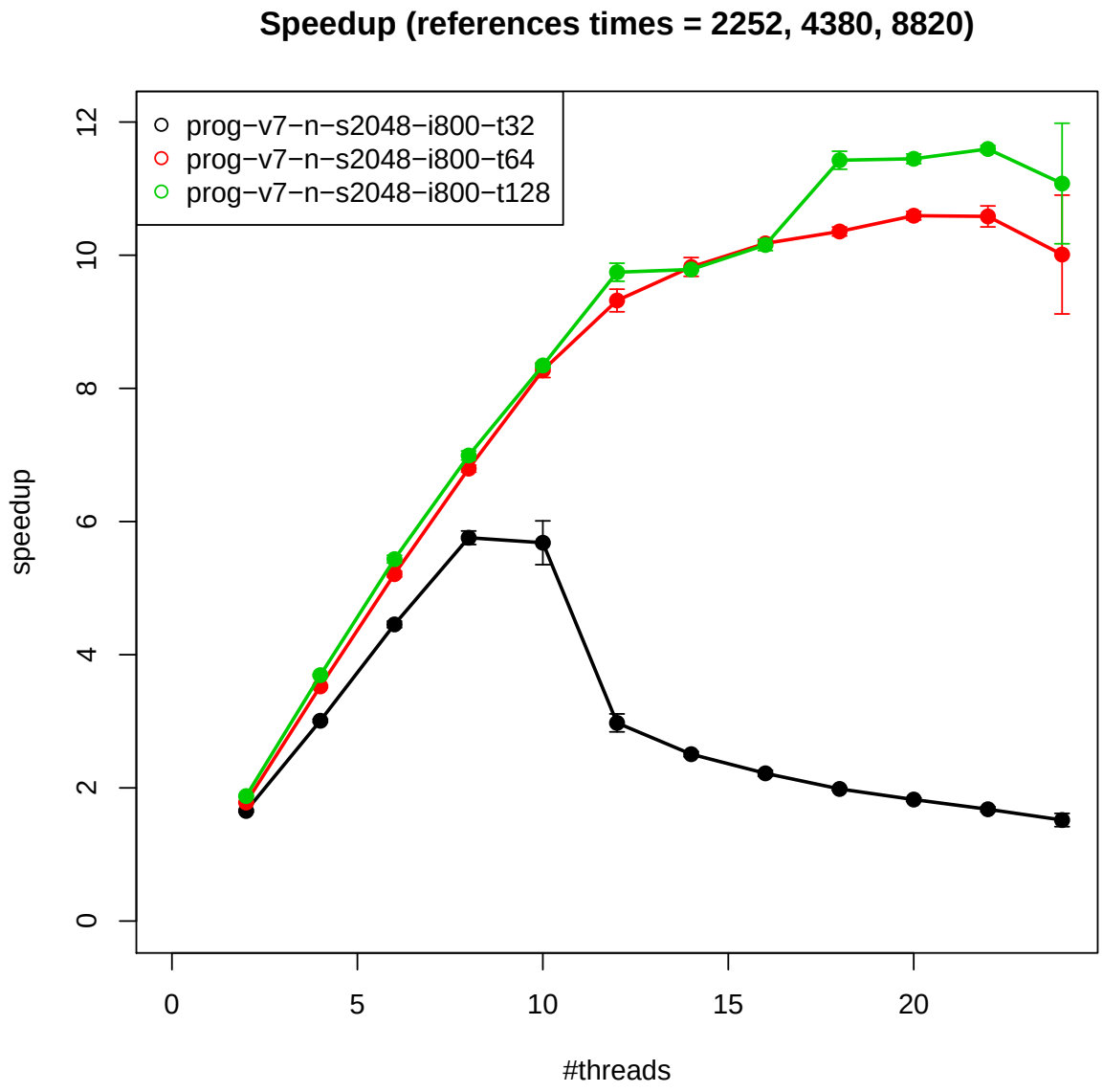


FIGURE 17 – Speedup de la version OpenMP task optimisée par rapport au nombre de cœurs

5 Implémentation sur GPU

De par leur nature hautement parallélisée, les cartes graphiques sont adaptées aux calculs s'effectuant sur des images. En effet, on peut assimiler la grille de jeu à une image où chaque cellule serait un pixel de la grille. Ainsi, on peut créer autant de threads sur la carte graphique que de cellules à calculer à chaque génération, ce qui permet d'exploiter le plus possible le parallélisme et la puissance de cet accélérateur graphique. Les différentes implémentations qui suivent sont donc toutes réalisés avec la bibliothèque openCL et s'exécutent sur GPU.

5.1 Version naïve sur GPU

La première version réalisée avec l'aide d'OpenCL, s'approche de la version naïve séquentielle. Il s'agit simplement de créer autant de threads GPU que de cellules dans le terrain de jeu, et chacun de ces threads effectue le calcul pour la case à laquelle il est associé. Il est toutefois nécessaire de tester la position de la cellule à faire évoluer pour ne pas faire de mauvais calcul si l'on se situe au bord du terrain de jeu. Cette opération n'était pas nécessaire avec la version séquentielle car on contrôlait le parcours de la grille de jeu or ici ce n'est pas le cas, la bibliotèque OpenCL appelant le Kernel pour chaque cellule de la grille quelque soit sa position.

5.2 Version tuilée sur GPU

La seconde version s'approche de son homologue séquentielle, à la différence que cette fois-ci, les tuiles sont stockées dans un bloc de mémoire locale afin de permettre des accès plus rapides. En effet chaque cellule de la grille de jeu est lu jusqu'à 9 fois pour calculer la grille de jeu de la génération suivante (une fois pour calculer la génération suivante de cette même cellule et une fois pour chacune de ses cellules voisines effectuant le même calcul). Or les lectures en mémoire globale sur GPU sont lentes et peuvent ralentir l'exécution du programme.

L'idée est donc de mettre en cache dans la mémoire locale des différentes unités de calcul les données nécessaires au calcul de la tuile à la génération suivante afin de ne lire la mémoire globale qu'une seule fois. Cependant copier les données de la tuile en cours de traitement n'est pas suffisant car pour calculer une tuile, il faut également connaître l'état des cellules voisines à cette tuile. Ainsi, ces cellules sont également recopiées dans la tuile locale (qui est alors plus grande pour cette raison) avant le début des calculs.

Une fois ces calculs effectués, chaque thread peut alors accéder à la mémoire locale de l'unité de calcul et ne fera plus de lecture de la grille d'entrée dans la mémoire globale. Seule une lecture dans la mémoire globale de sortie sera effectué à la fin dans le but de renseigner les données sur la génération suivante.

5.3 Version optimisée sur GPU

Cette dernière version reprend exactement le même fonctionnement que la version séquentielle mais utilise en plus, comme la version précédente, la mémoire locale pour mettre en cache les tuiles à calculer. Il a toutefois été nécessaire de rajouter deux paramètres au kernel et d'allouer deux nouveaux buffers sur la carte graphiques, ceux-ci correspondant alors aux deux tableaux de booléens décrivant les changements d'états de la version séquentielle et qui sont utilisés pour déterminer si une tuile doit être calculé ou non.

5.4 Comparaison des différentes versions entre elles

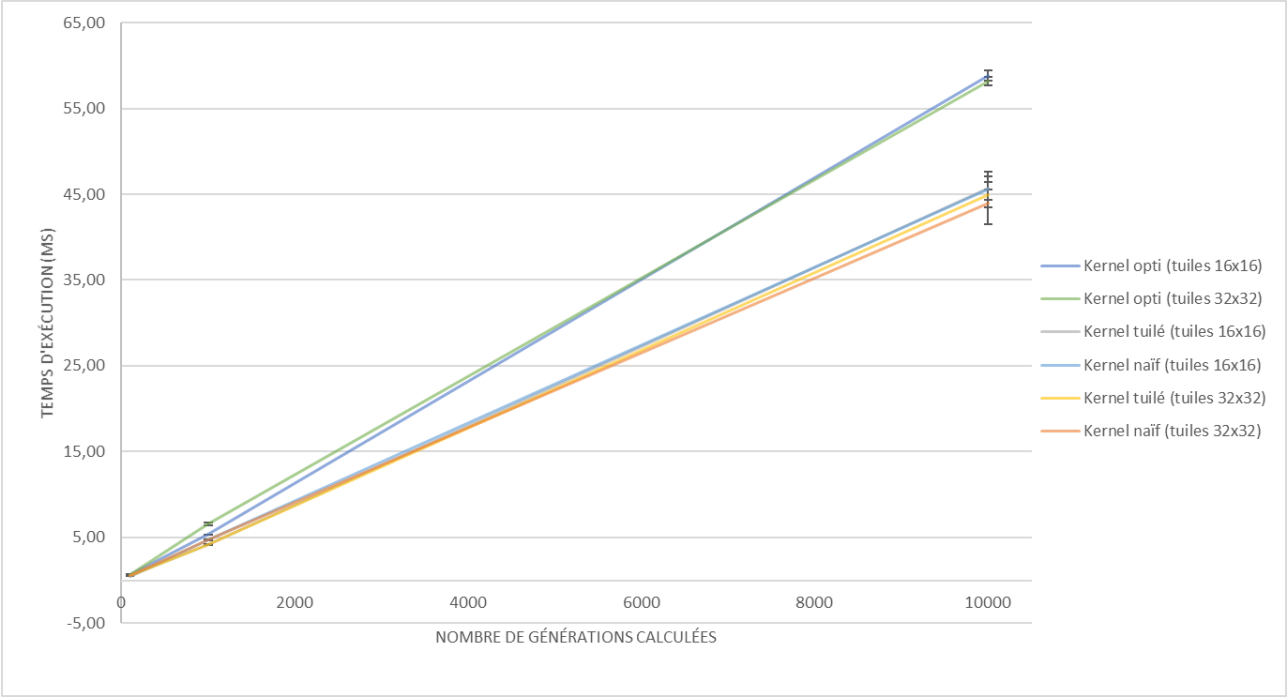


FIGURE 18 – Vitesse d'exécution des différentes versions OpenCL en fonction du nombre de générations (grille 256x256)

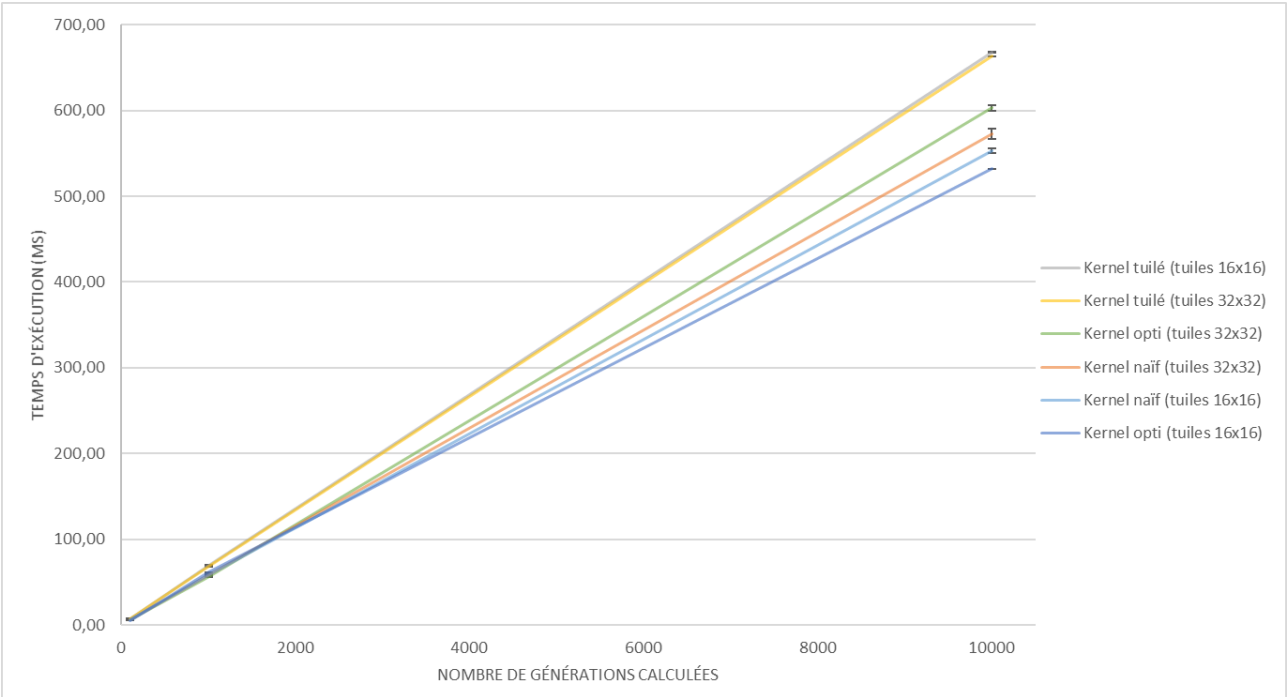


FIGURE 19 – Vitesse d'exécution des différentes versions OpenCL en fonction du nombre de générations (grille 1024x1024)

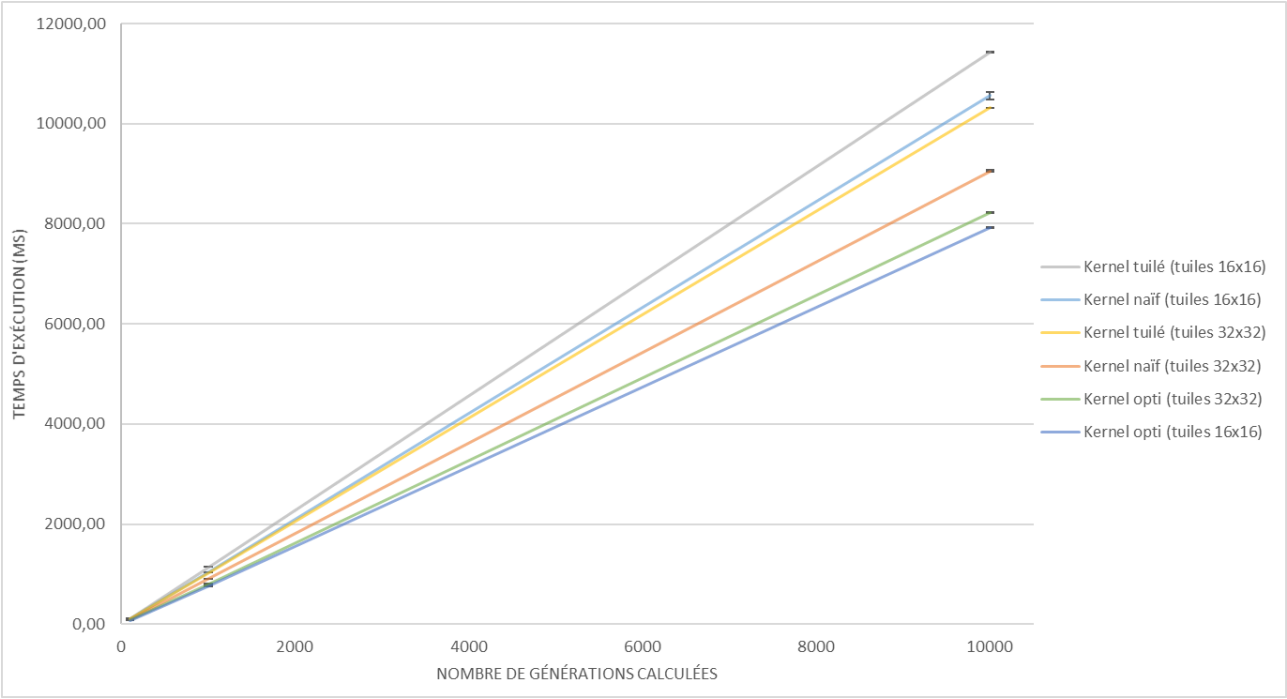


FIGURE 20 – Vitesse d'exécution des différentes versions OpenCL en fonction du nombre de générations (grille 4096x4096)

On peut voir sur les figures 18, 19 et 20 des courbes d'évolution du temps de calcul des différents kernels avec différentes tailles de tuiles et de grilles en fonction du nombre de générations calculées.

On peut premièrement remarquer que pour une grille de taille 256×256, le kernel optimisé est plus lent que les autres pour les deux tailles de tuile. Cela peut s'expliquer par le fait qu'à cette taille là, la grille se remplit relativement vite et que le GPU se retrouve rapidement à devoir calculer beaucoup de tuiles en plus de devoir calculer si oui ou non il doit calculer ces tuiles. L'écart type important des autres versions sur ce graphique nous empêche de déduire plus d'informations de ce dernier.

En s'intéressant aux deux autres graphiques on peut remarquer que la version tuilée est systématiquement plus longue à effectuer les calculs que la version naïve. Ce résultat n'est pas cohérent de prime abord car la version tuilée est justement censée mettre en mémoire locale une partie de la grille de calcul dans le but d'accélérer grandement les calculs par la suite. L'hypothèse la plus probable pour expliquer ce phénomène est que la mise en mémoire locale d'une partie du tableau global coûte aussi cher voir plus cher que le temps perdu par l'accès multiple à la même case mémoire de ce tableau dans la version naïve. En effet, il est nécessaire de charger, en plus de la tuile que l'on va calculer (ce qui se fait rapidement car les appels sont alignés), le contour de cette tuile. Pour effectuer ce chargement beaucoup de tests sont réalisés afin de limiter l'accès à la mémoire centrale mais chaque thread devant exécuter des instructions en même temps, chaque condition ralentit tous les threads de calculs. Le ralentissement est accentué par le fait que le chargement du contour se fait de manière fragmentée ce qui empêche le GPU d'aller chercher plusieurs valeurs dans la mémoire centrale en même temps alors que les appels à la mémoire centrale dans le kernel naïf se font tout le temps sur des valeurs alignées.

Enfin on peut voir sur ces graphiques que la version optimisée est souvent meilleure que ses homologues mais la différence de vitesse est relativement faible. D'autant plus que si la grille n'est plus creuse mais pleine cette version peut devenir deux fois plus lente que n'importe laquelle des autres versions à cause des tests effectuées inutilement. Encore une fois on peut expliquer ce gain faible par le rajout de tests dans le kernel ralentissant l'exécution global de ce dernier.

6 Implémentation hybride

Cette version consiste à faire travailler conjointement le GPU et le CPU. Il a été choisi d'utiliser les versions tuilées pour les deux.

Après comparaison des temps de calculs, il a été remarqué que pour obtenir un temps de calcul similaire sur CPU et GPU, 92% des calculs devaient être effectués sur GPU, et les 8% restants sur CPU (sur l'architecture de test). Cependant, il est nécessaire pour synchroniser ces calculs d'effectuer des communications entre GPU et CPU, afin d'uniformiser les données. Ces communications étant coûteuses, il est important de minimiser leur nombre. Ceci s'est fait en échangeant une ligne de tuiles toutes les $tuile\_y$ itérations, qui est le nombre d'itérations maximal avant que des données erronées soient utilisées pour des calculs dans le but de limiter les échanges de mémoire entre la mémoire centrale et celle du GPU qui coûtent cher. Dans les sources du projet, c'est une version basique qui est implémentée, des essais de vitesses ont été réalisés sur la version optimale mais celle-ci n'est pas présente dans les sources car les échanges de mémoires n'étaient pas tout à fait correct et le calcul des générations ne l'était pas non plus même si les temps de calculs restaient pertinent.

Malgré cet ensemble de précautions, il a été remarqué que le coût des communications est beaucoup trop important pour compenser le gain obtenu par le travail du CPU en plus du GPU. Ainsi le programme est 108 fois plus lent en moyenne que la version tuilée s'exécutant uniquement sur la carte graphique et est 3.5 fois plus lente que la version séquentielle.

7 Comparaison des différentes versions entre elles

La figure 21 présente les temps d'exécutions de l'ensemble des versions de chaque implémentation (Séquentielle, OpenMP for, OpenMP task et OpenCL) en fonction du nombre d'itérations, sur des fichiers exemples et des cas aléatoires. On remarque d'abord que les trois versions optimisées offrent les meilleurs résultats dans les cas d'exemples pour toutes les versions, même si le gain relatif est inférieur pour la version s'exécutant sur GPU. Dans les cas aléatoires, les test supplémentaires associés aux versions optimisées causent des baisses de performances, en effet les terrains de jeu générés aléatoirement sont dans la majorité des cas des matrices pleines, les calculs sont donc quasiment toujours effectués, d'où la baisse de performances.

Dans les cas d'exemples, on observe un facteur 19 entre la version séquentielle optimisée et la version OpenMP for optimisée. Les architectures de test comportant 12 noyaux physiques hyperthreadés, ce facteur corresponds à une utilisation de tous les cœrs logiques à 80% ce qui s'approche d'une utilisation optimale.

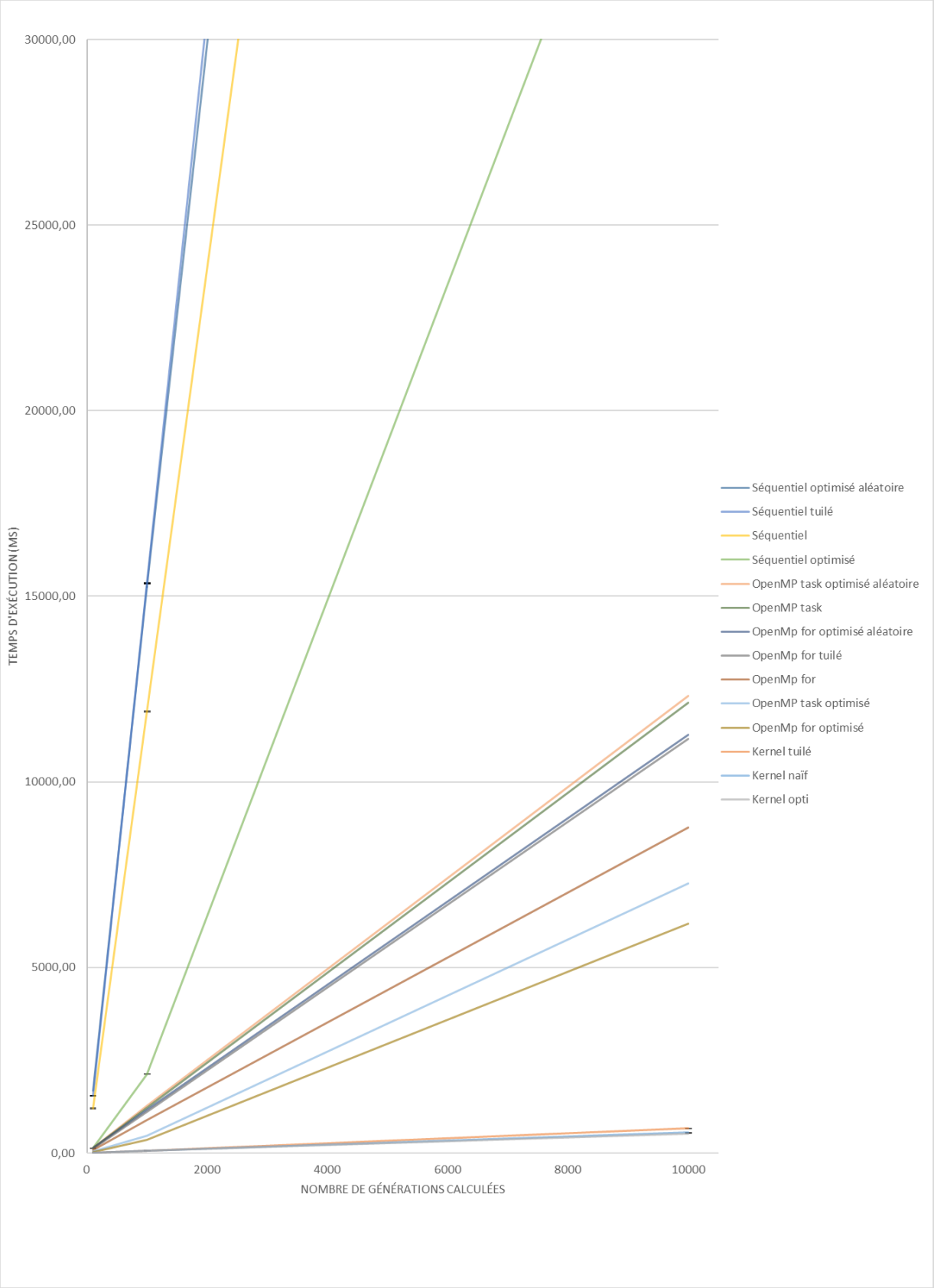


FIGURE 21 – Temps d'exécution de l'ensemble des versions en fonction du nombre d'itérations

La version optimisée par tâches reste plus lente que la version OpenMP for (environ 15% plus lente). En effet les contraintes sur les calculs étant quasiment inexistantes, les performances de la version OpenMP for sont déjà presque optimales. La création de tâches devient alors un

surcoût, c'est pourquoi la version par tâches est moins efficace que la version for.

Enfin on remarque que la version optimisée sur GPU s'exécute 10 fois plus rapidement que la version OpenMP for optimisée. En effet la structure des GPU est idéale pour les calculs effectués dans ces implémentations, un très grand nombre de petits calculs indépendants. D'où la supériorité de l'implémentation sur GPU pour cet algorithme.

Les résultats des exécutions sur des exemples aléatoires diffèrent légèrement, les versions optimisées n'occasionnant que des surcoût, ce ne sont plus les meilleurs choix dans ce cas de figure, cependant on observe des facteurs de vitesse similaires entre les différentes plate-formes.

8 Parser de fichiers RLE

Le logiciel Golly est un logiciel open source utilisant l'algorithme Hashlife décrit par Bill Gosper dans le but de générer très rapidement les générations d'un terrain du jeu de la vie ou d'autres jeux avec des règles différentes.

Il a été décidé d'écrire un parser pour les fichiers de sauvegarde de ce logiciel (extension rle) qui puisse reconnaître une grille de jeu encodée dans ce format et puisse la charger dans la mémoire de notre programme dans le but de faire évoluer cette grille de jeu. Cela nous a ainsi permis de lancer les fichiers d'exemples de Golly et de comparer leurs vitesses d'exécution entre celles de Golly et celles de notre programme mais également de vérifier que le calcul des différentes générations est bien correct ou de faciliter la création de grilles avec des conditions particulières.

Ainsi il suffit d'utiliser l'option de chargement de fichiers qui était déjà inclue dans le programme de base mais de lui passer en paramètre un fichier dont l'extension est ".rle", le programme va ensuite automatiquement les reconnaître et les charger en mémoire de le bonne façon. Cependant la taille du terrain de jeu n'est pas initialisée de manière automatique et l'utilisateur doit passer une taille de grille suffisamment grande pour que la configuration tienne dessus. Le parser vérifiant que la taille est suffisamment grande et centrant alors le pattern au centre de la grille. Ce choix permet de renseigner une taille de grille plus grande que le pattern de base et peut par exmple être utile dans le cas de pattern se déplacant ou créant des vaisseaux sortant de sa zone initiale par exemple.

Figure 22 – Génération 5640 du fichier rle/Guns/golly-ticker.rle d'une largeur de 3040 et faisant défiler le texte Golly de droite à gauche.

On peut voir en figure 22 et 23 deux exemples de configurations que l'on peut obtenir grâce à ce parser. Mais c'est également grâce à ce dernier que la figure 1 a été crée. Tous les fichiers d'exemple de Golly compatibles avec notre programme sont situées dans le dossier "rle" situé à la racine du projet.

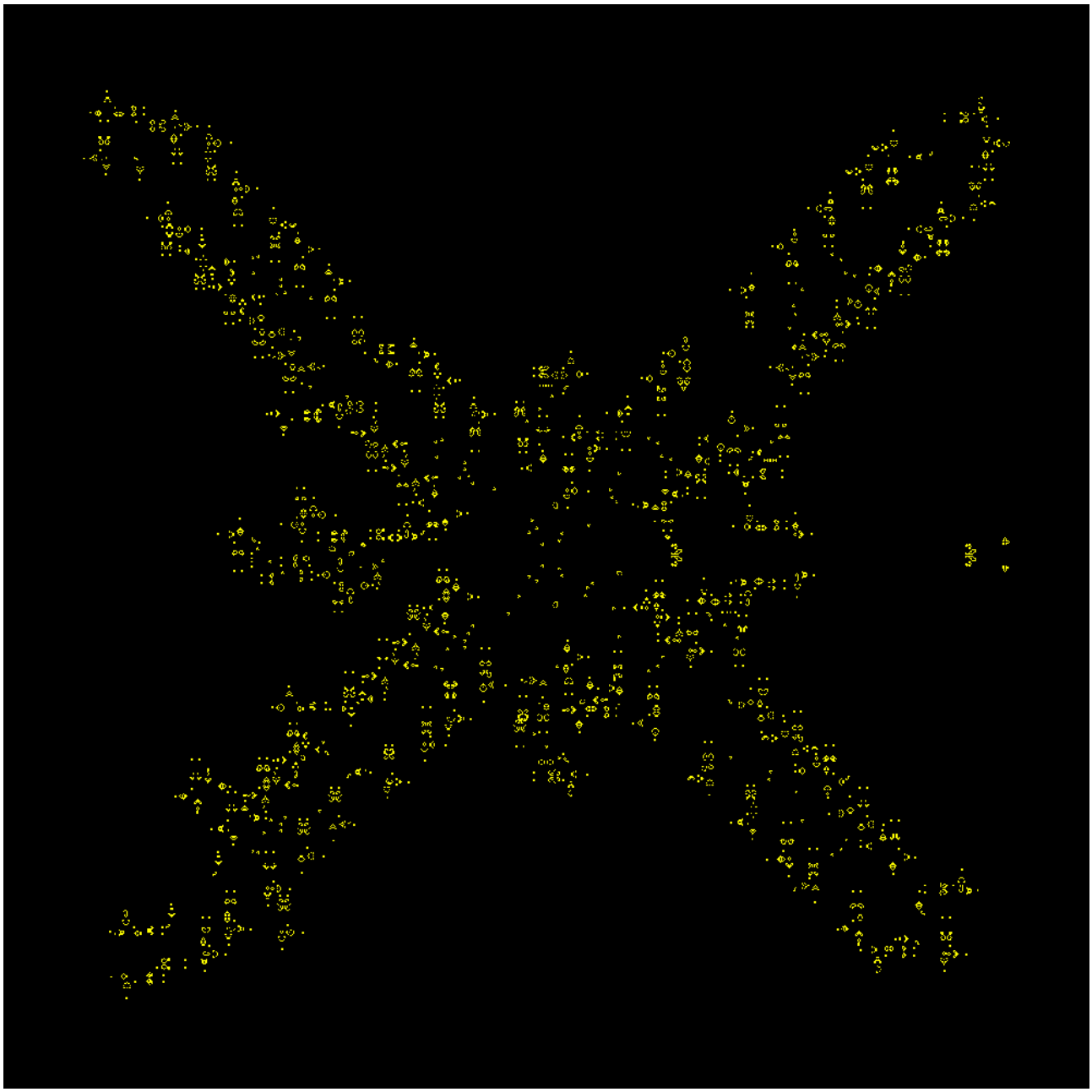


FIGURE 23 – Génération 565 du fichier rle/Guns/2c5-spaceship-gun-p690.rle d'une largeur de 1024 et générant des vaisseaux s'écrasant sur la structure à droite de l'écran.

9 Conclusion

On a pu voir lors de ce projet différentes façon d'implémenter le fonctionnement du même jeu à travers différents algorithmes et en utilisant différentes librairies.

On a ainsi pu se rendre compte lors de ce projet que pour ce genre de calculs l'implémentation sur GPU est la plus efficace de toutes les versions, cela est principalement du au fait que la grille de jeu se décompose facilement en un grand nombre de calculs parallèles.

De plus, les courbes de SpeedUp pour les implémentations utilisant OpenMP nous ont permis de nous rendre compte qu'il ne suffisait pas d'exécuter le code sur plusieurs cœurs pour que le programme tourne beaucoup plus vite. Par exemple, il n'est pas forcement possible de savoir immédiatement que la version utilisant OpenMP task allait être aussi inefficace pour un grand nombre de threads utilisées et une petite taille de tuile. Ainsi on se rend compte que de paralléliser du code est beaucoup plus difficile qu'il n'y parait au premier abord.

On peut, de plus, observer l'importance de la complexité des algorithmes dans le choix de l'implémentation. Cependant, il faut également prendre en compte quelle méthode de parallélisation est utilisé avec cette algorithme. En effet on se rend compte que la version optimisé est beaucoup plus rapide pour une utilisation séquentielle alors que ce n'est pas le cas pour l'implémentation sur GPU notamment à cause du fonctionnment de celui-ci.

Enfin, en implémentant la lecture des fichiers de Golly on a pu comparer la vitesse d'exécution de notre programme avec celle de ce dernier. Ce dernier utilise l'algorithme HashLife qui utilise des tables de hashages et des quadtrees pour essayer de trouver des patterns récurrents dans l'évolution de la configuration initiale du terrain de jeu et pour ainsi accélérer la vitesse des calculs ultérieurs. Par exemple ce dernier peut, sur une grille d'une taille de 30000×30000 cellules calculer les premières générations à une vitesse de quelques centaines de générations par seconde, puis monter à une vitesse de plusieurs centaines de milliers de génération dans une phase ou le pattern de base ne s'est pas encore répété. Une fois cette répétition passée, Golly est capable de calculer plusieurs centaines de milliards de générations par seconde car la période du pattern de départ à été déterminée et est facilement répétable. Golly n'utilisant jamais plus de 80% d'un cœur logique du processeur pour effectuer ses calculs.

La vitesse est incomparable à celle d'exécution de notre programme qui ne peut même pas se lancer car la machine ne possède pas assez de mémoire pour créer les deux grilles d'entrées et de sortie dans le but d'effectuer les différents calculs. Et quand bien même la grille est suffisamment petite (1024×1024 cellules) la version la plus rapide de notre programme n'est capable que de générer quelques milliers de générations à la seconde ce qui est très loin des performances de Golly.

On peut ainsi se rendre compte que bien au dessus de la parallélisation d'un algorithme, c'est la complexité de ce dernier qui restera le facteur clé dans sa rapidité d'exécution. Il est ainsi plus intéressant d'essayer d'améliorer la complexité d'un programme dans un premier temps avant de chercher à le paralléliser tant les différences de performances peuvent être colossales à cause de cette dernière.