Master Informatique – UE PSAR 2023-2024

Pixel War : France-Espagne saison 2

Etudiants : Mélissa Lacour et Noura Aljane

Encadrant : Swan Dubois

Table des matières

1 Introduction

Ce projet s'inspire de la Pixel War, la guerre des pixels. Le concept de cette guerre est le suivant : une toile blanche de taille fixe est mise à disposition des utilisateurs pour y ajouter des pixels de la couleur de leur choix, et ce de manière simultanée, l'objectif étant de créer le meilleur dessin. Le projet a pour but d'implémenter une structure de données thread-safe qui représente cette toile, de proposer plusieurs mécanismes de synchronisation de granularités différentes pour assurer un accès cohérent à la toile, puis d'étudier expérimentalement les performances de l'application en fonction du mécanisme de synchronisation choisi.

Dans ce document, nous commencerons par exposer le cahier des charges du projet, puis nous détaillerons la structure de données utilisée ainsi que les stratégies de verrouillage développées, enfin nous présenterons les expérimentations effectuées et les résultats obtenus.

2 Cahier des charges

Pour modéliser cette application, la consigne était d'utiliser une structure de données arborescente, dont les feuilles représentent les pixels de la toile, et qui serait construite pour que les pixels soient disposés dans l'arbre de telle manière à ce qu'ils puissent être regroupés hiérarchiquement selon leur position dans la toile (figure 1).

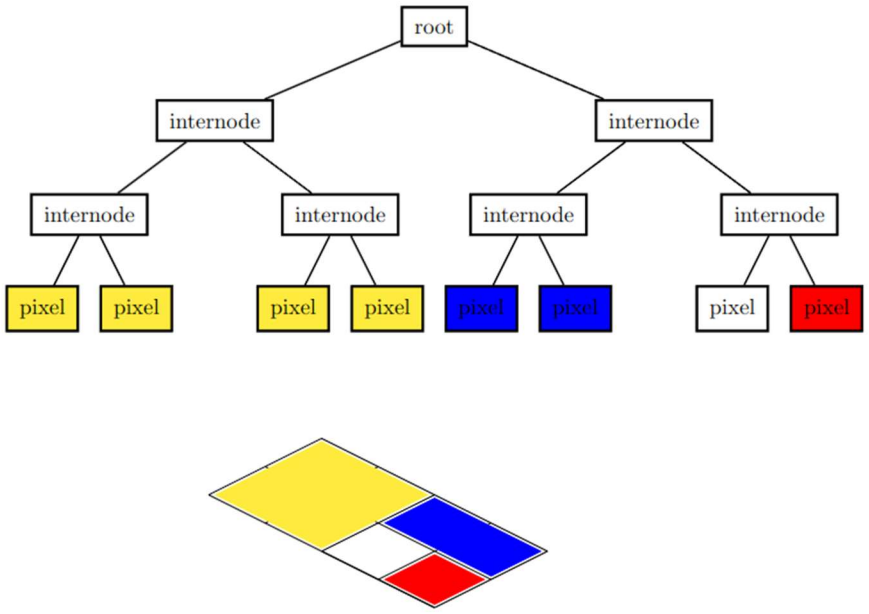

Figure 1 – Représentation de la structure arborescente et de la toile associée. Les zones colorées montrent comment les pixels sont organisés hiérarchiquement dans le même sous-arbre selon leur position dans la toile.

Les threads doivent modifier la toile en posant des tuiles, autrement dit des ensembles de pixels de taille donnée, et la pose d'une telle tuile doit être atomique. Pour assurer cette atomicité de pose de tuile, il était demandé de développer trois stratégies de verrouillage de granularités différentes, qui sont les suivantes :

1. La première stratégie, que nous appellerons la stratégie **GiantLock**, est celle ayant la granularité la moins fine. Elle consiste à verrouiller l'ensemble de l'arbre pour déposer la tuile.

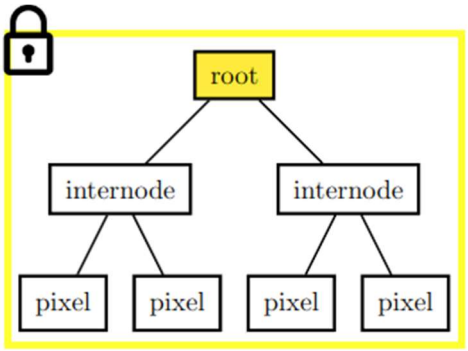

Figure 2 – Schéma logique du verrouillage dans la stratégie GiantLock.

2. La deuxième stratégie, nommée **InterLock**, est de granularité intermédiaire, et consiste à trouver dans l'arbre le nœud le plus bas dont le sous-arbre contient la tuile, et de le verrouiller pour poser la tuile.

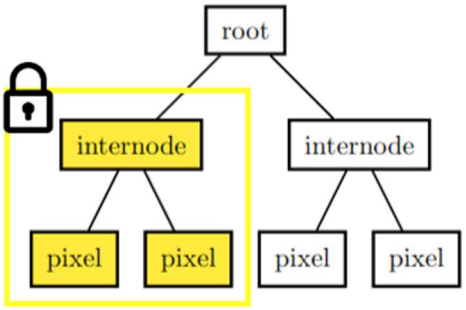

Figure 3 – Schéma logique du verrouillage dans la stratégie InterLock.

3. La troisième et dernière stratégie, nommée **PixelLock**, est celle qui a la granularité la plus fine et consiste à verrouiller individuellement tous les pixels de la tuile avant de poser celle-ci.

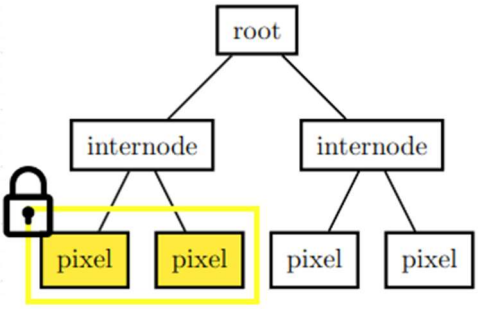

Figure 4 – Schéma logique du verrouillage dans la stratégie PixelLock.

Une fois ces fonctionnalités implémentées, il était demandé de déterminer des métriques pour évaluer les performances de l'application, ainsi que des paramètres qui pourraient influer dessus, afin de déterminer comment ces paramètres et le choix de la stratégie de verrouillage influent sur les performances de l'application.

3 Présentation succincte de l'archive et du code

Le langage de programmation utilisé pour ce projet est Java. Ce choix a été motivé par les différents avantages que propose ce langage et en particulier les nombreux mécanismes proposés pour la programmation multi-thread, la gestion de la mémoire implicite, ainsi que le paradigme objet qui nous a été très utile dans ce projet afin de factoriser le code commun des trois stratégies et d'éviter des duplications de code.

3.1 Descriptif des fichiers sources

Afin que le lecteur puisse s'y retrouver facilement dans le contenu du répertoire de code (dont voici le lien : https://github.com/nouraalj/pixelwar), voici un résumé rapide de ce qu'il contient :

- Package `pixelwar.tree` : contient la définition de la structure de données utilisée pour représenter la toile, ainsi que les fonctions permettant de la manipuler.
- Package `pixelwar.strategy` : contient les implémentations des stratégies de synchronisation.
- Package `pixelwar.experiment` : contient le code des expérimentations menées pour évaluer les performances de l'application.
- Package `pixelwar.drawing` : contient le code qui sert à poser des tuiles sur la toile.
- Package `pixelwar.utils` : contient quelques fonctions utilitaires.

3.2 Descriptif des fichiers exécutables

Tests unitaires. Le package `pixelwar.tests` contient des tests unitaires qui permettent de vérifier le bon fonctionnement de la structure de données, des stratégies de synchronisation, et de la pose des tuiles. Afin de faciliter la lecture des résultats des tests, nous invitons le lecteur à se reporter aux figures 7 et 8 qui représentent un arbre de la même taille que celui qui est utilisé dans nos tests.

Expérimentations. Le package `pixelwar.experiment` contient les différentes expérimentations menées. Les résultats sont produits dans des fichiers texte dans le répertoire `data`. Il est possible de les visualiser graphiquement grâce aux exécutables fournis dans le répertoire `data/time` (resp. `data/pixelSum`) pour l'expérience sur le délai d'attente (resp. sur le débit de pixels posés). Sur Windows, les exécutables sont `time.bat` et `pixelSum.bat`. Sur Linux/MacOs, les exécutables sont `time.sh` et `pixelSum.sh`.

4 Conception de la structure de données

4.1 Implémentation de la toile

Nous avons choisi de modéliser la toile par un arbre binaire. Dans le but de pouvoir découper de manière simple la toile en zones hiérarchiques, comme énoncé dans le cahier des charges, cet arbre doit être un arbre parfait : tous ses niveaux doivent être complets, pour que les zones aient strictement la même taille. Par souci de simplicité, nous avons choisi de représenter uniquement des toiles carrées. Afin de respecter ces contraintes sur la forme parfaite de l'arbre et sur la forme carrée de la toile, il n'est donc possible de construire que des arbres représentant des toiles de dimension $2^k * 2^k$ pixels.

Les figures 5 et 6 montrent comment l'arbre découpe la toile. Dans cet exemple, les deux sous-arbres de la racine séparent la toile horizontalement en deux zones de taille égale. Ensuite, les nœuds du niveau suivant divisent à nouveau ces deux zones mais cette fois-ci horizontalement. Et ainsi de suite,

en alternant des divisions verticales et des divisions horizontales. De manière générale, tous nos arbres sont divisés en zones de cette manière, la seule variation est qu'au niveau de la racine, certains seront d'abord divisés verticalement tandis que d'autres seront d'abord divisés horizontalement (selon que la hauteur de l'arbre est paire ou impaire).

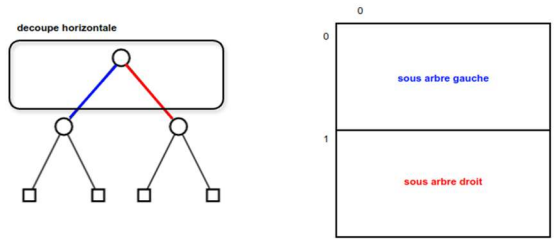

Figure 5 – Découpage horizontal de la toile.

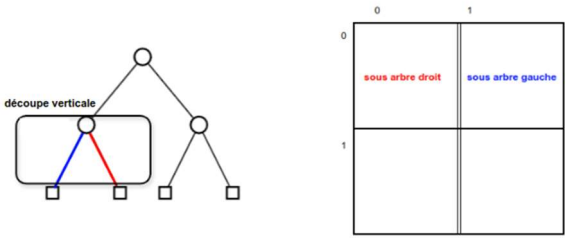

Figure 6 – Découpage vertical de la toile

L'arbre utilisé est constitué comme suit : de nœuds internes, et de feuilles, qui représentent les pixels de la toile.

Les pixels sont identifiés de deux manières. Ils sont tout d'abord identifiés par un nombre entier, dont les bits correspondent au chemin à parcourir pour aller de la racine de l'arbre jusqu'à ce pixel (figure 7). Cet identifiant est calculé lors de la construction de l'arbre, qui se fait de manière récursive : à chaque nouvel appel récursif, on passe un paramètre de type `int` auquel on ajoute à droite un bit valant 1 si l'appel récursif concerne un sous-arbre droit, et un bit valant 0 si l'appel récursif concerne un sous-arbre gauche. Une fois arrivé au dernier niveau de la construction de l'arbre, qui consiste à ajouter les pixels en tant que feuilles de l'arbre (cas terminal de la récursion), on attribue au pixel ce nombre entier comme étant son identifiant.

Cet identifiant permet de situer hiérarchiquement les pixels les uns par rapport aux autres ; en particulier, il permet de retrouver dans l'arbre l'ancêtre commun d'un groupe de pixels, grâce au préfixe commun entre leurs identifiants. Pour retrouver cet ancêtre commun, nous avons utilisé le principe suivant :

```
int i = 1 /* poids courant */
Pixel[] pixels /* liste des pixels du groupe */
int ref = bit de poids i du premier pixel de la liste
int[] prefixe /* tableau qui contiendra les bits du préfixe */

Tant que i < hauteur de l'arbre :
     Pour chaque pixel de la liste, comparer le bit de poids i
de son identifiant à ref
     Si ils sont différents :
          Retourner prefixe
     Sinon :
          prefixe[i] = ref
     i++
Retourner prefixe
```

Les pixels sont également identifiés par une paire de coordonnées entières `(x,y)` qui nous permet de les situer dans un plan, ce qui est plus représentatif d'un canevas, et nous permet notamment de choisir des tuiles aléatoirement et de représenter la toile facilement (figure 8).

Dans les deux cas, nous pouvons retrouver les pixels à partir de leur identifiant ou de leurs coordonnées avec une complexité temporelle en Θ (log N), où N est la taille de l'arbre en nombre de nœuds.

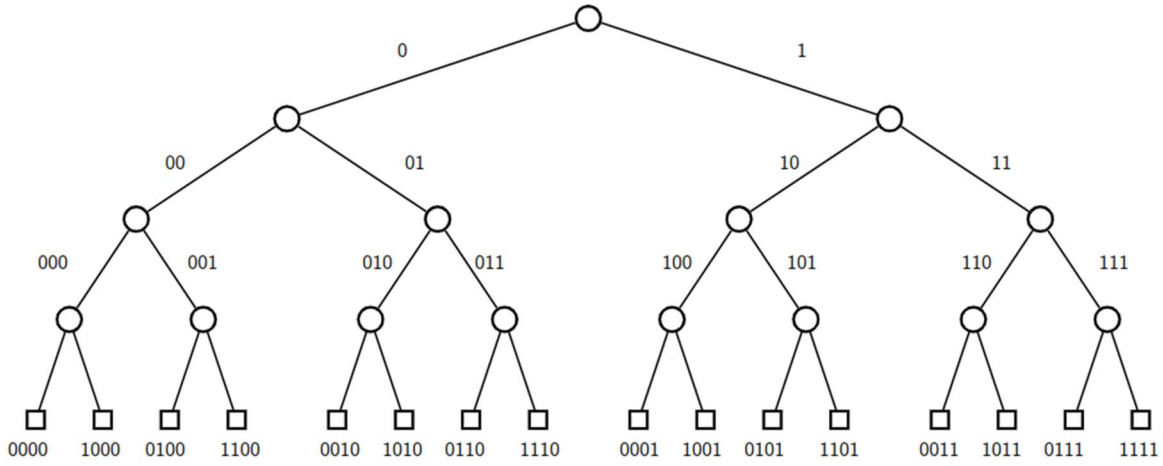

Figure 7 – Identification hiérarchique des pixels (exemple avec une toile de dimension 4x4 pixels). A noter que le bit de poids faible d'un identifiant correspond à la branche la plus proche de la racine, et le bit de poids fort à la branche la plus proche du pixel.

0	1	2	3
pixel 1010 x = 0 y = 0	pixel 1110 x = 1 y = 0	pixel 1011 x = 2 y = 0	pixel 1111 x = 3 y = 0
pixel 0010 x = 0 y = 1	pixel 1001 x = 1 y = 1	pixel 0011 x = 2 y = 1	pixel 0111 x = 3 y = 1
pixel 1000 x = 0 y = 2	pixel 1100 x = 1 y = 2	pixel 1001 x = 2 y = 2	pixel 1101 x = 3 y = 2
pixel 0000 x = 0 y = 3	pixel 0100 x = 1 y = 3	pixel 0001 x = 2 y = 3	pixel 0101 x = 3 y = 3

Figure 8 – Identification des pixels par leurs coordonnées sur un plan (exemple avec une toile de côté 4x4 pixels).

4.2 Implémentation des tuiles

Nous avons modélisé une tuile par un carré de pixels, choisi aléatoirement de telle sorte qu'il soit inclus dans les dimensions de la toile. Pour laisser une trace de la pose d'une tuile par un thread, chaque pixel de la tuile a un champ `owner` qui indique quel est le dernier thread qui l'a posé.

5 Implémentation des stratégies de verrouillage

Dans cette section, nous allons détailler les trois stratégies de verrouillage telles que nous les avons implémentées ainsi que justifier de leur bon fonctionnement.

5.1 Problématiques de concurrence

Afin que les tuiles soient posées de manière cohérente par les threads, les stratégies de verrouillage doivent assurer que la pose des tuiles respecte les propriétés suivantes :

1. La pose de la tuile doit être atomique
2. Il ne peut pas se produire d'interblocage
3. Il ne peut pas se produire de famine

La propriété 1 se résume à garantir la sûreté, et les propriétés 2 et 3 à garantir la vivacité.

5.2 Stratégie GiantLock

La stratégie GiantLock propose comme son nom l'indique un verrouillage de type « giant lock ». Pour accéder à la toile, les threads verrouillent l'arbre en entier au moyen d'un unique verrou qui doit être utilisé pour tout accès à l'arbre. Voici le pseudo-code de l'algorithme que nous avons utilisé :

```
Verrouiller l'arbre
Poser la tuile
Déverrouiller l'arbre
```

Cet algorithme assure d'une part la sûreté car les tuiles ne sont posées qu'après verrouillage de l'arbre, donc un seul thread à la fois peut poser sa tuile. D'autre part, il assure la vivacité : étant donné qu'il n'y a qu'un seul verrou, il n'y a pas d'interblocage possible ; de plus, les threads ont tous la même priorité de réveil sur cet unique verrou donc il n'y a pas non plus de risque de famine.

5.3 Stratégie InterLock

La stratégie InterLock a pour but de proposer un verrouillage de granularité intermédiaire. Le concept est le suivant : pour poser une tuile `t`, un thread doit verrouiller le nœud le plus bas de l'arbre qui contient sa tuile dans son sous-arbre.

Un thread doit donc tout d'abord déterminer quel est le nœud `n` le plus bas dans l'arbre qui recouvre entièrement la tuile `t` qu'il veut poser – c'est-à-dire que `t` doit être contenue dans le sous-arbre de `n`, mais qu'il ne doit pas exister un autre nœud `n2` dans l'arbre, tel que `n2` contient aussi la tuile `t` dans son sous-arbre et tel que `h(n2) < h(n)` (`h` étant la fonction qui donne la hauteur d'un nœud).

Pour trouver le nœud cible à verrouiller, nous utilisons l'identifiant hiérarchique des pixels de la tuile. En comparant les identifiants de tous les pixels, nous pouvons déterminer le chemin vers la racine du sous-arbre qui les contient tous, qui est le nœud cible que l'on cherche : ce chemin est défini par le préfixe binaire commun entre tous les identifiants (cf. section 4.1).

Une fois le nœud cible `n` trouvé, le thread doit le verrouiller afin de poser sa tuile, et cela tout en s'assurant de deux choses :

1. d'une part, il doit s'assurer que `n` n'est pas dans le sous-arbre d'un autre nœud qui aurait été verrouillé par un autre thread pour poser une tuile dans ce sous-arbre (phase 1 de l'algorithme).

2. d'autre part, il doit vérifier qu'aucun autre thread n'est déjà en train de poser une tuile dans le sous-arbre de `n` (phase 2 de l'algorithme).

Voici le pseudo-code de l'algorithme que nous avons utilisé, qui est expliqué en détail plus bas :

```
Calculer le chemin c vers le nœud cible n à verrouiller
Soit le nœud cur tel que cur = racine de l'arbre, ainsi que le nœud next = null
/* Phase 1 */
Verrouiller cur
Tant que cur != n :
    next = prochain nœud sur le chemin c
    Verrouiller next
    Déverrouiller cur
    cur = next
/* en sortant de la boucle, le nœud cible n est verrouillé (c'était le dernier nœud référencé par cur) */
/* Phase 2 */
Tant que le sous-arbre de n n'est pas complètement libre :
    Vérifier tous les nœuds du sous-arbre un par un, s'endormir sur le 1er nœud verrouillé trouvé
Poser la tuile
Déverrouiller n
```

Les deux conditions citées plus haut ont besoin d'être vérifiées avant de poser une tuile afin d'assurer que la tuile `t` que l'on pose n'est pas en superposition avec une autre tuile déjà en train d'être posée, que ce soit une plus petite tuile incluse dans `t`, ou une plus grosse tuile dans laquelle `t` est incluse, ou une tuile partiellement superposée à t (ce cas d'intersection se gère de la même manière que les deux cas précédents d'inclusion, en effet lors de la synchronisation, une tuile est assimilée au sous-arbre du nœud cible de cette tuile, ce qui fait que les deux sous-arbres correspondant à deux tuiles partiellement superposées sont soit identiques, soit l'un inclus dans l'autre).

Pour garantir la première condition, nous avons choisi de procéder comme suit : avant de verrouiller le nœud cible `n`, le thread vérifie sur le chemin `c` depuis la racine jusqu'à `n`, qu'il n'y a pas de nœud déjà verrouillé (cela signifierait qu'un thread a déjà l'intention de travailler dans son sous-arbre). Pour garantir la cohérence de cette vérification, le thread verrouille successivement chaque nœud qu'il rencontre sur `c` selon le procédé suivant (figure 9) :

1. il verrouille le nœud courant `cur` qu'il est en train de vérifier
2. il verrouille le nœud `next` suivant sur le chemin
3. il déverrouille le nœud courant `cur`
4. le nœud suivant devient le nœud courant

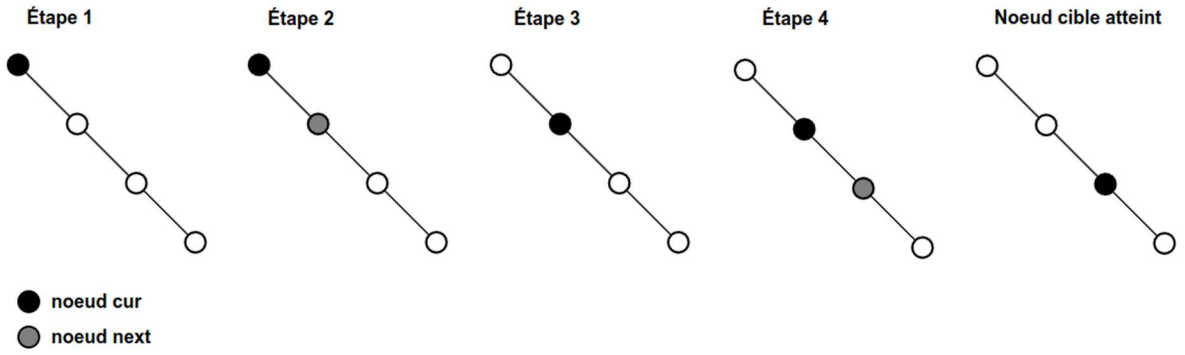

Figure 9 – Etapes de la vérification du chemin allant de la racine jusqu'au nœud cible. Les nœuds colorés (gris et noir) sont les nœuds verrouillés.

Ainsi, si un thread `t1` est commuté pendant qu'il effectue la vérification du chemin `c`, il reste une trace de son passage (i.e. un nœud `ni` verrouillé), et cela permet d'assurer la cohérence de la vérification. Par exemple, supposons qu'il y a un autre thread `t2` qui est élu et qui doit poser une tuile sur le même chemin `c`, mais au-dessus de `ni`. Si `t2` verrouille son nœud cible, il ne pourra pas poser sa tuile immédiatement car dans son sous-arbre il y a `ni` qui est toujours verrouillé et qui l'empêche de poser sa tuile tant que `t1` n'a pas terminé avec sa tuile (cf. phase 2). Si `ni` n'était pas verrouillé, `t2` commencerait à poser sa tuile, et si ensuite `t1` était réélu à ce moment-là, il reprendrait la vérification du chemin `c` là où il s'était arrêté, au niveau de `ni`, arriverait à son nœud cible et éventuellement commencerait à poser sa tuile, mais peut-être au même endroit et au même moment que `t2`, car chacun des threads ignore que l'autre thread travaille sur la même zone : l'atomicité de la pose de la tuile ne serait pas assurée. Le fait que `ni` reste verrouillé même si `t1` est commuté assure que tout autre thread arrivant après `t1` et voulant poser une tuile sur la même zone, restera bloqué sur `ni` tant que `t1` n'a pas terminé avec `ni`. De plus, pour les threads voulant travailler sur la même zone, cela instaure un ordre FIFO pour l'accès aux nœuds d'un même chemin : les threads qui arrivent en dernier restent bloqués sur ceux qui sont arrivés précédemment, et ce tout le long du chemin.

Pour garantir la deuxième condition, nous avons choisi de procéder comme suit : une fois le nœud cible `n` verrouillé, le thread vérifie dans le sous-arbre de `n` qu'il n'y a pas de nœud déjà verrouillé : sinon, cela signifierait que des threads arrivés antérieurement sont en train (ou en attente) de poser une tuile dans ce sous-arbre. Pour effectuer cette vérification de manière cohérente, le thread vérifie tous les nœuds du sous-arbre. Tant qu'ils ne sont pas tous libres, il se met en attente sur le 1er nœud non libre rencontré et à son réveil, il revérifie tout le sous-arbre. Si à un instant `t` aucun nœud n'est verrouillé dans le sous-arbre, cela signifie qu'il n'y a aucun thread ni en attente, ni en train de travailler dedans. En effet, notre algorithme garantit que tout thread voulant poser une tuile (ou en train de poser une tuile) a toujours au moins un nœud verrouillé quelque part qui signifie sa présence : soit le nœud cible pour sa tuile, soit un nœud sur le chemin entre la racine et le nœud cible (sauf dans le cas où il n'a pas encore réussi à verrouiller la racine, mais dans ce cas il ne peut pas poser de problème de modification non cohérente).

Par ailleurs, nous savons que l'attente du thread pour que le sous-arbre de son nœud cible soit complètement libre est d'une durée finie, car une fois le nœud cible `n` verrouillé, aucun autre thread ne peut postérieurement venir travailler dans le sous-arbre : il restera bloqué sur `n`. Ainsi, il y a un nombre fini de threads présents dans le sous-arbre, qui eux-mêmes vont terminer dans un temps fini.

Enfin, aucun thread voulant travailler au-dessus de `n` ne pourra commencer son travail s'il arrive postérieurement à `n`, car il sera bloqué sur `n` qui est verrouillé dans son sous-arbre.

Les explications ci-dessus justifient de l'atomicité de la pose d'une tuile et donc de la sûreté de l'algorithme. Les figures 10 et 11 illustrent des schémas des différents cas possibles de superposition de tuile, et montrent pourquoi la pose reste bien atomique.

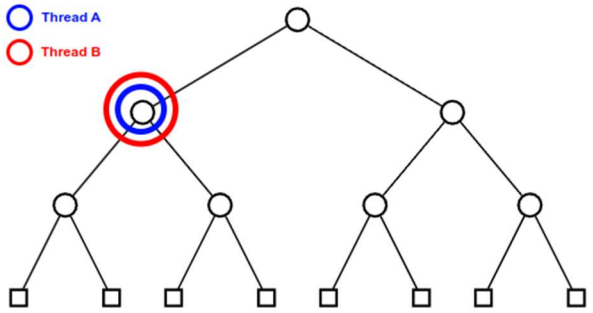

Figure 10 – Cas n°1 : Deux threads ont le même nœud cible. Le thread qui verrouille le nœud en 1er posera sa tuile en 1er.

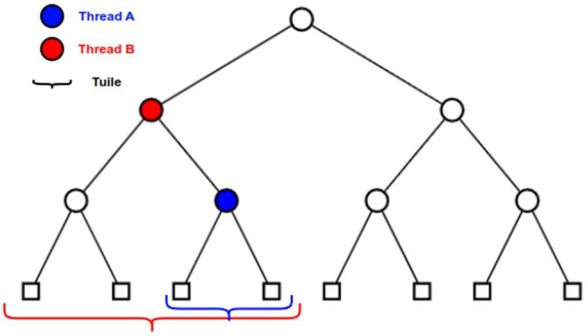

Figure 11 – Cas n°2 : le thread B arrive en 1er et verrouille non nœud cible (en rouge). Le thread A qui arrive ensuite, restera bloqué dans le chemin qui va à son nœud cible (en bleu) sur le nœud rouge jusqu'à que thread B le déverrouille. Cas n°3 : le thread A arrive en 1er et verrouille son nœud cible. Même si le thread B arrive ensuite à verrouiller son nœud cible, il sera bloqué par le nœud cible de A tant qu'il est verrouillé dans son sous-arbre.

Pour ce qui est de la vivacité, examinons en premier le cas de l'interblocage. Dans la première phase de l'algorithme, il ne peut pas y avoir d'interblocage car les threads acquièrent toujours les verrous du haut vers le bas de l'arbre le long d'un chemin : si un thread possède un verrou `v1` et cherche à acquérir un deuxième verrou `v2`, `v2` se trouve forcément plus bas que `v1` dans l'arbre. De plus, si un autre thread possède déjà `v2`, il est impossible que ce thread cherche à verrouiller `v1` car `v1` est au-dessus de lui sur le chemin. Donc il n'est pas possible que deux threads possèdent chacun un verrou et qu'ils veuillent chacun acquérir le verrou de l'autre.

Ensuite, à la fin de la première phase de l'algorithme, chaque thread possède exactement un verrou : celui de son nœud cible. Lors de la deuxième phase de l'algorithme, un thread va attendre uniquement sur des verrous situés dans son sous-arbre, mais jamais sur des verrous situés au-dessus de lui. Cela signifie que pour les threads qui sont actuellement dans son sous-arbre avec un (ou deux) nœuds verrouillés, c'est la même chose : ni dans la phase 1, ni dans la phase 2, ils ne peuvent chercher à verrouiller un verrou qui se trouve au-dessus d'eux. Donc, si un thread `t1` a verrouillé son nœud cible `n` et attend que les sous-nœuds de `n` soient tous libres, aucun des threads qui possèdent les verrous des nœuds de ce sous-arbre ne peuvent chercher à acquérir le verrou de `n`, et ainsi créer un cycle dans les acquisitions de verrous.

Par conséquent, nous pouvons en conclure qu'il n'est pas possible qu'il y ait un interblocage lors de l'exécution de l'algorithme.

Enfin, pour ce qui est de la famine, rappelons tout d'abord que tous les threads ont la même priorité pour l'acquisition des verrous. Ensuite, lors de la première phase de l'algorithme, nous savons que les threads sont ordonnés en ordre FIFO pour l'acquisition des verrous le long d'une branche. Cet ordre FIFO garantit qu'aucun thread ne peut se retrouver bloqué par une famine durant cette première phase, car il n'est pas possible qu'un autre thread le « double ». Par ailleurs, comme nous l'avons vu plus haut, lors de la deuxième phase de l'algorithme, le thread attend pendant un temps qui est fini, car il attend un nombre fini de threads qui eux-mêmes auront terminé leur exécution en un temps fini. Ainsi,

nous pouvons en conclure que cet algorithme ne peut pas produire de famine, et par conséquent que la vivacité est assurée lors de l'exécution.

5.4 Stratégie PixelLock

La stratégie PixelLock est celle qui propose la granularité de verrouillage la plus fine. Pour poser une tuile, un thread doit verrouiller individuellement les pixels qui composent la tuile. Dans cette stratégie, il a fallu en particulier faire attention à ne pas créer d'interblocage. Pour cela, nous avons utilisé l'algorithme suivant :

```
Ordonner la liste des pixels de la tuile par ordre d'identifiant croissant
Pour chaque pixel de la liste :
    Verrouiller ce pixel
Poser la tuile
Pour chaque pixel de la liste :
    Déverrouiller ce pixel
```

Dans cet algorithme, la vivacité est assurée. En effet, les pixels sont verrouillés selon un ordre total que tous les threads suivent : par ordre croissant d'identifiant. De plus, l'image n'est pas torique : il n'est pas possible d'avoir une tuile qui commence, par exemple, sur le bord droit de la toile et qui se termine sur le bord gauche. Ces deux conditions garantissent qu'il ne peut pas y avoir de cycle dans les acquisitions de verrous et donc qu'il ne peut pas y avoir d'interblocage (les figures 12 et 13 illustrent les cas potentiels d'interblocage et montrent pourquoi ils ne sont pas problématiques avec cet algorithme). De plus, les threads ont tous la même priorité pour l'acquisition des verrous donc il n'y a pas non plus de risque de famine.

Enfin, la tuile n'est posée que lorsque tous les pixels ont été verrouillés, ce qui garantit l'atomicité de la pose de la tuile et donc la sûreté

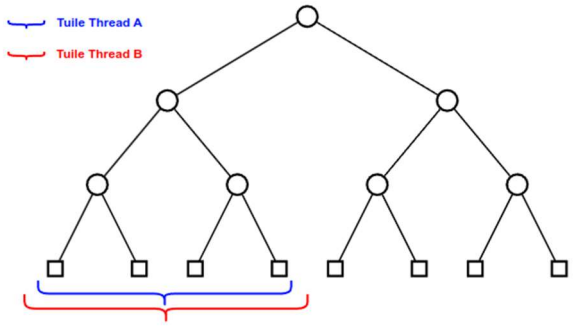

Figure 12 – Cas n°1 : les deux threads A et B veulent poser exactement la même tuile. Le premier arrivé verrouillera le premier pixel, grâce à l'ordre de verrouillage des pixels, le deuxième thread devra attendre que l'autre ait finit de poser sa tuile avant de pouvoir à son tour le premier pixel, puis les autres.

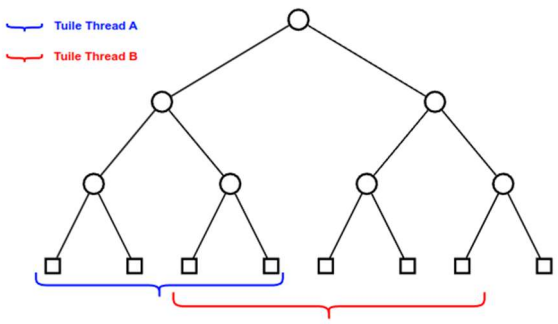

Figure 13 – Cas n°2 : Les threads A et B ont des tuiles qui se superposent. Les pixels en commun sont verrouillés par ordre croissant d'identifiant, ce qui garantit que le thread 1er à verrouiller un pixel en commun entre les deux tuiles bloquera l'autre thread, sans créer d'interblocage.

6 Expérimentations sur les performances de l'application

6.1 Métriques choisies pour mesurer la performance

Dans le but de pouvoir évaluer qualitativement les trois stratégies de verrouillage, nous avons déterminé des métriques qui permettent de mesurer les performances de l'application.

La première métrique que nous avons sélectionnée est le délai d'attente pour qu'un thread obtienne tous les verrous nécessaires à la pose d'une tuile (et que les conditions de pose de tuile soient réunies, si applicable).

Dans la stratégie PixelLock, il s'agit du temps écoulé entre le moment où le thread cherche à acquérir le verrou du premier pixel de la tuile, et le moment où le thread a obtenu le verrou du dernier pixel. Dans la stratégie InterLock, il s'agit du temps écoulé entre le moment où le thread cherche à acquérir le verrou de la racine, et le moment où il a acquis le verrou de son nœud cible et que son sous-arbre est complètement libre. Dans la stratégie GiantLock, il s'agit simplement du délai pour obtenir le verrou de l'arbre.

Cette mesure temporelle est essentielle pour évaluer la performance du mécanisme de verrouillage de l'application. Elle permet de déterminer quelle stratégie nécessite le moins d'attente pour la pose d'une tuile. Une durée d'attente élevée pourrait indiquer une mauvaise gestion de la synchronisation, par exemple une granularité de verrouillage trop faible ou trop élevée, ou bien des paramètres inadaptés (cf. section 6.2).

La deuxième métrique que nous avons sélectionnée est le débit de pixels posés par les threads. Pour cette métrique, nous mesurons le nombre de tuiles posées en un temps donné. Le but est de déterminer si une stratégie permet de poser plus rapidement des tuiles que les autres.

6.2 Paramètres à faire varier

Nous avons également déterminé quatre paramètres qui influent sur la performance de l'application. Le premier paramètre est la taille de la toile : nous avons voulu voir si une grande toile peut donner des résultats différents d'une petite toile. Le deuxième paramètre est la taille des tuiles, et en particulier la taille des tuiles par rapport à celle de la toile. Le troisième paramètre est le nombre de threads en train de travailler simultanément dans l'arbre, et enfin, le quatrième paramètre est la durée de l'expérimentation. Nous détaillerons plus précisément dans la section suivante comment nous avons fait varier ces paramètres, et ce pour obtenir quelles informations.

6.3 Expérimentations et résultats

Nous avons imaginé plusieurs expériences pour mesurer les performances de l'application, qui sont décrites ci-après. Pour chaque expérimentation, nous avons effectué les mesures avec les trois stratégies afin de pouvoir les comparer qualitativement.

6.3.1 Spécifications matérielles

Les expérimentations ont été menées sur un ordinateur disposant d'un processeur à 12 cœurs et d'une mémoire vive de 32 Go. Ces caractéristiques permettent de supporter des allocations mémoires assez lourdes et un haut niveau de parallélisme des threads. En effet, une capacité mémoire insuffisante pourrait provoquer un dépassement de pile lors de la construction d'un arbre de taille excessive.

Dans le cas de configurations matérielles moins puissantes, il est impératif d'ajuster les paramètres pour garantir une exécution fiable. Pour une mémoire vive inférieure à 32 Go, il est recommandé de réduire la valeur de la taille maximale du côté de la toile à 512 ou moins. De même,

pour un nombre de cœurs de processeur inférieur à 12, il est conseillé d'ajuster le nombre maximum de threads en conséquence, en optimisant ainsi l'utilisation des ressources matérielles disponibles.

6.3.2 Expérimentations sur le délai d'attente des threads

Pour chacune des expériences ci-dessous, nous avons fixé tous les paramètres sauf un que nous avons fait varier. Pour chaque combinaison de paramètres, nous avons lancé un pool de threads qui pose 100 tuiles, et mesuré pour la pose de chaque tuile le délai d'attente du thread. Ensuite, nous avons représenté les résultats dans des diagrammes en boîte à moustaches.

Dans les expérimentations de cette section, nous n'avons pas utilisé le paramètre correspondant à la durée de l'expérimentation car il n'a pas de sens dans ce contexte : il n'influe pas sur le délai d'attente des threads pour la pose d'une tuile.

Note 1 : Lorsque ce n'est pas le paramètre à faire varier, le nombre de threads est toujours fixé à 20.

Note 2 : Tous les délais dans les graphiques sont exprimés en nanosecondes

Expérimentation n°1. Dans cette expérimentation, nous avons fait varier la taille de la toile, allant d'une dimension de 8x8 pixels à une dimension de 4096x4096 pixels. La taille des tuiles était fixée à une taille de 6x6 pixels.

Analyse des résultats. Pour la stratégie PixelLock, nous pouvons remarquer (figures 14 à 16) que le temps d'attente diminue avec l'augmentation de la taille de la toile. Cela semble cohérent car les tuiles sont dispersées sur une plus grande surface et ont donc moins de chances d'être superposées (ce qui provoquerait de l'attente pour certains threads). En ce qui concerne la stratégie InterLock, on remarque que l'augmentation de la taille de la toile va de pair avec une légère augmentation du temps d'attente des threads, ce que nous avons du mal à expliquer. De son côté, la stratégie GiantLock ne semble pas affectée par la variation.

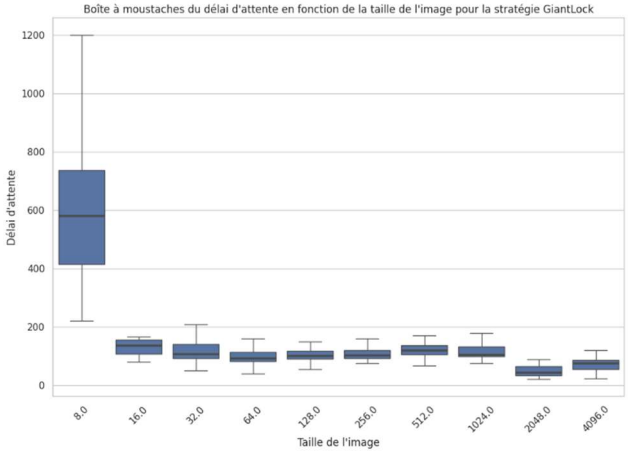

Figure 14 – Expérimentation n°1. Délai d'attente des threads en fonction de la taille de la toile, pour la **stratégie GiantLock**.

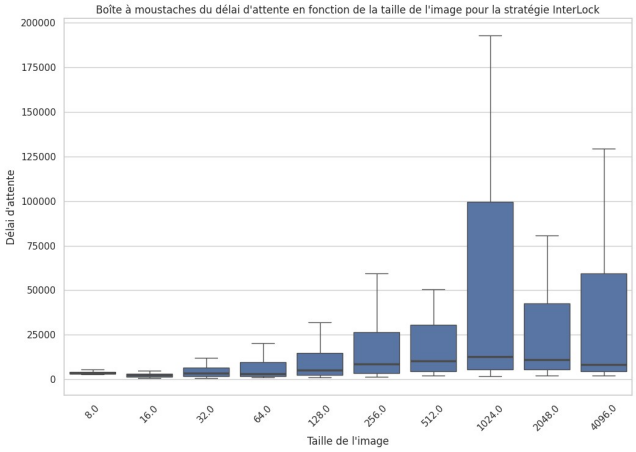

Figure 15 – Expérimentation n°1. Délai d'attente des threads en fonction de la taille de la toile, pour la **stratégie InterLock**.

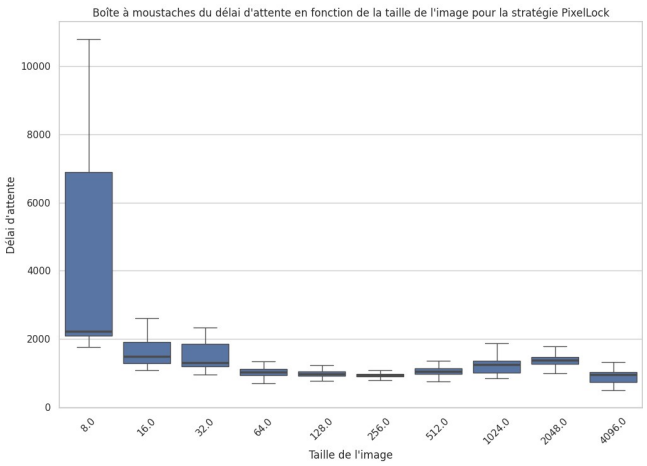

Figure 16 – Expérimentation n°1. Délai d'attente des threads en fonction de la taille de la toile, pour la **stratégie PixelLock**.

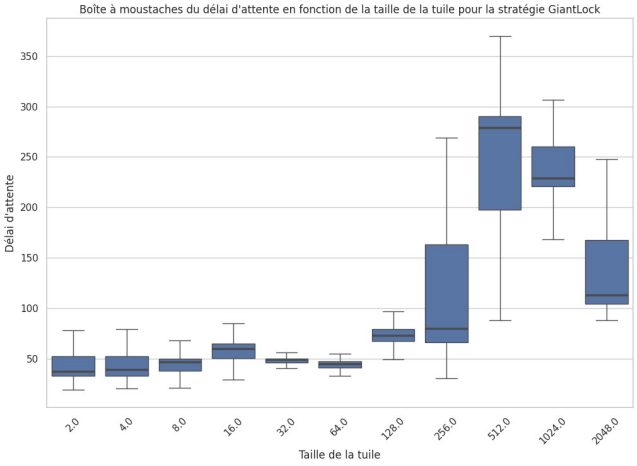

Figure 17 – Expérimentation n°2. Délai d'attente des threads en fonction de la taille de la tuile, pour la **stratégie GiantLock**.

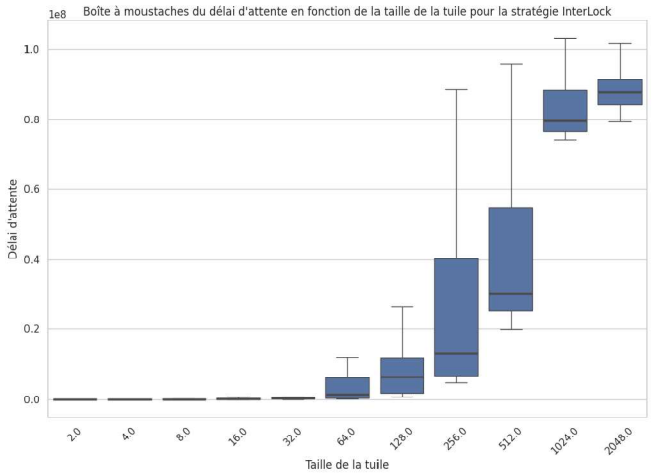

Figure 18 – Expérimentation n°2. Délai d'attente des threads en fonction de la taille de la tuile, pour la **stratégie PixelLock**.

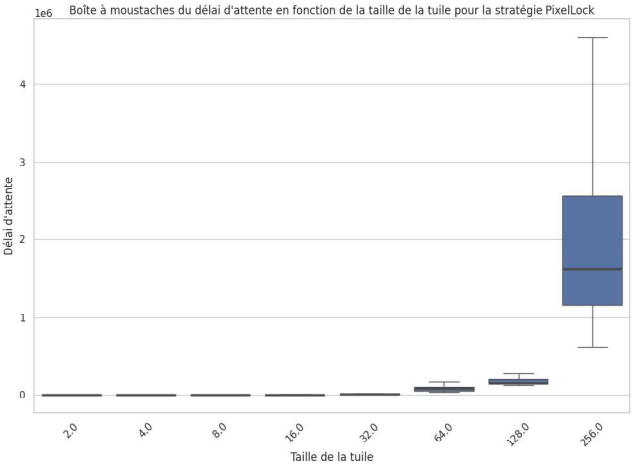

Figure 19 – Expérimentation n°2. Délai d'attente des threads en fonction de la taille de la tuile, pour la **stratégie InterLock**.

Expérimentation n°2. Dans cette expérience, nous avons fait varier la taille des tuiles, allant d'une dimension de 2x2 pixels à 256x256 pixels, ce qui représente des tuiles variant de 0.001% de la taille de la toile, à 25% de sa taille. La taille de la toile était fixée à 512x512 pixels.

Analyse des résultats. Pour toutes les stratégies, nous constatons que l'augmentation de la taille des tuiles provoque une augmentation du temps d'attente des threads (figures 17 à 19). Pour les stratégies InterLock et PixelLock, cela s'explique encore une fois par le fait que des tuiles plus grosses vont avoir davantage tendance à se superposer, ce qui cause de l'attente pour pouvoir les poser en exclusion mutuelle. Pour la stratégie GiantLock cela peut éventuellement s'expliquer par le fait qu'il y a un peu plus d'opérations à exécuter pour poser une plus grande tuile (proportionnel au nombre de pixels de la tuile).

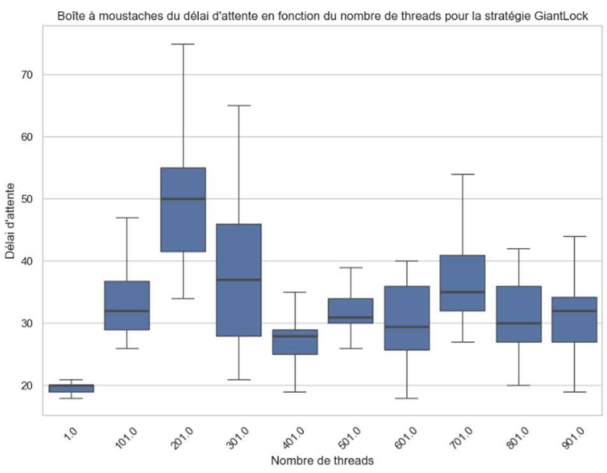

Figure 20 – Expérimentation n°3. Temps d'attente des threads en fonction du nombre de threads, pour la **stratégie GiantLock**.

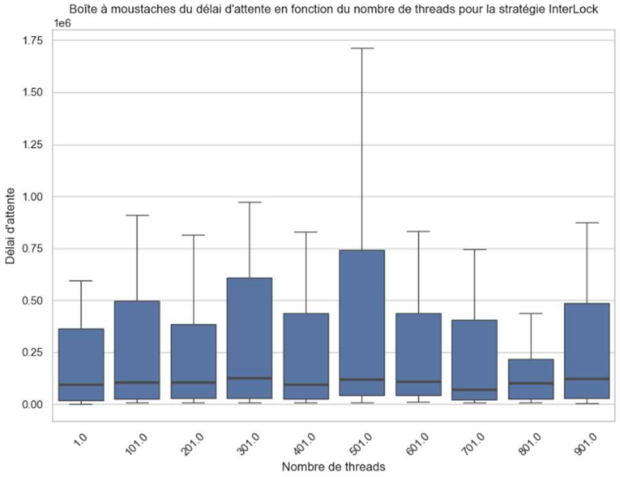

Figure 21 – Expérimentation n°3. Temps d'attente des threads en fonction du nombre de threads, pour la **stratégie InterLock**.

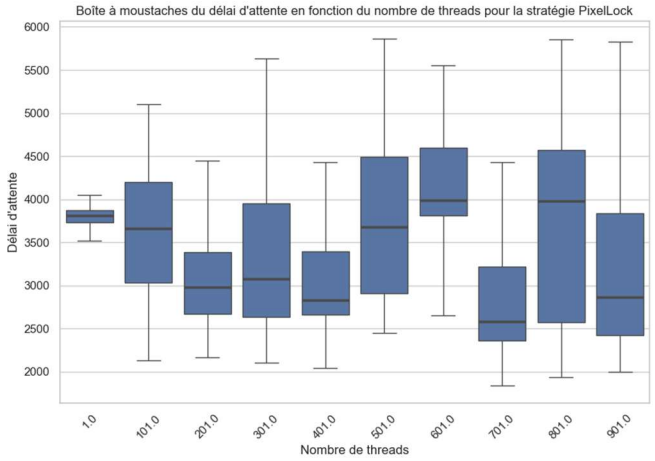

Figure 22 – Expérimentation n°3. Temps d'attente des threads en fonction du nombre de threads, pour la **stratégie PixelLock**.

Expérimentation n°3. La quatrième expérimentation pour cette métrique a consisté à faire varier le nombre de threads en train de travailler simultanément, allant de 1 à 1000 threads avec un pas d'augmentation égal à 100. La taille de la toile était fixée à 2048x2048 pixels et la taille des tuiles à 16x16 pixels.

Analyse des résultats. Les résultats (figures 20 à 22) montrent que pour les stratégies GiantLock et InterLock, le nombre de threads ne semble pas influer notablement sur le temps d'attente moyen des threads. Ils mettent cependant en avant le fait que dans la stratégie InterLock, il arrive que certains threads attendent beaucoup (les moustaches sont grandes), ce qui s'explique probablement par le fait qu'un thread dont le nœud cible est haut dans l'arbre, va parfois devoir attendre que la quasi-majorité des threads de l'arbre aient fini de travailler avant de pouvoir commencer son travail (ce qui ne peut pas arriver avec les stratégies GiantLock et InterLock). De plus pour la stratégie InterLock, les temps d'attente sont considérablement plus élevés que pour les autres stratégies (10^5 fois plus que pour la stratégie GiantLock !). Pour la stratégie PixelLock, le temps d'attente semble légèrement augmenter lorsqu'il y a davantage de threads.

Synthèse des résultats pour le délai d'attente des threads :

La conclusion qui peut s'appliquer aux trois stratégies est que l'augmentation de la taille de la tuile par rapport à celle de la toile cause une augmentation du délai d'attente. Pour les autres aspects, les résultats sont variés en fonction des stratégies.

Stratégie GiantLock. Les résultats mettent en avant le fait que cette stratégie ne semble pas sensible aux variations des paramètres, ce qui paraît cohérent : comme la toile en entier est en exclusion mutuelle, il n'y a pas de raison que la taille de la toile, la taille des tuiles ou le nombre de threads affecte sensiblement le délai d'attente des threads. On note également que pour la stratégie GiantLock, les threads attendent en moyenne beaucoup moins que pour les deux autres stratégies.

Stratégie InterLock. La stratégie InterLock est celle qui présente en moyenne le plus long délai d'attente. Cela s'explique par le fait que les threads ne doivent pas seulement attendre les threads qui ont une tuile superposée à la leur, mais également ceux qui ont une tuile *potentiellement* superposée à la leur (dans le même sous-arbre ou sur le même chemin). Nous avons pu remarquer que les plus grosses tuiles provoquent davantage de temps d'attente que les petites tuiles. Pour ce qui est des autres paramètres il est difficile de tirer des conclusions exactes d'après les résultats que nous avons obtenus, car ils ne semblent pas affecter les performances de manière concluante.

Stratégie PixelLock. Cette stratégie se place entre les deux autres en termes de délai d'attente. Son comportement est cohérent : le délai d'attente diminue avec l'augmentation de la taille de la toile (plus de surface pour disperser les tuiles), des tuiles très grosses par rapport à la surface de la toile nuisent au délai d'attente, et l'augmentation du nombre de threads cause une augmentation du délai d'attente.

6.3.3 Expérimentations sur le débit de pixels

Pour chacune des expériences ci-dessous, nous avons également fixé tous les paramètres sauf un que nous avons fait varier. Afin de mesurer le plus précisément possible le débit de pixels posés, nous avons procédé dans nos expérimentations comme suit : d'un côté, nous lançons un pool de threads qui posent chacun des tuiles. D'un autre côté nous lançons un thread qui est chargé de fermer le pool de thread et d'interrompre les tâches en attente, dès que la durée de l'expérimentation touche à sa fin. Pendant ce temps, les threads du pool posent des tuiles tant qu'ils ne sont pas interrompus. Les threads sont chargés de compter les tuiles qu'ils ont posées et d'ajouter ce compte à une variable partagée entre tous les threads. Cette variable partagée nous sert en fin de compte à savoir combien de tuiles ont été posées au total au cours de la durée de l'expérience. Cette fois-ci, nous avons représenté les résultats par des courbes simples.

Note : Dans les expériences où ils sont fixés, le nombre de threads est toujours fixé à 20 et la durée de l'expérimentation est toujours fixée à 4 secondes (dans les courbes qui suivent, les quantités de pixels posés correspondent donc au nombre de pixel posés en 4 secondes).

Expérimentation n°1. Dans cette expérimentation, nous avons fait varier la taille de la toile de 8x8 pixels à 4096x4096 pixels. La taille des tuiles était fixée à 6x6 pixels.

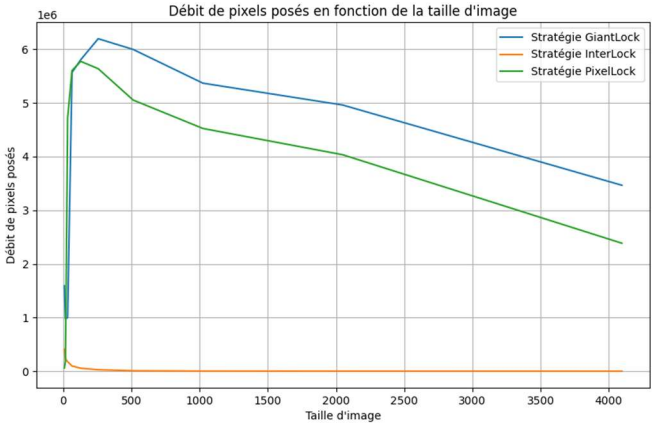

Figure 23 – Expérimentation n°1. Nombre de pixels posés en fonction de la taille de la toile.

Analyse des résultats. Pour les stratégies GiantLock et PixelLock, nous observons (figure 23) que jusqu'à une taille de toile de 256x256 pixels, le débit de pixels posés augmente très fortement, puis ensuite chute progressivement au fur et à mesure que la taille de la toile continue d'augmenter. Pour la stratégie PixelLock, ce résultat semble cohérent. En effet, une plus grande toile signifie que les tuiles ont moins de chances d'être tirées au même emplacement, et donc moins de chances que les threads doivent attendre pour obtenir leurs verrous (donc, plus de tuiles peuvent être posées en moins de temps). Cependant à partir d'une certaine taille, la complexité de certaines opérations (par exemple : parcourir l'arbre de la racine jusqu'au pixel, qui est une opération très fréquente dans notre application) peut prendre le dessus et abaisser la performance. Pour la stratégie GiantLock, l'augmentation initiale du débit semble un peu surprenante : aucun parallélisme n'est possible donc une plus grande toile ne devrait pas influer positivement sur le nombre de tuiles posées. Pour la stratégie InterLock, nous pouvons constater que la taille de la toile ne semble pas avoir d'impact sur le débit de pixels posés : il reste constant quelle que soit la taille de la toile. Ce résultat est surprenant, on pourrait s'attendre à ce qu'une plus grande toile permette de poser plus de tuiles, car elles ont moins de chances d'être tirées au même endroit ; d'un autre côté une plus grande toile induit également plus de nœuds à verrouiller / attendre, ce qui rajoute du temps pour la pose de chaque tuile.

Expérimentation n°2. La seconde expérience a servi à étudier l'effet de la variation de la taille de la tuile, celle-ci étant allée de 2x2 pixels à 16x16 pixels, soit par rapport à la taille de la toile, qui était fixée à 32x32 pixels, des tuiles allant de 0.4% à 25% de sa surface (figure 24). Une seconde exécution (figure 25), avec une toile de 1024x1024 pixels et des tuiles allant de 2x2 pixels à 512x512 pixels (soit jusqu'à 25% de la taille de la toile) a permis de confirmer les résultats.

Analyse des résultats. Nous pouvons constater que pour toutes les stratégies, le débit de pixels posés décroît très rapidement lorsque la taille des tuiles se rapproche de la taille de la toile. La décroissance semble exponentielle. Cela s'explique par le fait que les grosses tuiles réduisent le

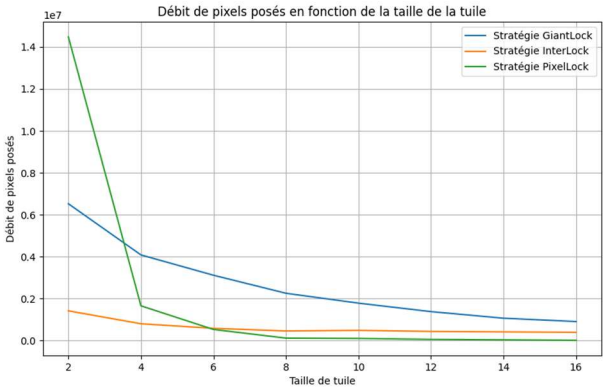

Figure 24 – Expérimentation n°2. Nombre de pixels posés en fonction de la taille de la tuile.

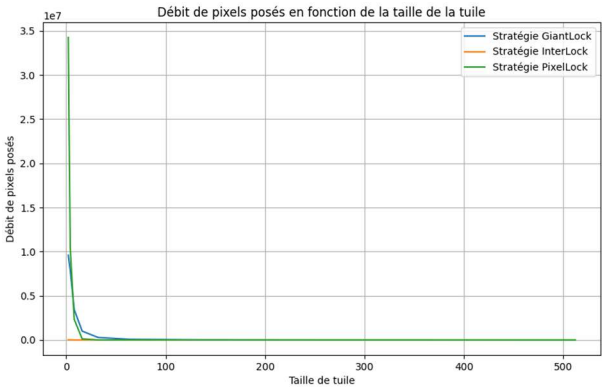

Figure 25 – Expérimentation n°2. Nombre de pixels posés en fonction de la taille de la tuile.

parallélisme, pour les raisons évoquées plus haut, et donc le nombre de tuiles qui peuvent être posées simultanément.

Expérimentation n°3. Cette expérience a eu pour but d'étudier l'effet de la variation du nombre de threads, allant de 1 threads à 140 threads. Les autres paramètres étaient fixés à 32x32 pour la taille de la toile et 4x4 pour la taille des tuiles.

Analyse des résultats. Les résultats (figure 26) montrent que les stratégies GiantLock et InterLock ne sont pas affectées par la variation du nombre de threads. Le résultat semble cohérent pour la stratégie GiantLock, mais il est surprenant pour la stratégie InterLock, nous avons du mal à l'expliquer. Pour la stratégie PixelLock, nous observons que le débit augmente avec le nombre de threads, puis stagne, ce qui semble cohérent car plus de threads qui travaillent en même temps permettent de poser plus de tuiles, pour peu que la taille des tuiles soit adaptée (autrement dit, pas trop grosse par rapport à la taille de la toile, ce qui est le cas ici car les tuiles font 1.6% de la toile) ; cependant après un certain nombre de threads les limitations physiques de la machine font que le parallélisme ne peut pas indéfiniment augmenter.

Expérimentation n°4. La quatrième et dernière expérimentation pour le débit de pixels, a consisté à faire varier la durée de l'expérimentation. Nous avons utilisé une toile de 32x32 pixels et des tuiles de 4x4 pixels. Nous avons testé ces paramètres avec des exécutions durant de 0.01 secondes à 5 secondes.

Analyse des résultats. Nous observons sur la figure 27 que pour les trois stratégies, le débit de pixels est linéaire à la durée de l'expérimentation. Ce graphique nous permet également de voir qu'en termes d'efficacité, la stratégie PixelLock est celle qui a le meilleur débit de tuiles posées, suivie par la stratégie GiantLock, et enfin en dernier par la stratégie InterLock.

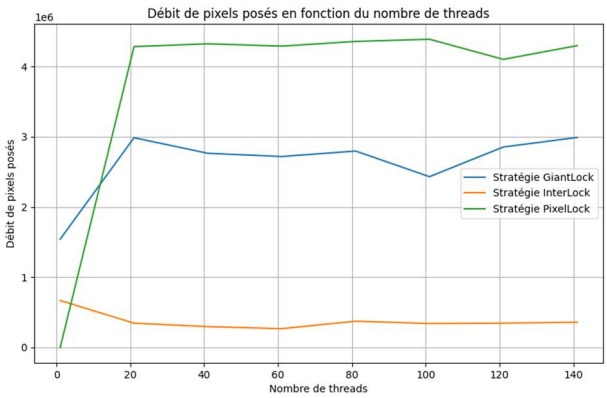

Figure 26 – Expérimentation n°3. Nombre de pixels posés en fonction du nombre de threads.

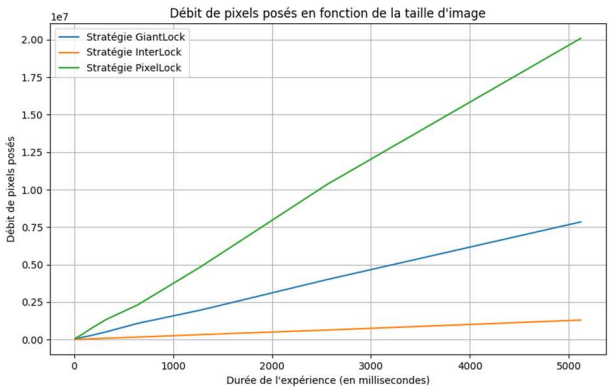

Figure 27 – Expérimentation n°4. Nombre de pixels posés en fonction de la durée de l'expérimentation.

Synthèse des résultats pour le débit de pixels.

La seule conclusion qui peut s'appliquer aux trois stratégies est que l'augmentation de la taille de la tuile par rapport à celle de la tuile cause une baisse de débit de pixels. Pour les autres aspects, les résultats diffèrent en fonction de la stratégie considérée.

Stratégie InterLock. Ces expérimentations mettent tout d'abord en avant les performances assez faibles de la stratégie InterLock. Cela peut s'expliquer par le fait que la pose d'une tuile est une action très peu coûteuse en termes d'opérations (il suffit de changer la valeur du champ `owner` de

chaque pixel de la tuile), alors que la stratégie InterLock implique de nombreuses opérations de synchronisation pour la pose d'une tuile, ce qui fait que le thread passe peut-être au final plus de temps à synchroniser sa pose de tuile qu'à effectivement poser la tuile. De plus, avec cette stratégie, les threads doivent parfois attendre d'autres threads qui posent des tuiles dans le même sous-arbre (ou qui suivent le même chemin), sans pour autant que leur tuile se superposent effectivement (ils n'ont pas de moyen de le savoir), ce qui rajoute du temps d'attente supplémentaire « inutile ». Cela laisse penser qu'une telle stratégie de synchronisation aurait été plus pertinente pour synchroniser des actions longues (plus longues que la pose de nos tuiles, et idéalement plus longues que le temps que le thread met à synchroniser l'action voulue).

De manière générale, les résultats présentés montrent qu'aucun paramètre n'influe sur ses performances de manière concluante.

Nous pouvons conclure de ces résultats que notre implémentation de la stratégie InterLock ne permet pas assez de parallélisme, en plus d'être trop coûteuse en calcul.

Stratégie GiantLock. Il semble que cette stratégie ne soit pas non plus très affectée par les variations de paramètres, mis à part celle de la taille de la tuile. Ce résultat est parfaitement cohérent, cette stratégie ne présentant pas de parallélisme, on ne voit pas pourquoi le nombre de threads ou la taille de la toile influerait sur le débit de pixels. Pour ce qui est de la taille de la tuile, le fait que ce paramètre influe légèrement s'explique peut-être par le fait qu'une plus grosse tuile prend un peu plus de temps à poser, et donc que l'augmentation de la taille de la tuile impacte négativement le débit de pixels. En conclusion, la stratégie GiantLock, malgré son manque total de parallélisme, fait quand même mieux que la stratégie InterLock.

Stratégie PixelLock. Cette stratégie est celle qui a le meilleur débit de pixels posés et semble donc être celle qui favorise le plus le parallélisme. Elle exécute en effet moins d'opérations de synchronisation pour la pose d'une tuile que la stratégie InterLock ; de plus un thread ne peut attendre que des threads dont la tuile est effectivement en superposition avec la sienne (contrairement à la stratégie InterLock). Nous observons que l'augmentation du nombre de threads et de la taille de la toile favorisent l'augmentation du débit de pixels, ce qui semble cohérent du point de vue du parallélisme : davantage de threads qui travaillent sur une surface plus grande (donc moins de chances de collisions sur l'emplacement des tuiles).

6.3.4 Synthèse globale des résultats

D'après nos résultats, nous pouvons conclure que la stratégie la plus performante pour notre application est la stratégie PixelLock.

La stratégie PixelLock, bien qu'elle ait un délai d'attente des threads assez élevé, a le meilleur débit de pose de pixels, car même si les threads attendent davantage en moyenne, ils peuvent poser des tuiles de manière parallèle.

La stratégie GiantLock, malgré un délai d'attente des threads significativement plus faible, n'a pas le débit de pose de pixels le plus élevé à cause du fait qu'elle ne laisse pas de possibilité de parallélisme.

Enfin, la stratégie InterLock a des performances assez faibles dans les deux domaines évalués : les threads attendent trop, et le parallélisme ne semble pas optimal non plus car le débit de pixels est le plus faible de tous. Cela peut s'expliquer par le fait que lorsque les pixels de la tuile se trouvent dans des sous-arbres trop éloignés, le préfixe commun est court et le nœud cible à verrouiller proche de la racine, ce qui bloque une grosse partie de l'arbre pour les autres threads. De plus, lors de la première phase de l'algorithme, un nœud temporairement verrouillé sur le chemin jusqu'au nœud cible va bloquer des threads qui ont peut-être juste le début du chemin en commun mais qui ensuite ne vont pas dans les

mêmes sous-arbres. Par exemple, un thread peut être bloqué sur un nœud verrouillé, dans l'intention d'aller dans le sous-arbre gauche, alors que le thread qui a le verrou du nœud attend peut-être pour aller dans le sous-arbre droit (leurs tuiles ne peuvent pas se superposer, mais un des threads doit quand même attendre l'autre).

Cette approche a pour avantage de ne pas provoquer de conflit d'interblocage, mais réduit tout de même beaucoup la possibilité de parallélisme entre les threads, à cela s'ajoutent les conditions de verrouillage d'un nœud qui diminuent la performance au niveau du temps.

De manière générale, nous pouvons également conclure de ces expériences que des tuiles trop grosses par rapport à la taille de la toile nuisent aux performances de l'application (séquentialisation de la pose des tuiles).

Enfin, pour la stratégie PixelLock, qui est celle qui fonctionne le mieux, les paramètres idéaux qui ressortent de nos expérimentations sont une grande toile et des tuiles proportionnellement pas trop grosses, afin de favoriser la dispersion des tuiles, et un nombre de threads plutôt élevé, afin de maximiser le parallélisme.

7 Conclusion

En fin de compte, la granularité de verrouillage qui semble la plus adaptée pour notre application est celle de la stratégie PixelLock, à savoir la granularité la plus fine. La stratégie InterLock qui promettait d'être un bon compromis, s'est révélée avoir beaucoup de défauts, qui ne sont pas nécessairement liés à la granularité de son verrouillage mais qui la rendent inadaptée pour cette application. La stratégie GiantLock qui ne paraissait pas idéale en termes de granularité de verrouillage a tout compte fait des performances correctes.

Une autre proposition de stratégie de verrouillage de granularité intermédiaire, plus élaborée et complexe à mettre en œuvre mais qui pourrait réduire les inconvénients décrits ci-dessus, serait d'effectuer un verrouillage par segment. La toile serait divisée en différents segments, de taille supérieure à un pixel mais pas trop élevée non plus, et on appliquerait un verrouillage sur les segments nécessaires à la pose de la tuile. Plutôt que de choisir un seul nœud cible pour l'ensemble de la tuile, plusieurs nœuds cible seraient calculés : un pour chaque segment à verrouiller. Cela permettrait davantage de parallélisme que dans notre stratégie InterLock, tout en ayant une granularité plus élevée que notre stratégie PixelLock.