Systèmes Distribués

Jeu de la vie A

Rapport de projet

Yann Trou

23 décembre 2021

Table des matières

Introduction

L'objectif de ce projet est de programmer le jeu de la vie de Conway. Il s'agit d'un automate cellulaire crée dans les années 70 ne nécesitant aucun joueur. Il calcule son état suivant à partir de son état précédent. Dans ce projet, il se déroule dans un environnement en trois dimensions parallélisé sur plusieurs machines.

1 Analyse

1.1 Généralités

Les règles utilisées pour calculer l'état des cellules sont les mêmes que celui du jeu original.
- Si une cellule possède moins de 4 voisins en vie ou plus de 5 voisins en vie, elle meurt.
- Si une cellule possède 5 voisins en vie, elle vivra
- Si une cellule possède 4 voisins en vie, elle conservera son état actuel.

Bien que les règles soient similaires, les seuils de vie et mort sont différents des originaux, 2333. En effet, les seuils originaux appliqués dans un contexte en 3D conduisent à une croissance incontrôlable des cellules. D'après les réflexions de Carter-Bays , on peut retenir deux sets de seuils pouvant conduire à des ensembles stables : 4555 et 5766

Pour la structure de l'environnement, plusieurs approches sont possibles, on en retiendra deux.

Une approche matricielle considérée naive, où chaque cellule est représentée dans un tableau à trois dimensions. Cette approche sera considérée comme référence.

Une approche par Hashage, où l'on a un HashSet contenant les cellules en vie, et une HashMap qui contient des paires cellule-nombre de voisins.

Dans la première approche, on représente l'état d'une cellule par un booléen avec true = en vie, false = mort. Elles sont contenues dans un tableau en trois dimensions. Pour calculer leur état, on itère sur l'intégralité du tableau. Pour chaque cellule, on compte son nombre de voisins en vie. On soumet ce nombre aux règles énoncées ci-dessus. Ensuite, on met son nouvel état dans un autre tableau représentant l'environnement futur. Une fois que toutes les cellules sont traitées, l'environnement futur devient l'environnement courant et le cycle peut recommencer. On utilise deux environnements afin d'éviter de fausser les calculs d'état.

Dans la seconde approche, une cellule est représentée par un entier équivalent à sa position. Si une cellule est en vie, sa position sera stockée dans un HashSet. Les cellules mortes ne sont pas représentées. L'environnement autour des cellules en vie, plus précisément le nombre de voisins autour des cellules en vie, est stocké dans une HashMap avec la position de la cellule en clé et son nombre de voisins en valeur.

Le calcul d'une génération se fait en quatre étapes :
- Pour chaque cellule en vie, si son nombre de voisins l'indique comme morte, on l'ajoute à la liste des cellules à supprimer.
- Pour chaque cellule dans la liste des voisins, si son nombre de voisins l'indique comme vivante, on l'ajoute à la liste des cellules à créer.
- Pour chaque cellule parmi les cellules à créer, on l'ajoute à la liste des cellules vivantes et pour chacun de ses voisins, s'il est déjà répertorié, on incrémente sa valeur, sinon on le crée avec une valeur de 1 dans la HashMap
- Pour chaque cellule parmi les cellules à détruire, on la supprime des cellules vivantes et pour chacun de ses voisins, on décrémente sa valeur. Si la valeur du voisin est inférieure ou égale à zéro, on supprime le voisin.

1.2 Adaptation en paradigme distribué

Cet algorithme est orienté calcul plutôt que messages. Java RMI et MPI sont des approches viables pour ce type de problématique. On retient Java RMI avec un pattern Bag Of Task, seul le serveur aura conaissance de la nature de la tâche calculée sur le client. On part du principe que les données vont aux processus et pas l'inverse.

Au niveau de la parallélisation des algorithmes, pour l'approche matricielle, une tâche correspond à devoir calculer l'état futur d'un sous-ensemble de cellules.

Pour l'approche par hashage, une tache correspond à, si l'on est à la première étape, lister les cellules mortes du sous ensemble des cellules en vie. sinon, elle correspond à devoir trouver les cellules à créer dans un sous-ensemble de la HashMap de voisinnage.

Les étapes trois et quatre ne sont pas effectuées en parallèle car il est impossible de reconstituer la liste des cellules en vie et la liste de voisinage. une HashMap ayant un ordre de placement aléatoire, une tâche pourrait indiquer qu'un voisin est supprimé alors alors qu'une autre pourrait indiquer que non. Elles consistent à créer et détruire les cellules devant l'être puis à mettre a jour leurs voisins.

2 Spécifications

L'application va être organisée en trois grandes parties :
- une partie pour le jeu de la vie (package core)
- une partie pour le rendu graphique (package engine)
- une partie pour la gestion d'évènements (package Engine.Events)

2.1 Core

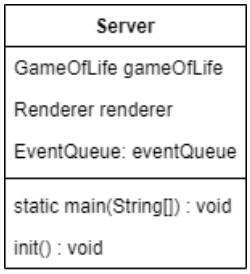


FIGURE 1 – Server : point de démarrage de la partie serveur. Instancie le jeu de la vie et le rendu si nécessaire. Configure l'application en fonction des paramètres donnés par l'utilisateur.

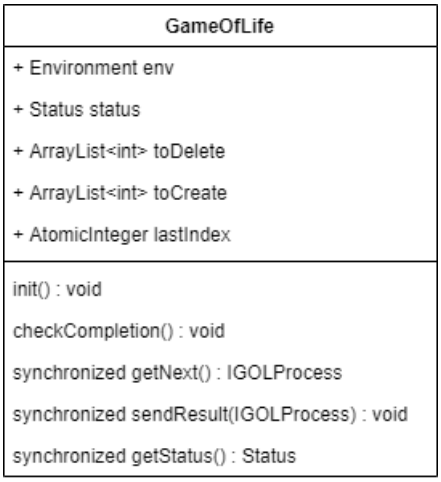


FIGURE 2 – GameOfLife / IGameOfLife : gère la distribution et récupération des taches, ainsi que la progression du traitement. Si le traitement se termine, il lancera une nouvelle génération.

GenerateProcess : Processus indiquant, pour un sous-ensemble de voisins et la liste des cellules en vie données, quelles cellules doivent être crées

PurgeProcess : Processus indiquant, pour un sous-ensemble de cellules vivantes et la liste des voisins, quelles cellules doivent être détruites

2.2 Engine

Renderer : gère le rendu graphique. initialise tous les composants nécessaires (shaders, fenêtre) et boucle pour mettre à jour l'affichage. Gère les entrées clavier/souris, la caméra, les évènements reçus pour l'affichage,

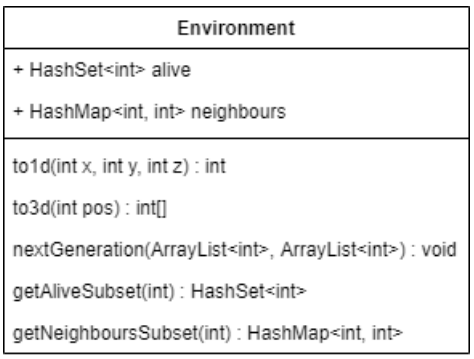


FIGURE 3 – Environment : gère les listes de cellules. Crée des sous-ensembles à la demande de GameOfLife pour des tâches à traiter. Génère les configurations initiales et les changements de génération.

SpriteManager : gère la liste des objets à afficher. contient une grille de cubes à l'échelle de l'environnement du jeu de la vie. Un objet GameOfLife peut appeller la méthode update (via l'instance statique) pour actualiser la liste des cellules (cubes) affichés / masqués à l'écran.

TextureAtlas : gère les textures pour les objets. Les textures sont stockées dans une HashMap avec un TextureID comme clé afin d'économiser de la mémoire et centraliser les données. Quand le Renderer veut afficher la texture d'un objet, il demande au TextureAtlas de lui activer la texture nécessaire.

2.3 Events

EventDispatcher : plaque tournante pour les évènements. Il contient une HashMap regroupant les EventQueues actives. Elles sont associées à un ThreadID unique servant d'adresse. Quand une EventQueue veut envoyer un message, c'est l'EventDispatcher qui fera la distribution vers une autre EventQueue qui gérera la réception.

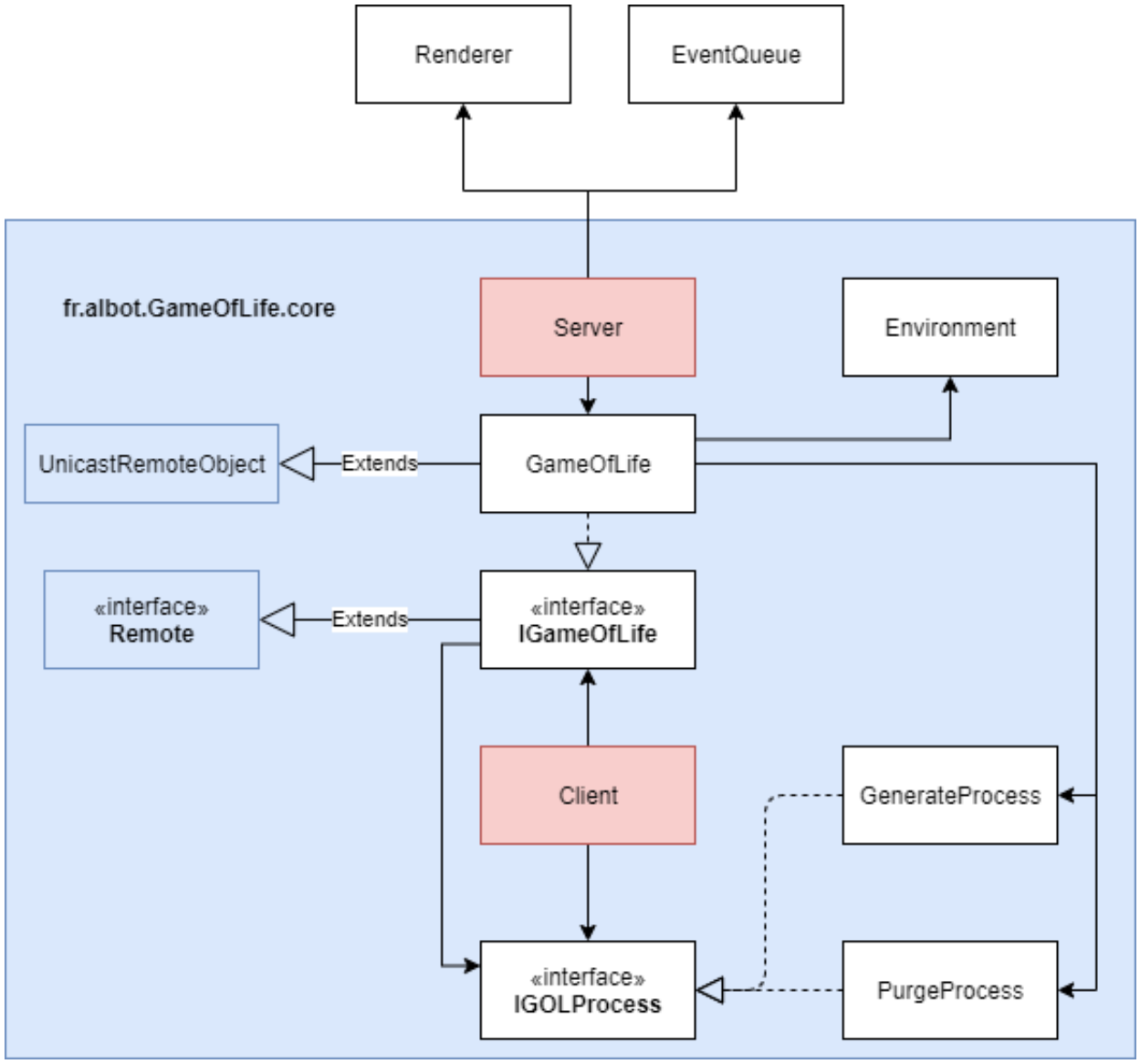
EventQueue : Objet qui gère la réception / l'envoi de messages. Quand un objet veut envoyer un message, c'est elle qui le distribue au dispatcher. Aussi, c'est elle qui sert de boite aux lettres. Comme c'est une Queue, les messages sont lus dans l'ordre de leur arrivée.

2.4 Config

le package contient la classe CONFIG, regroupant un ensemble de valeurs de configuration pour le rendu et le traitement. Certaines des valeurs sont finales pour éviter la modification alors que d'autres peuvent être altérées au besoin.

3 Architecture

3.1 Organisation du package core.

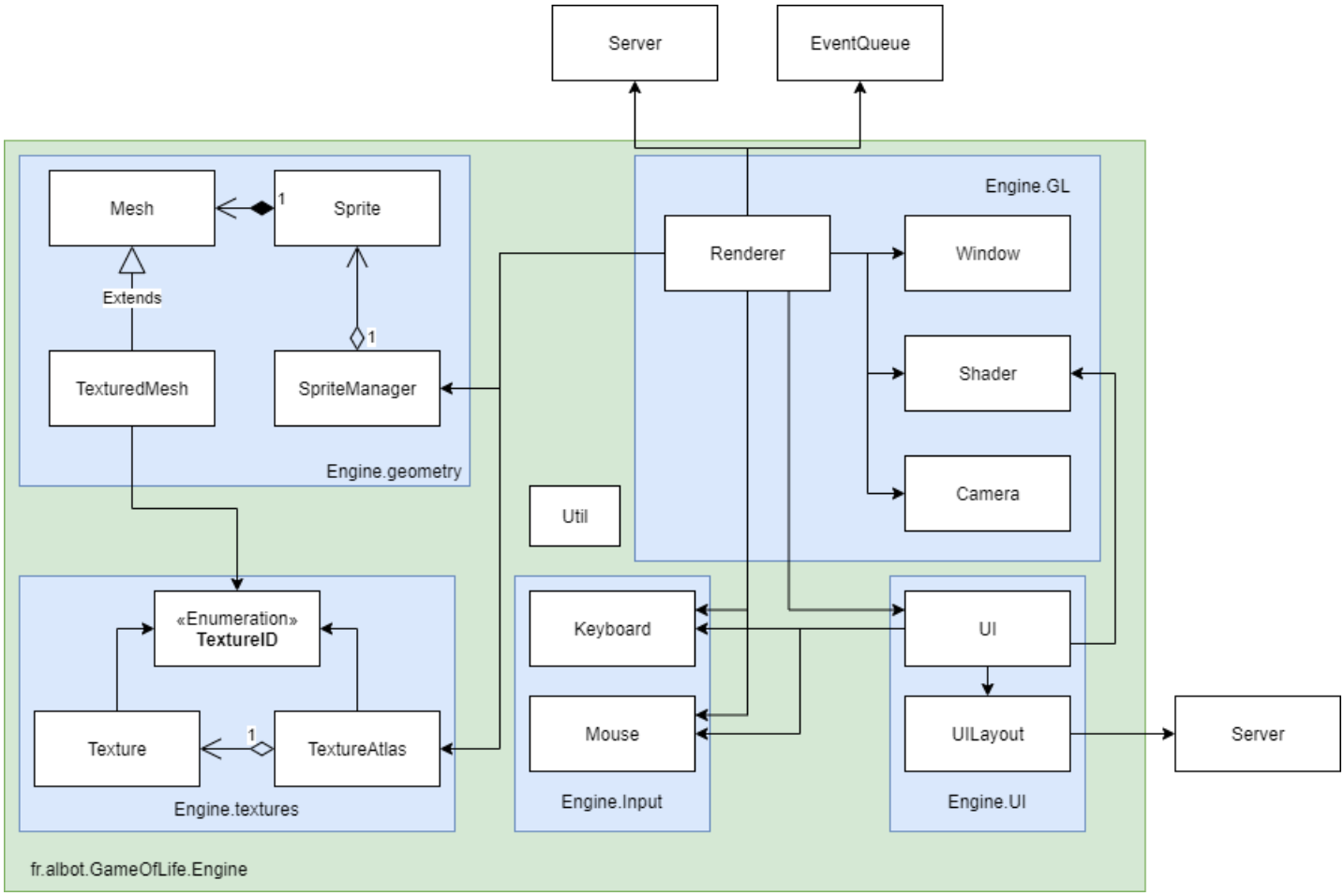


Les classes indiquées en rouges sont exécutables. Les classes indiquées en bleues sont des classes provenant de JavaRMI

Pour l'utilisation de JavaRMI, la classe GameOfLife hérite de UnicastRemoteObject car c'est elle qui exposera ses méthodes sur le réseau et il ne peut y avoir qu'une seule instance d'elle même. Celles ci sont définies par l'interface IGameOfLife, héritant de Remote. L'énumération Status permet à GameOfLife d'indiquer aux clients d'attendre pour mieux gérer les accès en concurrence. Par exemple, le temps que la nouvelle génération soit calculée.

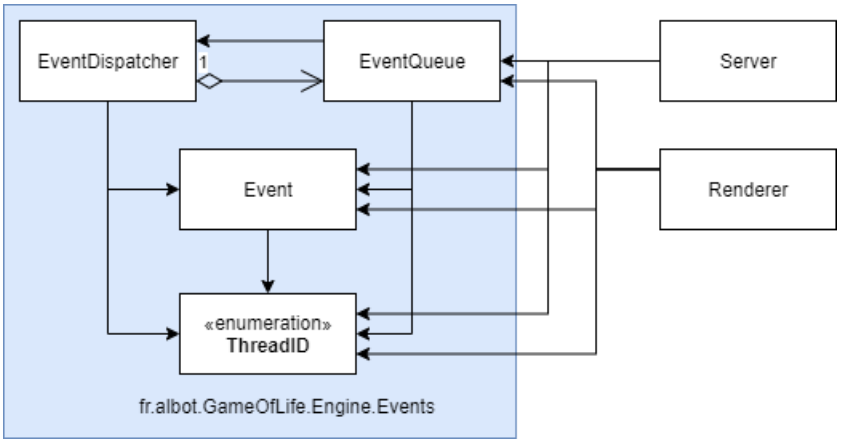
Par rapport à l'implémentation du pattern Bag Of Task, ici le Bag est la classe GameOfLife et la task est définie par l'interface IGOLProcess. Comme indiqué précedemment, le client n'a accès qu'a des interfaces pour traiter des tâches anonymes.

3.2 Organisation du package Engine



Le package contient le Renderer et ses classes auxiliaires. Util ne contient que des méthodes statiques utilisées notamment pour le chargement des ressources.

3.3 Organisation du package Events



Le package contient les classes qui gèrent les évènements ainsi que les variations d'Events. Pour créer un évènemenent, il faut créer une classe héritant de Event. On peut vérifier le type de

l'évènement reçu avec le mot clé java instanceof. Il est possible d'ajouter des valeurs, objets en arguments de l'évènement, par exemple passer la liste des cellules vivantes au rendu qui, quand le SpriteManager sera crée, le mettra a jour de manière asynchrone.

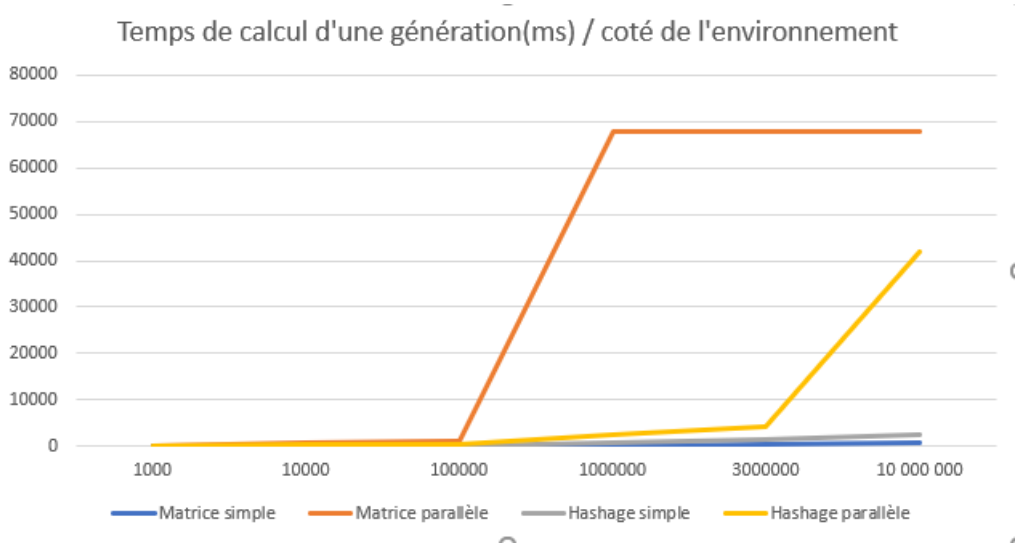
4 Performances

4.1 Protocole

Le serveur et les clients sont hébergés sur une seule et même machine. Les tests en mode distribué s'effectueront avec 4 clients. On teste la variante matricielle, mono-machine et distribuée, ainsi que la variante par hashage, mono-machine et distribuée également. Les tests seront effectués pour des tailles d'environnement de 1000, 1 000 000, 10 000 000 de cellules correspondant à des tailles d'environnement de 10, 100, 215. L'affichage sera désactivé afin de ne pas impacter les traitements. La configuration de départ sera un remplissage de moitié avec des cellules vivantes à des positions aléatoires. On examinera le temps de calcul d'une génération, initialisation et changement de génération non compris.

4.2 Mesures

Type	Coté env.	Temps de calcul	clients	Taille d'un chunk
Matrice Simple	10	0 - 1	n/a	n/a
Matrice Simple	20	0 - 1	n/a	n/a
Matrice Simple	50	9 - 10	n/a	n/a
Matrice Simple	100	75	n/a	n/a
Matrice Simple	150	250	n/a	n/a
Matrice Simple	200	600 - 650	n/a	n/a
Matrice parallèle	10	70	4	n/a
Matrice parallèle	20	560	4	n/a
Matrice parallèle	50	1000 - 12000	4	n/a
Matrice parallèle	100	68 000	4	n/a
Matrice parallèle	150	trop long	4	n/a
Matrice parallèle	200	trop long	4	n/a
Hashage simple	10	0 - 1	n/a	n/a
Hashage simple	20	2 - 3	n/a	n/a
Hashage simple	50	40 - 55	n/a	n/a
Hashage simple	100	350 - 4000	n/a	n/a
Hashage simple	150	1200 - 1300	n/a	n/a
Hashage simple	200	1500 - 2800	n/a	n/a
Hashage parallèle	10	35 - 45	4	100
Hashage parallèle	20	250 - 290	4	1000
Hashage parallèle	50	350 - 450	4	12500
Hashage parallèle	100	2000 - 3000	4	50000
Hashage parallèle	150	3500 - 4800	4	85000
Hashage parallèle	200	42000	4	1000000



Pour les variantes avec hashage, comme elles sont dépendantes du nombre de cellules, on observe un premier cycle très long, avant que les temps de calculs ne se réduisent.

4.3 Résultats

On constate que les versions parallèles sont bien plus longues que les versions simples. Cela vient de plusieurs facteurs : une mauvaise optimisation des algorithmes, une mauvaise parallélisation des calculs, un overhead trop important et un nombre de clients trop faibles.

On obtient les paramètres suivants en comparant matrice simple et hashage parallèle pour un environnement de taille 3 000 000.

- Facteur d'accélération réel : 0.05
- Facteur d'accélération maximal théorique : 4
- Efficacité : 0.01

5 Conclusion

Pour conclure, bien que les résultats ne soient pas présents, on peut tout de même constater l'impact des structures de données et algorithmes sur le temps de traitement en parallèle.

force est de constater que les problèmes de parallélisation, avec des threads ou des machines, restent parmi les plus complexes et intéressants à traiter.