Systèmes Distribués
Sujet 4 - Le jeu de la vie A

Leestmans Richard

14 Décembre 2022

Table des matières

1 Introduction

Le jeu de la vie est un automate cellulaire, c'est-à-dire une grille contenant des cellules pouvant posséder plusieurs états (ici deux états: morte ou vivante). Ce problème mathématique a été développé par John Conway en 1970 avec l'objectif initial de créer une machine capable de s'auto reproduire, en se basant sur le travaux de John Van Neumann. Les cellules évoluent dans une grille régulière à deux dimensions et sont soumises à des règles influençant la survit, la naissance ou bien la mort de ces cellules. Les règles initiales énoncées par John Conway sont les suivantes:

- *une cellule vivante possédant un nombre de cellules vivantes adjacentes compris entre 2 et 3 survit*;
- *une cellule morte possédant un nombre de cellules vivantes adjacentes égal à 3 revient à la vie.*

Ces règles ont pour but de ne pas faire croitre de façon infinie la population cellulaire, mais surtout de mettre en évidence des structures intéressantes de formes et de période variées. Ce projet a pour but de mettre en place une simulation du jeu de la vie en 3 dimensions tout en y incorporant un paradigme de système distribué afin d'accélérer les temps de calculs.

2 Présentation et analyse du modèle retenu pour le jeu de la vie en 3D

Afin d'appliquer les principes des systèmes distribués, une partie du programme va être parallélisée, c'est-à-dire qu'une portion du code va être découpée en plusieurs tâches qui seront exécutées simultanément par plusieurs processus. La portion choisie est la détermination d'une nouvelle génération de cellules. En effet, chaque processus, participant à la portion de code parallélisable, va recevoir une portion de la grille de la génération de cellules actuelle, va déterminer la nouvelle génération dans cette portion et va renvoyer le résultat afin d'obtenir intégralement la nouvelle génération. Pour permettre cela, un processus (le processus 0) aura, entre guillemets, le rôle de chef d'orchestre. C'est lui qui déterminera le nombre de feuillets à envoyer à tous les autres processus, qui découpera le cube et qui récupèrera les résultats afin de reconstruire la grille de la nouvelle génération. En d'autres termes lorsque l'utilisateur choisira d'utiliser **n** cœurs de processeurs, **1 cœur** (le processus 0) sera assigné à toute la partie séquentielle du code tandis que tous les autres processus (**processus 1 à n - 1**) seront assignés à toute la portion du code qui est parallélisable. Par conséquent, si l'utilisateur souhaite utiliser 10 cœurs pour la partie parallélisable, il devra choisir 11 cœurs à l'exécution du programme.
Il est précisé que la grille est découpée en feuillets qui sont envoyés aux autres processus. On va en fait découper le cube selon l'axe des x et les plans en deux dimensions ainsi obtenus vont être envoyés aux processus adéquats en fonction de leur rang et des positions des plans dans le cube. Il faudra également prendre en compte les feuillets voisins à ceux envoyés afin de déterminer la nouvelle génération de cellules. De plus, comme la grille est découpée en feuillets, il faut veiller à ce que le nombre de processus effectuant les tâches en parallèle n'excèdent pas le nombre de feuillets. Ces deux derniers points seront développés plus loin dans le rapport.

Comme le processus 0 s'occupe de la partie séquentielle du programme, il est donc le seul à savoir quand toutes les analyses sont terminées. On devra donc mettre en place un système permettant au processus 0 d'avertir les autres processus de la fin de l'exécution du programme afin qu'ils puissent tous s'arrêter. Ces derniers seront de leur côté à l'intérieur d'une boucle par défaut infinie, que seul le processus 0 pourra arrêter.

Afin de générer la première génération de cellules, on va utiliser le principe de seed. Cette valeur, qui sera choisie par l'utilisateur, permettra d'initialiser le générateur de nombre pseudo-aléatoire de la méthode srand. Cette façon de faire a un double avantage : pour une valeur de seed donnée, un pattern de la première génération de cellules donné et un résultat qui sera toujours le même en

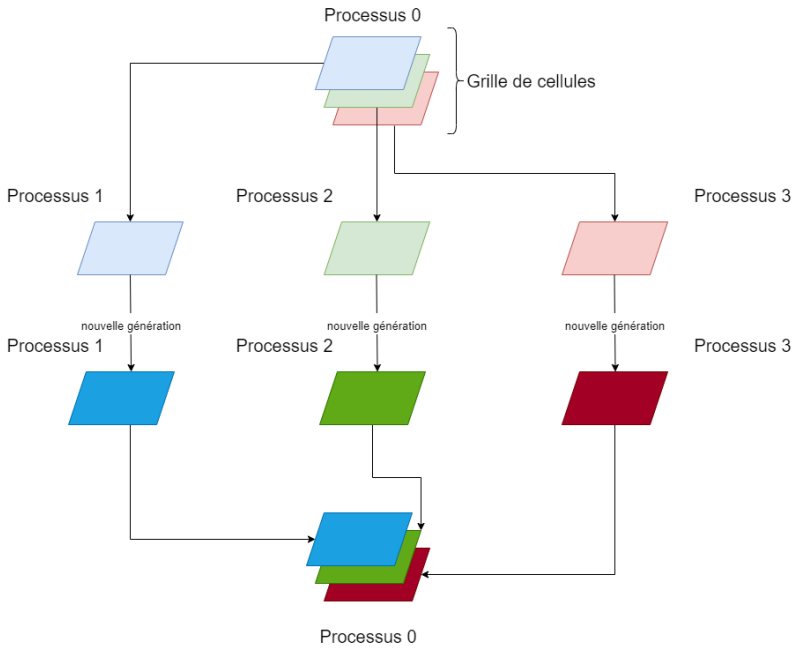


Figure 1: *Schéma du fonctionnement du découpage de la grille de cellules en feuillets avec envoi aux processus effectuant les tâches parallèles et reconstruction de la nouvelle génération de cellules à partir des résultats obtenus.*

fonction de la règle du jeu de la vie utilisée par le programme. De plus, comme il y a un pattern possible par valeur de seed, cela sera utile notamment dans la cas des analyses de performances qui seront abordées plus tard dans la rapport.

Concernant la grille en trois dimensions qui contiendra les cellules vivantes, elle sera représentée sous la forme d'un cube dans le programme. Cela a l'avantage de n'avoir à manipuler la grille qu'avec une seule valeur car les dimensions du cube sont les mêmes dans les trois dimensions de l'espace. Le contenu de la grille sera constitué de 0 (cellule morte ou pas de cellule) et 1 (cellule vivante) mais sous le forme de **char** et non des int afin d'économiser de la place en mémoire (un char est codé sur 1 octet tandis qu'un int est codé sur 4 octets). De plus, les générations de cellules seront toutes créés dans le tas afin de mieux les manipuler et d'éviter de faire des copies en mémoire.

Le but du jeu de la vie est d'arriver à trouver des configurations stables, c'est-à-dire qu'une grille de cellules de génération **n** est identique à la grille de cellules de la génération précédente **n - 1**. Mais on trouve d'autres types de configurations: ce sont les configurations oscillatoires. Ce sont les générations qui sont identiques à des générations antérieures toutes les x périodes. Par exemple une configuration est oscillatoire de période 2 si la grille de cellules de la génération n est identique à celle de la génération n - 2. Mais, pour pouvoir constater ces résultats, il va donc falloir stocker les générations antérieures en mémoire. Mais combien? Car les générations stockées prennent de la place en mémoire et, plus on en stocke, plus on sature la mémoire. Le choix a donc été fait de ne stocker en mémoire que les trois dernières générations. Ce chiffre a été choisi à partir de la fréquence d'apparition de structures intéressantes constatée dans le jeu de la vie en deux dimensions.[1]. On va supposer que les structures trouvées dans la version en deux dimensions seront similaires à celles trouvées en trois dimensions au détail près qu'elles seront dans un cube et non plus un carré. On remarque que parmi les structures avec les fréquences d'apparition les plus élevées, la plupart sont, soit de période 1, soit de période 2. On trouve une structure de période 3 (le pulsar) en position 19 des fréquences d'apparition. Et la première structure qui n'est ni de période 1, ni de période 2, ni de période 3 est le pentadécathlon en position 55 mais avec une période de 15 et une fréquence d'apparition constatée de 1 toutes les 1 631 317 grilles de cellules. D'où le fait de ne stocker que les

[1] Fréquence d'apparition de structures stables ou oscillatoires dans le jeu de la vie en deux dimensions

trois dernières générations afin d'économiser de la mémoire pour d'autres traitements.

En plus de pouvoir déterminer une configuration pour une seed et une taille de grille de cellules données, le programme va permettre également de pouvoir faire de même mais avec une plage de seeds et de taille de grilles. Cette implémentation est utile dans le cas où l'utilisateur souhaite avoir un aperçu global sur un ensemble de données. Et s'il trouve un cas intéressant, il pourra ensuite faire une recherche plus approfondie sur un cas particulier de seed et de taille de grille de cellules. Ce choix sera abordé plus tard dans le cas de la configuration du jeu de la vie par l'utilisateur.

Afin de ne pas surcharger le contenu de la console de beaucoup d'informations qui seraient illisibles à l'œil, seules des informations précises seront affichées : le résumé de la configuration du jeu de la vie par l'utilisateur, le nombre de feuillets qui sont envoyés aux différents processus, un message si une génération de cellules pour une seed et une taille de grille données n'a pas donné de résultat concluant, un autre lorsqu'un résultat intéressant est trouvé, l'affichage de ce résultat, et enfin, la liste de tous les processus une fois qu'ils ont arrêté leur exécution.
Enfin, le programme s'articulera autour de 4 structures de données. La première, appelée **Configuration**, s'occupera de tout de ce qui est la configuration du jeu de la vie par l'utilisateur. La seconde, **Data**, s'occupera de tout ce qui est parallélisable dans le programme : de la réception des données envoyées par le processus 0, à l'envoi des résultats à ce dernier, en passant par la détermination de la nouvelle génération de cellules. Tous les processus (autres que le processus 0) possèderont une instance de cette structure de données. La troisième, **Grid**, est la partie principale du programme gérée par le processus 0. C'est grâce à cette structure de données, que sera générer la première génération de cellules, que les grilles de cellules seront découpées en données et envoyées aux processus effectuant les tâches parallélisables. C'est également grâce à cette structure que le processus récupèrera les résultats et construira la grille de cellules de la nouvelle génération et pourra donc l'analyser derrière et donner un résultat sur cette analyse. Enfin, la dernière structure de données, **FileResults**, va permettre de créer un rendu visuel de l'exécution du programme afin de permettre à l'utilisateur de mieux interpréter les résultats. Ce rendu se matérialisera par un fichier au format html dans lequel sera écrit les informations utiles sur les résultats obtenus. De plus, selon la taille des données, cette structure proposera également à l'utilisateur de sauvegarder les résultats au format pdf. Pour résumer, le programme sera subdivisé en deux parties: une partie séquentielle effectuée uniquement avec le processus 0 et une partie parallèle effectuées par tous les autres processus. La partie parallèle sera gérée par la structure **Data**, tandis que la partie séquentielle sera subdivisée en 3 parties: la partie configuration du programme gérée par **Configuration**, le déroulement principal du programme géré par **Grid** et enfin la partie écriture dans un fichier html et exportation des résultats par **FileResults**.

3 Choix du paradigme de système distribué

Le paradigme de systèmes distribués retenu pour ce projet est **MPI (Message Passing Interface)** car c'est le paradigme le plus approprié pour la parallélisation des tâches. En effet, MPI permet d'effectuer du calcul haute performance mais, surtout, avec MPI, tous les processus exécutent le même programme mais avec un sous-ensemble de données. Et c'est exactement ce que fait le notre. Les processus effectuant la portion du code parallélisable ont tous la même tâche: recevoir une fraction de la génération de cellules actuelle, déterminer une nouvelle génération de cellules à partir des données reçues et renvoyer les données au processus 0 et c'est tout. De plus ces processus ont un rôle uniquement calculatoire: ils déterminent seulement la nouvelle génération de cellules. L'analyse des résultats se fera dans le processus 0. On utilisera l'implémentation de MPI **MPICH** pour notre programme. MPI étant une bibliothèque de C, l'intégralité du programme sera réalisé dans ce langage de programmation.

Les communications entre le processus 0 et les autres processus se fera par l'intermédiaire des méthodes **MPI_Send** et **MPI_Recv**. On utilise spécifiquement ces fonctions car les données

envoyées et reçues par le processus 0 seront de tailles variables et différentes selon le processus concerné. On ne peut donc pas utiliser des fonctions comme MPI_Scatter ou encore MPI_Gather. De plus, on utilise la communication synchrone entre les différents processus car c'est le processus 0 qui régit l'envoi des données. Les processus effectuant les tâches en parallèle vont recevoir les données du processus 0, vont faire leur calcul et renvoyer les résultats au processus 0. Puis ils vont de nouveau attendre que le processus 0 leur envoie des nouvelles données. Le nombre de communications inter-processus est corrélé au nombre de taille de grille de cellules à analyser, au nombre de seeds, au nombre de de générations maximales autorisées par l'utilisateur pour une seed et une taille de grille donnée, ainsi qu'au nombre de processus utilisés dans le programme. Si on note ces paramètres respectivement dans l'ordre m, n, p et q, le nombre de communications sera dans le pire des cas (c'est-à-dire qu'aucune configuration stable ou oscillatoire n'a été trouvée dans le nombre de générations maximal choisi) de $2 \times m \times n \times p \times (q-1)$. On prend q - *1* car le processus 0 communique avec tous les autres processus (lui excepté) et inversement. Mais, en plus de ces communications, comme il est précisé dans la *section 2*, les processus participant aux tâches parallèles n'ont aucune connaissance du déroulé total du programme donc quand devoir s'arrêter. Par conséquent, dès qu'il n'y a plus d'analyse à faire,le processus 0 va envoyer le mot *"arret"* aux autres processus pour leur dire d'arrêter leur exécution. Par conséquent le nombre total de communication (MPI_Send et MPI_Recv inclus) sera, dans le pire des cas:

$$\text{nombre communications inter-processus} = 2mnp(q-1) + (q-1) = (2mnp+1)(q-1)$$

4 Choix de la règle du jeu de la vie

Pour rappel, la règle du jeu de la vie originale de John Conway est : *"Si une cellule vivante possède entre 2 et 3 cellules vivantes adjacentes, elle survit. Si une cellule morte possède exactement 3 cellules vivantes adjacentes, alors elle devient vivante"*. Cette règle fonctionne bien avec un environnement à deux dimensions car chaque cellule peut posséder entre 0 et 8 cellules vivantes adjacentes. En revanche, dans un environnement en trois dimensions, on passe de 0 à 26 cellules vivantes adjacentes possibles. Par conséquent, la probabilité qu'au moins une deux règles se produise sur une cellule devient plus faible. Il faut donc trouver de nouvelles règles pour une configuration à trois dimensions. Pour le projet, nous allons utiliser les règles découvertes en 1987 par Carter Bays et publiées dans son article de recherche *Candidates for the Game of Life in Three Dimensions*[2]. Il explique dans son article que, pour qu'une règle soit considérée comme une règle du jeu de la vie, elle doit respecter les deux définitions suivantes:

1. *Un planeur doit exister et doit se produire "naturellement" si on applique cette règle, à plusieurs reprises, à des configurations de soupe primordiale* (une soupe primordiale correspond à un pattern de cellules de n'importe quelle dimension finie et remplie de cellules vivantes dispersées aléatoirement);

2. *Toutes les configurations de soupe primordiale, quand elles sont soumises à cette règle, doivent afficher une croissance limitée.*

Dans la première définition, un planeur est une structure du jeu de la vie qu'avait découvert John Conway en 1970 et qui est de période 4 (elle retrouve sa forme initiale toutes les 4 générations) tout en s'étant déplacée d'une case en diagonale pendant cette même période. Comme c'est une structure classique du jeu de la vie, Carter Bays a défini que cette structure devait être présente en utilisant n'importe quelle règle du jeu de la vie.
La seconde définition empêche que le nombre de cellules vivantes dans la génération de cellules croit de façon exponentielle.
De ses travaux, Carter Bays a déduit deux règles satisfaisant les deux définitions:

[2] Candidates for the Game of Life in Three Dimensions - Carter Bays (1987)

1. *Si une cellule vivante possède entre 5 et 7 cellules vivantes adjacentes, elle survit. Si une cellule morte possède exactement 6 cellules vivantes adjacentes, alors elle devient vivante*;

2. *Si une cellule vivante possède entre 4 et 5 cellules vivantes adjacentes, elle survit. Si une cellule morte possède exactement 5 cellules vivantes adjacentes, alors elle devient vivante.*

Dans la suite du rapport, on utilisera la première règle. De plus, on utilisera également le format de règle définit dans l'article que ce soit dans le rapport et dans le programme (l'utilisateur doit choisir sa règle pour lancer le jeu de la vie en utilisant ce format). Elle se présente sous la forme $X_1X_2Y_1Y_2$ avec X_1 et X_2 les bornes inférieure et supérieure de l'intervalle contenant l'ensemble des valeurs du nombre de cellules vivantes permettant à une cellule vivante de survivre et Y_1 et Y_2 les bornes inférieure et supérieure de l'intervalle contenant l'ensemble des valeurs du nombre de cellules vivantes adjacentes permettant à une cellule morte de devenir vivante. Ainsi, la règle du jeu de la vie en deux dimensions de John Conway s'écrit donc **2333** sous ce format et la règle, que l'on va utiliser pour la suite du projet, **5766**.

5 Présentation de l'algorithme du déroulé du programme

L'algorithme utilisé lors de l'exécution du programme est le suivant et fonctionne aussi bien pour une analyse sur une seed et une taille de grille de cellules que sur une plage de données:

DEBUT

- ***étape 1*** : lecture de la valeur suivante de la plage de taille de grille de cellules;
- ***étape 2*** : lecture de la valeur suivante de la plage de seeds;
- ***étape 3*** : détermination, par le processus 0, de la première génération de cellules en fonction de la valeur de la seed;
- ***étape 4*** : génération, par le processus 0, du nombre de feuillets à envoyer à tous les autres processus;
- ***étape 5*** : lecture de la valeur suivante du nombre de générations maximal autorisé;
- ***étape 6*** : envoi des feuillets à analyser plus leur feuillets voisins à tous les autres processus sous le forme de datagrammes;
- ***étape 7*** : les processus, autres que le processus 0, reconstruisent leur propre grille de cellules à partir des données reçues;
- ***étape 8*** : ils déterminent une nouvelle génération de cellules dans les feuillets d'intérêt (ils ne le font pas pour les feuillets voisins);
- ***étape 9*** : les processus, autres que le processus 0, repassent les feuillets d'intérêt sous forme de données (sans prendre en compte les feuillets voisins) et les renvoient au processus 0;
- ***étape 10*** : le processus 0 reçoit toutes les données et construit la grille de la nouvelle génération;
- ***étape 11*** : le processus 0 analyse le contenu de la grille: *si la grille est vide* (plus de cellules vivantes) → **passage à *l'étape 15***, sinon → **passage à *l'étape 12***;
- ***étape 12*** : *si le contenu de la grille est égal à celui d'une génération déjà stockée en mémoire* → **passage à *l'étape 15*** sinon → passage à l'étape 13;
- ***étape 13*** : stockage de la génération en mémoire par le processus 0;

- ***étape 14*** : s'il y a *encore des générations à déterminer* → **retour à *l'étape 5***, sinon → **passage à *l'étape 15***;
- ***étape 15*** : affichage dans la console du résultat obtenu et de la génération à laquelle il a été obtenu;
- ***étape 16*** : s'il y a *encore des seeds à analyser* → **retour à *l'étape 2*** sinon → **passage à *l'étape 17***;
- ***étape 17*** : s'il y a *encore des tailles de grille de cellules à analyser* → **retour à *l'étape 1*** sinon → **fin du programme**;

FIN

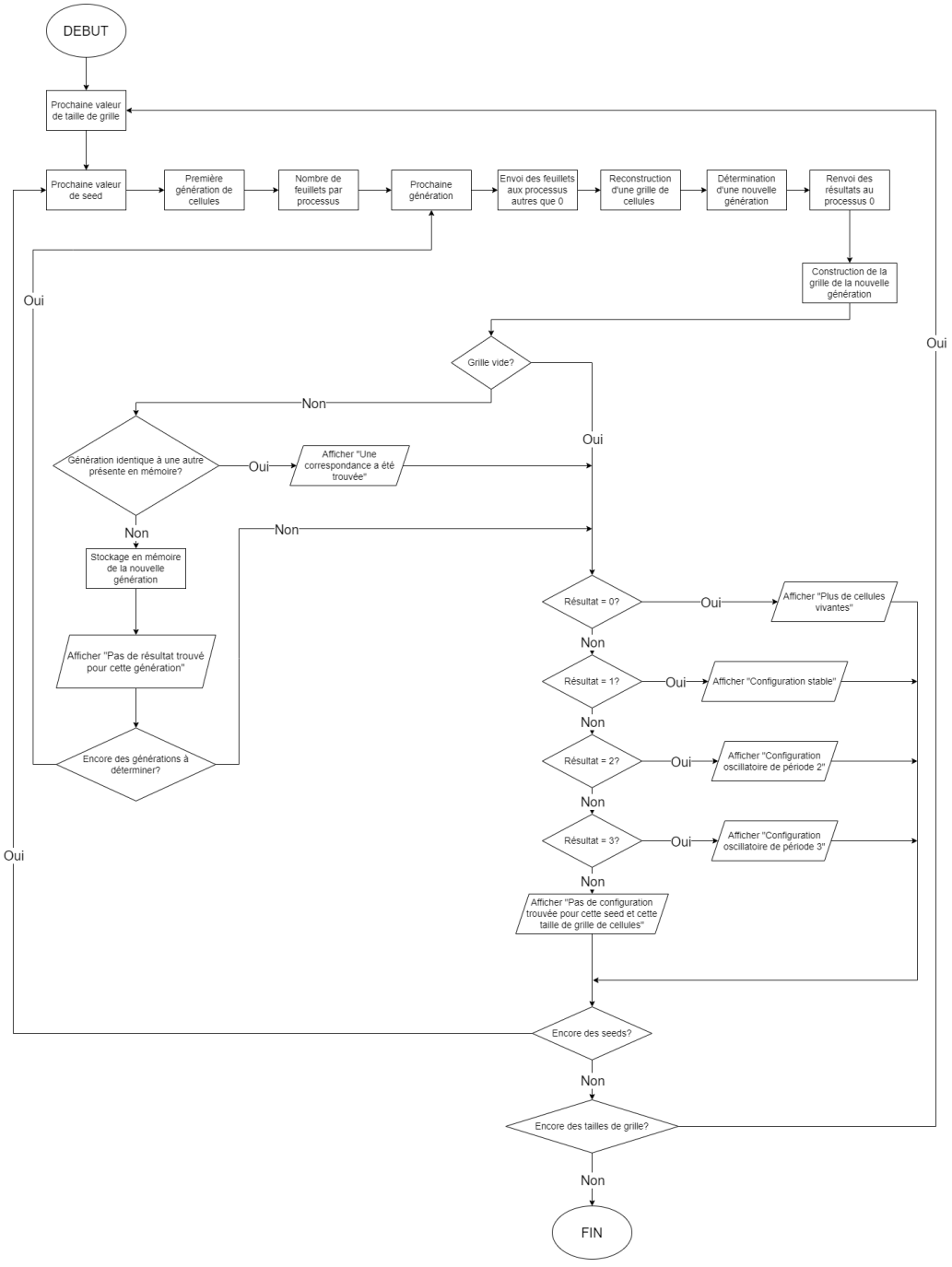


Figure 2: *Algorigramme du déroulement du programme.*

6 Documentation

Une documentation Doxygen de toutes les structures et toutes les fonctions du programme est présente dans le fichier *index.html* du dossier **doc**. Dans le fichier *index.html*, la présentation des structures et de leur attributs se fait dans la partie *structure de données* et celle des fonctions dans la partie *fichiers*.

7 Compilation, exécution et configuration du programme

7.1 Compilation

Pour pouvoir compiler le programme, un fichier **Makelile** est présent dans l'archive. L'utilisateur n'a donc qu'à rentrer la commande suivante dans la console pour compiler le programme:

```
make all
```

Deux autres commandes sont également présentes dans le Makefile:

- *make clean* : supprime les fichiers binaires ainsi que les fichiers de résultats contenus dans le dossier *results*;
- *make mrproper* : fait la même chose que *make clean* et supprime également l'exécutable.

7.2 Exécution

Pour exécuter le programme, plusieurs possibilités : on peut tout d'abord utiliser la commande classique (le nombre de processus et le fichier contenant les machines fournissant les cœurs de processeurs sont donnés à titre indicatif):

```
mpirun -np 25 -hostfile hostfile ./main
```

Cette commande va permettre à l'utilisateur de configurer lui-même l'intégralité des caractéristiques du programme. Mais il y a aussi la possibilité de lancer une configuration toute faite en ajoutant un paramètre supplémentaire. La commande:

```
mpirun -np 25 -hostfile hostfile ./main test1
```

permet de lancer une analyse sur une grille de cellules de dimension $50 \times 50 \times 50$, de seed 25, de règle 5766 et pour un nombre de générations maximal autorisé de 200. Et la commande:

```
mpirun -np 25 -hostfile hostfile ./main test2
```

permet de lancer une analyse sur plage de seeds allant de 0 à 100 et sur une plage de taille de grille de dimensions allant de 30 à 40. Ajouté à cela, un nombre de générations maximal autorisé de 200 et une règle du jeu de la vie de 5766.

7.3 Configuration du programme par l'utilisateur

On a vu dans le point **7.2** que la première commande d'exécution donnait la possibilité à l'utilisateur de configurer l'intégralité du programme du jeu de la vie. Les informations demandées par l'utilisateur vont être, dans l'ordre, les suivantes:

1. ***le type d'analyse*** : le type d'analyse utilisé pour l'exécution du programme parmi deux valeurs possibles:
   - *1* → Analyse sur une seule seed et une seule taille de grille (= longueur d'un des côtés du cube);
   - *2* → Analyse sur une plage de seeds et sur une plage de tailles de grille.
2. ***la règle utilisée pour la détermination de la génération suivante*** : bien que l'utilisateur possède une certaine liberté quant à la règle utilisée dans le programme, il est fortement conseillé d'utiliser les règles **5766** ou **4555** car ce sont les règles qui s'approchent le plus de celle du jeu de la vie en 2 dimensions. De plus, la longueur des deux intervalles, symbolisés respectivement par les 2 premiers et les 2 derniers chiffres de la règle ne peuvent pas excéder 2. Par conséquent **5766** est une règle valide mais **5866** n'en est pas une car la longueur du premier intervalle est de 3;
3. ***la borne inférieure de la plage de seeds*** : la valeur entrée par l'utilisateur doit être un nombre positif ou nul;
4. ***la borne supérieure de la plage de seeds*** : demandée uniquement dans le cas de l'analyse de type 2, cette valeur doit être supérieure ou égale à la valeur de la seed de la borne inférieure. Dans le cas de l'analyse sur une seule seed et une seule taille de grille de cellules, la valeur de la borne supérieure de la plage de seed est automatiquement complétée avec celle entrée pour la borne inférieure;
5. ***la borne inférieure de la plage de tailles de grille*** : cette valeur doit respecter toutes les conditions suivantes:
   - *elle doit être au moins égale à 3* → cela correspond au plus petit cube que le programme peut utiliser;
   - *elle doit être au moins égale au nombre de processus entrés par l'utilisateur moins 1* → comme chaque processus participant à la portion parallélisable (nombre de processus total moins le processus 0) doit recevoir au moins un feuillet, la valeur de la borne inférieure de la taille de grille de cellules ne peut donc pas y être inférieure. Par exemple, si l'utilisateur utilise **21 processus**, la valeur de la taille de la grille de cellules doit donc être **au minimum égale à 20**;
   - *elle ne doit pas permettre à un processus de recevoir plus de 3 375 000 cellules* → cette valeur correspond à la taille maximale du buffer utilisé pour recevoir les données issues du processus 0 (cela correspond à un cube de dimension $150 \times 150 \times 150$);
6. ***la borne inférieure de la plage de tailles de grille*** : cette valeur doit être au moins supérieure ou égale à la valeur de la borne inférieure de la taille de la grille. Comme pour la seed, cette valeur ne sera demandée à l'utilisateur que dans le cas d'une analyse de type 2 (sur une plage de données). Dans le cas d'une analyse simple, le programme aura également stocké la valeur de la borne inférieure dans celle de la borne supérieure;
7. ***le nombre de générations maximal autorisé*** : cette valeur doit être supérieure ou égale à 1.

8 Présentation et spécification des structures et des fonctions principales du programme

8.1 Structure Configuration

La structure Configuration, qui est gérée par le processus 0, est un objet permettant à l'utilisateur de configurer le jeu de la vie. En d'autres termes, cette structure va stocker toutes les informations qui vont être, soit entrées par l'utilisateur, soit définies automatiquement (dans le cas où l'utilisateur souhaite réaliser une analyse sur une grille de cellules prédéfinie) et, permettre de mener à bien l'exécution du programme. Les caractéristiques du jeu de la vie, stockées dans la structure, sont les suivants:

- ***int* typeOfAnalysis** : type de l'analyse choisi par l'utilisateur parmi deux valeurs:
  - *1* → Analyse sur une seule seed et une seule taille de grille (= longueur d'un des côtés du cube);
  - *2* → Analyse sur une plage de seeds et sur une plage de tailles de grille.
- ***char* rule[5]** : règle utilisée dans le jeu de la vie;
- ***int* seedInfRange** : Borne inférieure de la plage de seeds sur laquelle effectuer le jeu de la vie;
- ***int* seedSupRange** : Borne supérieure de la plage de seeds sur laquelle effectuer le jeu de la vie;
- ***int* gridSizeInfRange** : Borne inférieure de la plage de tailles de grille sur laquelle effectuer le jeu de la vie;
- ***int* gridSizeSupRange** : Borne supérieure de la plage de tailles de grille sur laquelle effectuer le jeu de la vie;
- ***int* nbGenerations** : Nombre de générations maximales sur lesquelles effectuer l'analyse d'une grille de cellules pour une taille et une seed données;
- ***int** tasksPerProc** : Tableau du nombre de feuillets (c'est-à-dire les niveaux ou plans du cube selon l'axe x) assignés à chaque processus, autre que le processus 0, en MPI.

Comme il a été précisé qu'on pouvait une analyse sur une plage de données de seeds et de tailles de cellules, on trouve la présence de bornes inférieure et supérieure pour ces paramètres. Ainsi, lorsque l'utilisateur choisira l'analyse d'une plages de données (analyse 2), le programme lui demandera la valeur des deux bornes. Dans le cas d'une analyse simple d'une seed pour une taille de grille, seule une valeur sera demandée pour chaque paramètre et le programme complètera automatiquement la valeur des bornes inférieure et supérieure avec la même valeur.

Toute la configuration du programme va se faire à l'aide de la fonction ***Configuration__setParameters*** qui va prendre en paramètres l'adresse d'une instance de la structure de type *Configuration* dans laquelle stocker les caractéristiques du jeu de la vie rentrées par l'utilisateur, ainsi que le nombre total de processus utilisés par le programme. Ce second paramètre va être utilisé lorsqu'on va demander à l'utilisateur la valeur de la borne inférieure de la taille de la grille de cellules qu'il souhaite. Comme précisé dans l'algorithme *(voir figure 2)*, le cube est découpé en feuillets selon l'axe x et seulement selon cet axe. Par conséquent, si le nombre de processus dépassent la longueur du cube selon l'axe x, le programme signifiera à l'utilisateur que le nombre de processus dépassent le nombre de feuillets et il s'arrêtera prématurément. Il faut également prendre en compte qu'un des processus ne participe pas à la portion parallélisable du programme.

Par exemple, si on prend un cube de dimensions $10 \times 10 \times 10$, par conséquent 10 feuillets, il faudra **au maximum** un nombre de processus égal au nombre de feuillets plus le processus 0 qui enverra les tâches aux autres processus, c'est-à-dire **11** processus maximums. Autrement dit, pour un cube de dimension $\boldsymbol{n}$, le nombre maximum de processus autorisés sera $\boldsymbol{n+1}$. Enfin, on teste aussi le rapport entre le nombre de cellules totales dans la grille et le nombre de processus disponible. En effet, comme le buffer de réception des processus effectuant a une taille de 3 375 000 octets (+15 pour les informations du datagramme), il y a donc une taille de données par cœur à ne pas dépasser (ici il s'agit de l'équivalent d'un cube de dimension 150 par 150 par 150).

```
//Choix de la borne inférieure de la taille de la grille de cellules
int validChoice = 0;
char gridSizeInfChoice[20];
do{
    strcpy(gridSizeInfChoice, "");
    printf("**********************************\n");
    printf("#### Selectionnez la borne inferieure de la plage de la taille de la grille sur laquelle vous souhaitez faire votre analyse?\n Elle doit etre au moins egale a 3 et superieure ou egale au nombre de processus choisi moins le processus 0 (%d)...\nLa taille correspond a la longueur d'un des cotes du cube (ex: taille 3 pour un cube de 27 cases): ####\n", nbProc-1);
    scanf("%s", gridSizeInfChoice);
    //On vérifie si le choix de l'utilisateur est un nombre
    int isNumber = Configuration__isANumber(gridSizeInfChoice);

    if(!isNumber)
    {
        printf("---- La valeur que vous avez entre pour la taille de la grille n'est pas un nombre. ----\n");
        validChoice = 0;
    }
    else
    {
        //Si c'est un nombre, on regarde s'il vérifie bien les conditions du programme
        int size = atoi(gridSizeInfChoice);
        if(size < 3 || size < nbProc-1)
        {
            printf("---- La valeur que vous avez entree pour la taille de la grille n'est pas conforme. ----\n");
            printf("---- Elle doit etre au moins egale a 3 et superieure ou egale au nombre de processus choisi moins le processus 0 (ici %d). ----\n", nbProc-1);
            //On détruit la structure de configuration
            Configuration__destroy(&self);
            //Et on quitte le programme
            exit(1);
        }
        else
        {
            //On vérifie si le nombre de coeurs est suffisant pour la taille du cube choisie (le programme permet à chaque coeur de pouvoir recevoir au maximum l'équivalent d'un cube de dimension 150 x 150 x 150 soit 3 375 000 cellules)
            //On récupère la taille théorique du cube choisi par l'utilisateur
            int theoreticSize = size*size*size;
            //On détermine le nombre de données par processus
            int dataPerProc = theoreticSize / (nbProc - 1);
            //si la valeur dépasse 3 375 000, on prévient l'utilisateur qu'il ne peut pas continuer
            if(dataPerProc > 3375000)
            {
                printf("---- Le nombre de coeurs choisi est trop faible par rapport à la taille de la grille de cellules choisie. Veuillez choisir une taille plus petite ou relancez le programme...\n");
                validChoice = 0;
            }
```

```
            //Sinon le résultat est valide
            else
                validChoice = 1;
        }
    }
}while(!validChoice);
//et on stocke la valeur choisie dans l'attribut idoine de la structure
self->gridSizeInfRange = atoi(gridSizeInfChoice);
```

Code 1: Conditions à respecter pour l'utilisateur dans le choix de la taille de la grille de cellules.

A noter que c'est dans cette partie du programme que l'on va également déterminer le nombre de feuillets à assigner à chaque processus autre que le processus 0. En effet, connaissant maintenant la taille de la grille de cellules choisie par l'utilisateur, ainsi que le nombre total de processus utilisés. Cela va se faire grâce à la fonction ***Configuration__setTasksPerProc*** qui prend en paramètres l'instance de la structure de *Configuration* contenant le tableau des feuillets à envoyer, la taille de la grille, ainsi que le nombre total de processus utilisés par le programme *(voir Code 2)*. On utilise un paramètre annexe pour la taille de la grille, et non ceux présents dans l'instance de *Configuration* car, dans le cas d'une analyse sur une plage de tailles de grille, la valeur de cette dernière va varier entre les bornes de l'intervalle, d'où l'utilisation d'un paramètre externe.

```
void Configuration__setTasksPerProc(Configuration* self, int gridSize, int nbProc)
{
    //Si la mémoire pour le tableau tasksPerProc n'a pas été allouée, alors on fait
    une allocation dans le tas en fonction du nombre de processuss
    if(self->tasksPerProc == NULL)
        self->tasksPerProc = (int*)malloc((size_t)(nbProc-1)*sizeof(int));

    //On met les valeurs du nombre de feuillets pour chaque processus à 0
    Configuration__resetNumOfTasks(self, nbProc);

    //On détermine le nombre de feuillets qu'auront à traiter chaque processus
    for(int i = 0; i < gridSize; i++)
    {
        self->tasksPerProc[i%(nbProc-1)]++;
    }

    //auxquels on ajoute les feuillets voisins (le processus 1 aura le sommet de la
    grille donc 1 feuillet voisin,
    //le dernier processus aura le fond de la grille donc 1 feuillet voisin
    //et les autres auront le feuillet du dessus et celui du dessous donc 2 feuillets
     voisins)
    //UNIQUEMENT S'IL Y A AU MOINS 2 PROCESSUS QUI TRAVAILLENT EN PARALLELE
    if((nbProc - 1) > 1)
    {
        for(int i = 0; i < nbProc - 1; i++)
        {
            //S'il s'agit du premier ou du dernier processus
            if(i == 0 || i == (nbProc - 2))
                self->tasksPerProc[i]++;
            //Sinon pour tout autre processus autre que le premier et le dernier
    processus
            else
                self->tasksPerProc[i]+=2;
        }
    }
}
```

Code 2: Détails de la fonction Configuration__setTasksPerProc.

On commence par créer le tableau d'une taille correspondant au nombre de processus total moins le processus 0. Puis, on balaye le tableau en incrémentant de façon séquentielle le nombre de feuillets à

envoyer à un indice donné et, ceci, tant qu'il reste des feuillets à assigner. Cela permet une répartition la plus équitable possible des feuillets à envoyer. Puis, on va y ajouter les plans voisins. En effet pour déterminer la génération de cellules suivantes, les processus gérant cette partie vont avoir besoin de toutes les cellules adjacentes à celles présentes dans les feuillets à analyser. Le cube étant découpé du haut vers le bas, le premier processus effectuant les tâches parallélisables (processus 1) recevra le sommet du cube tandis que le dernier processus (processus n-1 avec n le nombre total de processus) recevra le bas. On peut donc déduire que les processus 1 et n - 1 aura chacun un feuillet voisin, respectivement celui au niveau inférieur et celui au niveau supérieur. Tous les autres processus recevront eux le feuillet voisin supérieur et le feuillet voisin inférieur. Mais, on prend également prendre en compte le cas où un seul processus est utilisé pour la portion parallélisable du code. Dans ce cas, et uniquement dans ce cas, on envoie la totalité de la grille à ce processus.

8.2 Structure Grid

La structure Grid constitue la partie du programme qui va permettre de gérer les différentes générations de cellules côté partie séquentielle (donc utilisée par le processus 0 qui en possèdera une instance). Grâce à cette structure, on va pouvoir notamment générer la première génération de cellules grâce à la valeur de seed choisie par l'utilisateur, découper cette même grille de cellules en petits fragments pour être envoyés aux processus effectuant les tâches en parallèle. Le processus 0 pourra aussi, grâce à cette structure, récupérer les résultats de ces processus, former la grille de cellules de la nouvelle génération, l'analyser par rapport aux autres structures stockées en mémoire et enfin renvoyer le résultat final de l'analyse qui sera affiché dans la console de l'utilisateur. On trouve, à l'intérieur de la structure Grid, les attributs suivants:

- ***char****** **NGeneration** : attribut qui va garder en mémoire la dernière génération de cellules. C'est d'ailleurs dans cet attribut que sera stockée la toute première génération de cellule;
- ***char****** **NMinusOneGeneration** : gardera en mémoire l'avant dernière génération de cellules;
- ***char****** **NMinusTwoGeneration** : attribut qui gardera en mémoire l'antépénultième génération de cellules;
- ***int*** **gridSize** : contient la taille de la grille, c'est-à-dire la longueur d'un de ses côtés. Étant un cube, on n'a besoin de garder qu'une seule valeur pour avoir les dimensions totales du cube;
- ***int*** **typeOfAnalysis** : type de l'analyse choisie par l'utilisateur (contient soit 1, soit 2);
- ***char*** **rule[5]** : règle utilisée dans le jeu de la vie présente sous le format de 4 chiffres;
- ***int*** **nbGenerations** : nombre de générations maximales choisi par l'utilisateur;
- ***char***** **receivedData** : tableau de chaine de caractères dans lesquelles seront stockés les résultats envoyés par les processus participant aux tâches parallèles;
- ***int*** **nbProc** : nombre total de processus utilisés dans le programme;
- ***int**** **tasksPerProc** : tableau du nombre de feuillets à envoyer aux processus effectuant les tâches en parallèle.

Comme précisé précédemment, la structure va tout d'abord créer la première génération de cellules pour une seed et une taille de grille grâce à la fonction ***Grid__fillGrid***. On va tout d'abord allouer dans le tas un tableau de tableaux de chaines de caractères, ou char***, de la taille adéquate pour y faire tenir toute la génération de cellules. Puis on va le remplir "aléatoirement" avec des caractères '0' ou '1' en fonction de la seed utilisée pour la génération de nombres pseudo-aléatoires. Ainsi, pour une valeur de seed donnée, le pattern de cellules sera toujours le même. Cette génération de cellules sera initialement stockée dans l'attribut *NGeneration*.

Maintenant que l'on a notre première génération de cellules, il va falloir l'envoyer aux processus effectuant les tâches en parallèles via MPI en la découpant en de multiples fragments. Cela se fait dans la fonction ***createDataPerProc*** (*voir Code 3*) qui aura besoin, pour mener à bien sa tâche, de la nouvelle grille de cellules, de ses dimensions ainsi que du nombre de feuillets à envoyer à chaque processus grâce à l'attribut *tasksPerProc* qui contient l'adresse en mémoire du l'attribut du même nom présent dans la structure *Configuration*.

```
char** Grid__createDataPerProc(Grid* self)
{
    //On récupère la taille de la grille au format numérique
    int gridSize = Grid__getGridSize(self);
    //ainsi que le nombre de processus utilisés
    int nbProc = self->nbProc;
    //ainsi que sous la forme d'une chaine de caractères combien de caractères seront réservés pour la taille de la grille
    char buffer[10];
    sprintf(buffer, "_%d", gridSize);
    //On crée le tableau qui contiendra les portions de la grille destinées aux différents processus
    char** data = NULL;
    data = (char**)malloc((size_t)(nbProc-1)*sizeof(char*));
    //On crée un compteur sur le nombre de feuillets stockés
    int count = 0;
    for(int i = 0; i < nbProc - 1; i++)
    {
        size_t bufferSize = (size_t)(self->tasksPerProc[i]*gridSize*gridSize + 7) + strlen(buffer);
        //On récupère la taille des données sous la forme d'une chaine de caractères
        char dataSize[10];
        sprintf(dataSize, "_%d_", self->tasksPerProc[i]*gridSize*gridSize);
        //Pour la composition du "datagramme" qui sera envoyé à chaque processus, le format sera le suivant
        //modèle (4 char) + underscore (1 char) + taille de la grille (au moins 1 char) + underscore (1 char) + taille des données stockées (au moins 1 char) + underscore (1 char) + données
        //Exemple de datagramme: 5766_3_18_011000100110011101
        data[i] = (char*)malloc((bufferSize+strlen(dataSize))*sizeof(char));
        //On stocke les informations du modèle, de la taille de la grille, et de la taille des données
        strcpy(data[i], self->rule);
        strcat(data[i], buffer);
        strcat(data[i], dataSize);

        //Et on stocke les données
        for(int j = 0; j < self->tasksPerProc[i]; j++)
        {
            for(int k = 0; k < gridSize; k++)
            {
                strcat(data[i], self->NGeneration[j+count][k]);
            }
        }
        //On actualise la valeur de count du nombre de feuillet pour ce processus - 1 afin d'avoir les voisins des prochains feuillets stockés
        count = count + self->tasksPerProc[i] - 2;
    }

    return data;
}
```

Code 3: Détails de la fonction Grid__createDataPerProc.

Les feuillets sont stockés dans l'ordre croissant des rangs des processus. Ainsi le premier processus effectuant les tâches en parallèle héritera des premiers feuillets de la grille et le dernier processus des

derniers feuillets. Mais il va falloir ajouter des informations supplémentaires dans le message à envoyer. En effet, on a précisé dans le point *8.1*, que la configuration du jeu de la vie était gérée par le processus 0. Par conséquent, tous les autres processus n'ont aucune idée des choix de l'utilisateur. On va donc ajouter en plus des données la règle du jeu de la vie utilisée, la dimension de la grille et la longueur des données, chacun de ces paramètres étant séparé des autres par un underscore. Le format du datagramme envoyé sera donc le suivant:

règle_dimension de la grille_longueur des données_données

Les datagrammes seront stockés dans un tableau de chaines de caractères (char**) ce qui permettra un envoi plus pratique avec la fonction ***MPI_Send*** appelée consécutivement à l'intérieur d'une boucle for.

De la même façon que les datagrammes ont été envoyés, le processus 0 va recevoir les résultats des différents processus et les stocker dans un tableau de chaines de caractères (l'attribut *receivedData*). Puis, on va créer la grille de la nouvelle génération grâce à la fonction ***Grid__dataToGrid***. Afin de créer la grille (char***), les données contenues dans un (char**) seront toutes stockées à la suite dans une chaine de caractères, puis la grille sera construite à partir de cette dernière pour plus de facilité. On va ensuite tester si cette grille est vide (si elle ne contient que le caractère '0') et, si ce n'est pas le cas, la comparer avec les grilles déjà stocker en mémoire. Dans les deux cas, les deux fonctions (respectivement ***Grid__isEmpty*** et ***Grid__compareGen***) gérant ces étapes font de l'analyse case par case et dès qu'une case diffère, arrête l'analyse. De plus la deuxième fonction va retourner une valeur selon le résultat de la comparaison parmi 4 valeurs différentes:

- 0 → la nouvelle génération ne correspond à aucune génération stockée en mémoire;
- 1 → la nouvelle génération est identique à celle référencée par *NGeneration*;
- 2 → la nouvelle génération est identique à celle référencée par *NMinusOneGeneration*;
- 3 → la nouvelle génération est identique à celle référencée par *NMinusTwoGeneration*;

Et, selon ce résultat,, on va choisir de relancer ou non la recherche d'une nouvelle génération de cellules. Si la grille est vide, ou si la fonction ***Grid__compareGen*** retourne les résultats 1, 2, 3 ou 0 (dans le cas ou a atteint le nombre maximal de générations autorisé), on arrête l'analyse. Sinon on recommence une nouvelle recherche. Pour cela, on met à jour les générations stockées en mémoire: la nouvelle génération est référencée par *NGeneration* et on décale les générations stockées d'un rang (on change l'adresse de la grille contenue dans les attributs gérant cela). Si l'attribut *NMinusTowGeneration* référence déjà une grille en mémoire, alors on libère la mémoire allouée dans le tas pour cette grille avant de modifier l'adresse contenu dans cet attribut.

Une fois que l'analyse est terminée, la fonction ***Grid__analyzePattern***, qui est la super fonction englobant toutes les fonctions présentées précédemment, va retourner un résultat sous le forme d'un tableau contenant deux entiers. Le premier correspond au résultat final de l'analyse. Il peut prendre les valeurs suivantes:

- 0 → pas configuration stable trouvée car plus de cellules vivantes;
- 1 → configuration stable trouvée;
- 2 → configuration oscillatoire de période 2 trouvée;
- 3 → configuration oscillatoire de période 3 trouvée;
- 4 → pas de configuration trouvée pour le nombre de générations choisi.

Le second entier correspond à la génération à laquelle le résultat a été trouvé. A chaque fois que le processus 0 ne trouve pas de résultat concluant, il va incrémenter un compteur *genCount* qui correspond au nombre de générations déjà déterminées. Par conséquent, dès lors que l'on trouve un résultat intéressant, la valeur du compteur correspond à la génération à la laquelle le résultat obtenu mais pas à celle où la configuration spécifique s'est produite. En effet, si on prend par exemple un résultat trouvé sur une configuration oscillatoire de période 2, cela veut dire que la génération actuelle est identique à celle déterminée deux générations en arrière donc, que la configuration oscillatoire a commencé il y a deux générations. On doit donc retirer 2 à la valeur du compteur pour avoir la génération exacte à laquelle la configuration spécifique s'est produite.

```
1  //On crée et on retourne notre tableau de retour avec le résultat et la génération
2  int* tabResults = (int*)malloc(2*sizeof(int));
3
4  //Cas particulier où il y a encore des cellules vivantes mais aucune configuration
       stable n'a été trouvée dans le nombre de générations imparti
5  if(result == 0 && genCount == nbGenerations)
6      tabResults[0] = 4;
7  else
8      tabResults[0] = result;
9  //Si on a trouvé une configuration oscillatoire de période 3
10 if(result == 3)
11     genCount = genCount - 3;
12 //Sinon si on a trouvé une configuration oscillatoire de période 2
13 else if(result == 2)
14     genCount = genCount - 2;
15 //Sinon si on a trouvé une configuration stable ou bienpas de configuration pour le
       nombre de générations voulu
16 else if(result == 1 || result == 4)
17     genCount--;
18
19 tabResults[1] = genCount;
```

Code 4: Détail du calcul de la génération à laquelle une configuration spécifique a été trouvée.

8.3 Structure Data

La structure Data représente la partie du programme qui est utilisée par tous les processus effectuant des tâches en parallèle (tous les processus autres que le processus 0 vont en avoir une instance) et qui permet à ces derniers de de décrypter les données reçues par le processus 0, de déterminer la nouvelle génération de cellules et de la lui envoyer. Cette structure est composée des attributs suivants:

- ***char**** grid** : grille de cellules qui contiendra la portion du cube à analyser par le processus;
- ***int* numProc** : nombre total de processus utilisé dans l'exécution du programme;
- ***int* nbPlanes** : Nombre de plans ou feuillets envoyés par le processus 0 à ce processus;
- ***char* rule[5]** : règle utilisée pour l'analyse des cellules et la détermination d'une nouvelle génération;
- ***int* gridSize** : taille de la grille, c'est-à-dire ici la longueur d'un des côtés du cube.

Comme il s'agit de la structure des processus effectuant les tâches parallèles, on va tout d'abord extraire les informations contenues dans le datagramme envoyé par le processus 0 et reçu via une fonction **MPI_Recv**. L'extraction des données se fera grâce à la fonction ***Data__storeData***. En récupérant la position de tous les underscores dans le datagramme, on va pouvoir remplir les 4 derniers attributs présentés précédemment. La détermination du nombre de feuillets dans les données va se faire grâce à un petit calcul : comme un feuillet est un plan en deux dimensions de même dimension (la taille de la grille), le nombre de feuillets correspond donc au rapport entre la

longueur total des données sur la taille de la grille au carré. Les données vont être transformées en grille en trois dimensions avec la fonction ***Data__dataToGrid*** grâce aux informations de nombre de feuillets et taille de grille récupérées précédemment. Cette grille sera stockée dans l'attribut *grid* de la structure. Puis on va déterminer la nouvelle génération de cellule, dans la fonction ***Data__*** mais pas sur toute la grille. En effet, parmi les données envoyées par le processus 0, en plus des feuillets à analyser, il y a également leurs feuillets voisins. Mais on n'a pas besoin de les analyser (ils ont juste pour but d'avoir l'intégralité des cellules adjacentes aux cellules des feuillets d'intérêt). Et un processus peut savoir quels sont le ou les feuillets voisins qu'il possède. En effet, comme dit dans la partie *8.2*, les feuillets ont été envoyés par ordre croissant de rang de processus. Par conséquent si un processus a reçu **n** feuillets, le feuillet voisin du premier processus participant aux tâches parallélisables sera le feuillet **n - 1**, celui du dernier processus le feuillet **0** et pour tous les autres les feuillets **0** et **n - 1**. L'algorithme de la détermination de la nouvelle génération de cellules est très simple: on parcourt chaque feuillet, que l'on souhaite analyser, de la grille à l'aide de trois boucles for imbriquées puis, en fonction de la position de la cellule, on va tester de 7 (pour les cellules dans les coins) à 26 cellules adjacentes (pour les cellules à l'intérieure de la grille).

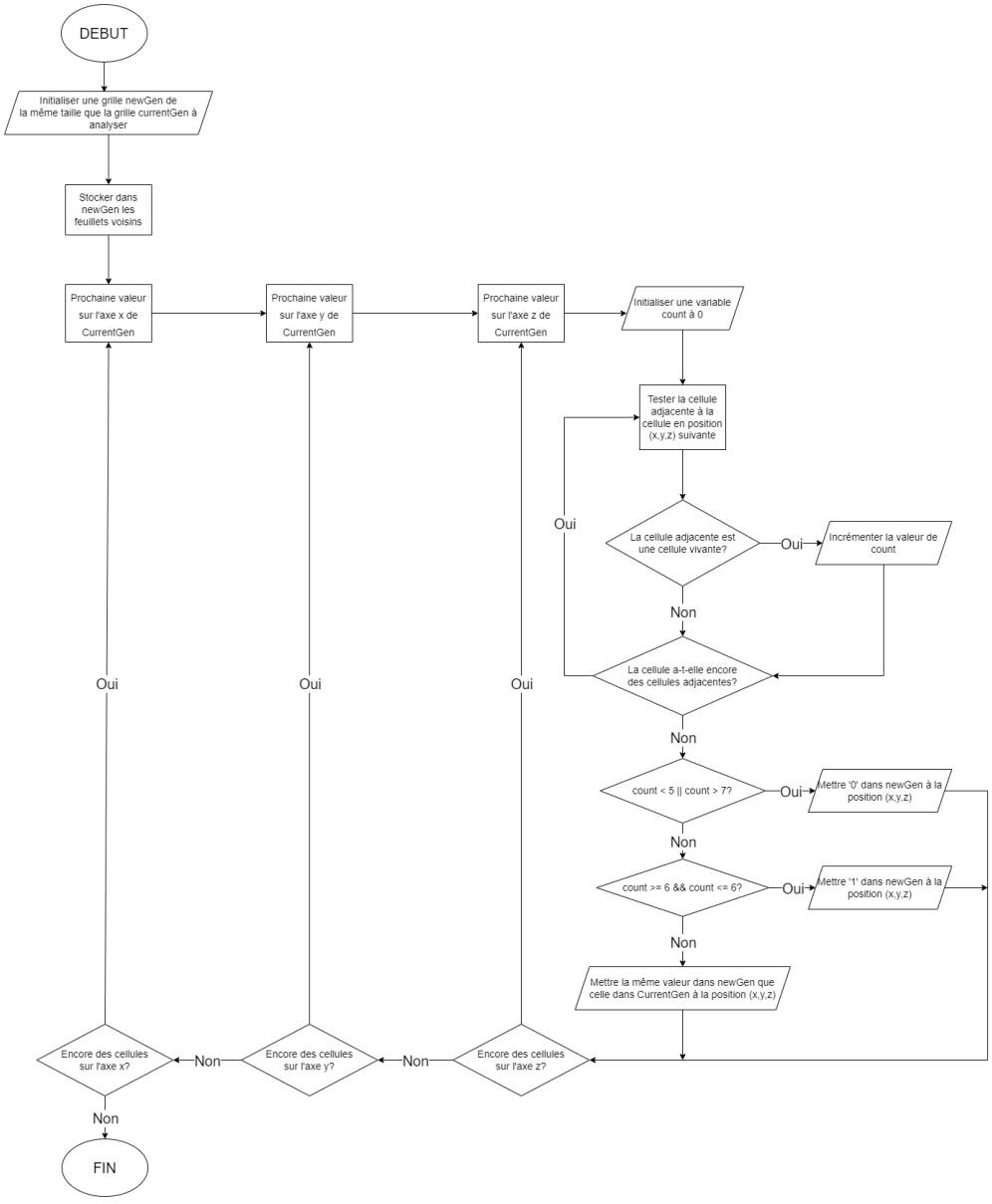


Figure 3: *Algorigramme de la détermination d'une nouvelle génération de cellules en utilisation la règle 5766.*

Pour chaque cellule, on va compter le nombre de cellules vivantes adjacentes et, en fonction du

nombre et des règles du jeu de la vie utilisées, savoir si la cellule survit, meurt ou revient à la vie.

```
//sur l'arête avant
/*      +---------+
       /         /|
      /         / |
     +---------+  |
     |         |  |
     |         |  +
     |         | /
     |         |/
     +xxxxxxxx+ */
if(k == 0)
{
    //5 cases sur le même niveau que la case en question
    if(currentGen[i][j - 1][k] == '1')
        count++;
    if(currentGen[i][j - 1][k + 1] == '1')
        count++;
    if(currentGen[i][j][k + 1] == '1')
        count++;
    if(currentGen[i][j + 1][k + 1] == '1')
        count++;
    if(currentGen[i][j + 1][k] == '1')
        count++;
    //6 cases sur le niveau supérieur
    if(currentGen[i - 1][j - 1][k] == '1')
        count++;
    if(currentGen[i - 1][j - 1][k + 1] == '1')
        count++;
    if(currentGen[i - 1][j][k + 1] == '1')
        count++;
    if(currentGen[i - 1][j + 1][k + 1] == '1')
        count++;
    if(currentGen[i - 1][j + 1][k] == '1')
        count++;
    if(currentGen[i - 1][j][k] == '1')
        count++;
}
```

Code 5: Exemple des tests effectués pour une cellule se trouvant sur l'arrête inférieure de la face inférieure de la grille.

Attardons-nous maintenant sur la complexité de cet algorithme. Comme on utilise trois boucles for, on pourrait penser à une complexité cubique (en $O(n^3)$). Seulement la dimension en x de la grille correspond aux nombres de feuillets à analyser, et ce nombre est variable. Il peut être égal à la totalité de la grille dans le pire des cas (exécution séquentielle donc complexité en $O(n^3)$ à un seul feuillet à analyser par processus, auquel cas on passerait à une complexité de $O(n^2)$. En tenant compte de cela, on peut supposer que le passage d'une exécution totalement séquentielle à une exécution parallèle pourrait avoir un impact non négligeable sur le durée d'exécution du programme. Enfin, le processus va envoyer le résultat au processus 0 en transformant la grille en chaine de caractères grâce à la fonction ***Data__gridToData*** qui a l'effet l'inverse de ***Data__dataToGrid***. Mais cette fois-ci, on va en plus retirer les feuillets voisins des données à envoyer en utilisant le rang du processus. Cela permet au processus 0 de ne récupérer que des fragments de la nouvelle génération de cellules et donc de construire la grille de cette nouvelle génération plus facilement.

8.4 Structure FileResults

La structure FileResults est la partie du projet permettant de centraliser les résultats et de générer un affichage dans une page html voire au format pdf (au choix de l'utilisateur et selon la taille des données) afin de permettre une meilleure interprétation des résultats. Cette partie du programme est gérée par le processus 0. La structure possède les attributs suivants:

- ***char* path[50]** : chemin du fichier de résultats;
- ***FILE** file** : adresse du fichier, permettant de l'ouvrir, d'y stocker des données et de le fermer.

Le seul but de cette structure est d'écrire des données dans un fichier de résultats exportable également en pdf. Le format du fichier dépendra du type d'analyse. Pour une analyse sur une seule seed et une taille de grille, le programme va d'abord écrire les informations de configuration de l'utilisateur, puis la première génération de cellules afin d'avoir une idée de à quoi elle ressemblait initialement. Enfin, il écrira des patterns de générations de cellules en fonction du résultat du programme:

- 0 → le programme écrira un message disant qu'il n'a pas trouvé de configuration car il n'y avait plus de cellules vivantes;
- 1 → le programme écrira la configuration stable ainsi que la génération à laquelle il l'a trouvé;
- 2 → le programme écrira les deux états de la configuration oscillatoire de période 2;
- 3 → le programme écrira les trois états de la configuration oscillatoire de période 3;
- 4 → comme pas de configuration intéressante n'a été trouvée dans le nombre de générations maximal choisi, le programme va écrire le contenu de la dernière génération de cellules obtenu.

Quand le programme affiche des patterns finaux, il n'affiche que les feuillets de ces patterns contenant au moins une cellule vivante afin de ne pas avoir de fichier inutilement long.
Dans le cas d'une analyse sur une plage de données, le programme va de nouveau écrire les informations de la configuration de l'utilisateur, mais, cette fois, les résultats seront représenter sous la forme d'un tableau avec la plage de seeds pour les lignes et la plage des tailles de grille de cellules pour les colonnes. De plus, chaque cellule contiendra une couleur en fonction du résultat obtenu (deux fichiers montrant les deux types de visuels sont présents dans le dossier *exemples* de l'archive).
Note : les exemples ayant été faits sur Windows, il se peut que les résultats diffèrent de ceux obtenus sur Linux suivant l'implémentation de la fonction rand() sur les deux plateformes.
En plus de tout cela, le programme offrira la possibilité à l'utilisateur de sauvegarder ses données au format pdf. Cela sera seulement possible, dans le cas d'une analyse sur une seed et une taille de grille si cette dernière n'excède pas une dimension de 85, et pour la plage de données, si l'intervalle de tailles de grille de cellules n'excède pas 12.

9 Architecture logicielle

L'architecture logicielle est plutôt simple : l'intégralité du programme est découpé en 4 gros blocs distincts (les structures de données précédemment), Configuration, Grid et FileResults étant gérées par le processus 0 et Data gérée par les processus effectuant les tâches en parallèle (1 à n-1). Les communications entre les processus ne se fait que via l'échange de données via ***MPI_Send et MPI_Recv***. Du côté du processus 0, des communications s'effectuent entre les différents blocs: Grid communique au début du programme avec Configuration afin de récupérer toutes les caractéristiques du jeu de la vie, puis toutes les modifications de valeurs de taille de grilles de cellules afin de récupérer le tableau des feuillets par processus actualisé. Enfin la structure de données FileResults va communiquer à la fois avec Configuration et Grid : avec Configuration au début du programme afin de récupérer les caractéristiques du fonctionnement du jeu de la vie, et avec Grid pour récupérer les patterns finaux contenus dans les attributs *NGeneration*, *NMinusOneGeneration* et *NMinusTwoGeneration* afin d'écrire les patterns obtenus lorsqu'un configuration intéressante a été trouvée.

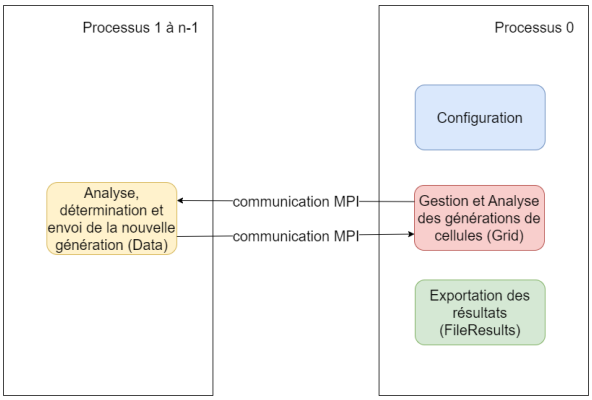


Figure 4: *Architecture du programme.*

10 Fonctionnement du programme à partir d'un exemple

Pour illustrer le fonctionnement du programme, on va partir d'un exemple tout simple : *un cube de dimension* $3 \times 3 \times 3$*, de seed 0 avec 4 processus ou cœurs de processeurs.* C'est-à-dire qu'on aura le processus 0 qui gèrera la répartition des feuillets et 3 processus qui effectueront les tâches parallélisables du programme. On utilise la règle du jeu de la vie *5766*, c'est-à-dire qu'une cellule vivante possédant 5 à 7 cellules vivantes adjacentes reste en vie et une cellule morte possédant exactement 6 cellules vivantes adjacentes revient à la vie.

Le processus 0 commence par générer la grille de cellules de la génération 1 à partir de la valeur de la seed choisie. Cette grille de cellules est stockée dans la variable *NGeneration* de l'instance de structure Grid que le processus 0 utilise. On obtient le résultat suivant:

-Génération 1-

Feuillet 0
1 0 1
1 1 1
0 0 1

Feuillet 1
1 0 1
0 1 1
0 0 0

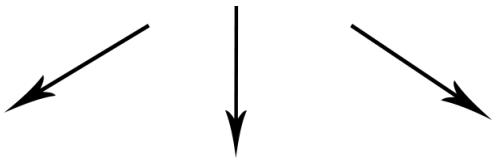
Feuillet 2
1 0 1
0 1 1
0 0 0

Ensuite, le processus va déterminer le nombre de feuillets à envoyer aux processus de rang 1 à 3. Comme on a trois plans pour trois processus, chacun d'entre eux va recevoir un feuillet. Puis, on y ajoute les feuillets voisins. Ainsi, le processus 1 qui hérite du sommet du cube (feuillet 0), va recevoir le plan du niveau inférieur (le feuillet 1), le processus 3 héritant du bas du cube (le feuillet 2) va recevoir le plan du niveau supérieur (le feuillet 1). Enfin le processus 2 n'étant ni le premier, ni le dernier processus, il va donc recevoir les feuillets des niveaux supérieur et inférieur (Feuillets 0 et 2). En outre, les processus 1 et 3 recevront donc 2 feuillets chacun et le processus 2 en recevra 3.

A partir de ces informations, le processus 0 va découper le cube en 3 chaines de caractères auxquelles il va ajouter des informations supplémentaires pour que les processus 1 à 3 puissent déterminer la

génération de cellules suivante:

processus 0



5766_3_18_101111001101011000 — *processus 1*

5766_3_27_10111100110101100010101100 — *processus 2*

5766_3_18_101011000101011000 — *processus 3*

Chaque processus va recevoir son datagramme et, grâce aux informations contenues à l'intérieur, va pouvoir reconstruire sa propre grille de cellules:

Processus 1

Feuillet 0
1 0 1
1 1 1
0 0 1

Feuillet 1
1 0 1
0 1 1
0 0 0

Processus 2

Feuillet 0
1 0 1
1 1 1
0 0 1

Feuillet 1
1 0 1
0 1 1
0 0 0

Feuillet 2
1 0 1
0 1 1
0 0 0

Processus 3

Feuillet 1
1 0 1
0 1 1
0 0 0

Feuillet 2
1 0 1
0 1 1
0 0 0

Ensuite, grâce à la fonction ***Data__cellNextGen***, chaque processus va déterminer la nouvelle génération de cellules dans chaque feuillet n'étant pas un feuillet voisin, c'est-à-dire le feuillet 0 pour le processus 1, le feuillet 1 pour le processus 2 et le feuillet 2 pour le processus 3. Les feuillets voisins servent uniquement à déterminer la nouvelle génération de cellules dans les feuillets d'intérêt et seront donc recopiés dans la grille contenant la nouvelle génération. On obtient donc après cette étape:

Processus 1

Feuillet 0
0 0 1
0 0 1
0 1 0

Feuillet 1
1 0 1
0 1 1
0 0 0

Processus 2

Feuillet 0
1 0 1
1 1 1
0 0 1

Feuillet 1
1 0 0
1 0 0
0 0 0

Feuillet 2
1 0 1
0 1 1
0 0 0

Processus 3

Feuillet 1
1 0 1
0 1 1
0 0 0

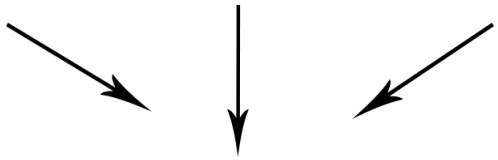
Feuillet 2
0 0 1
0 1 1
0 0 0

Puis, chaque processus ayant participé à une tâche parallèle va transformer la grille de cellules en chaine de caractères, afin d'être envoyée au processus 0, tout en ne prenant pas en compte les feuillets voisins. Cela veut dire que le processus 1 va uniquement envoyer les données contenues dans le feuillet 0, le processus 2 celles contenues dans le feuillet 1, et enfin le processus 3 les données contenues dans le feuillet 3:

processus 1 — **001001010**

processus 2 — **100100000**

processus 3 — **001011000**



processus 0

A partir des résultats obtenus, le processus 0 va reconstruire le cube, car les résultats sont reçus de façon à ce que le processus 0 sache à quelle position les feuillets reçus par tel processus sont positionnés. Ceci va constituer la seconde génération de cellules:

-Génération 2-

Feuillet 0

0 0 1
0 0 1
0 1 0

Feuillet 1

1 0 0
1 0 0
0 0 0

Feuillet 2

0 0 1
0 1 1
0 0 0

Puis, à partir de la génération nouvellement obtenue, le processus 0 va d'abord chercher à savoir si la nouvelle génération de cellules est vide ou non , c'est-à-dire si elle ne contient que des cellules mortes, ce qui se matérialise par une grille ne contenant que des 0. Comme ce n'est pas le cas, il va ensuite comparer avec les générations de cellules stockées en mémoire. Ici, seule la génération 1 est stockée en mémoire référencée par l'attribut de l'instance de la structure de Grid **NGeneration**, les valeurs des attributs **NMinusOneGeneration** et **NMinusTwoGeneration** étant quant à elles encore à NULL. Comme les générations 1 et 2 ne correspondent pas, le processus 0 va donc stocker cette nouvelle génération en mémoire. La génération 1 est maintenant référencée en mémoire par l'attribut **NMinusOneGeneration** et la génération 2 par l'attribut **NGeneration**.

Le programme vient donc ici de terminer une itération pour essayer de trouver une configuration stable ou oscillatoire à partir du pattern initial (génération 1). Comme ici, le processus 0 n'a pas trouvé de résultat probant (pas de correspondance entre la nouvelle génération et une génération de cellules stockées en mémoire ou pas de génération sans cellules vivantes), il va donc lancer la détermination d'une nouvelle génération de cellules en découpant le cube de la génération 2 et en envoyant ses feuillets ainsi que les feuillets voisins aux processus concernés. Il reproduit cela pour une génération supplémentaire et on obtient le résultat suivant:

-Génération 3-

Feuillet 0

0 0 0
0 0 0
0 0 0

Feuillet 1

0 0 0
0 0 1
0 0 0

Feuillet 2

0 0 0
0 0 0
0 0 0

Comme la grille n'est pas vide et qu'il n'y a pas de correspondance entre la génération 3 et les générations 1 et 2 stockées en mémoire, on passe à la suivante et on arrive à la génération 4 de cellules. Les générations 3, 2 et 1 sont référencées en mémoire respectivement par les attributs **NMinusTwoGeneration**, **NMinusOneGeneration** et **NGeneration** et le processus 0 obtient la génération de cellules suivantes:

-Génération 4-

Feuillet 0

0 0 0
0 0 0
0 0 0

Feuillet 1

0 0 0
0 0 0
0 0 0

Feuillet 2

0 0 0
0 0 0
0 0 0

Ici, on arrive dans un cas particulier (plus de cellules vivantes), plutôt logique compte tenu de la règle du jeu de la vie utilisée et la taille de la grille de cellules. Par conséquent on va arrêter la recherche de la génération suivante et la fonction **Grid__analyzePattern** va retourner un tableau contenant le résultat de l'analyse (ici 0 → pas de configuration stable trouvée car plus de cellules vivantes) ainsi que la génération à laquelle le résultat a été obtenu. Enfin, un message va être affiché dans la console afin d'indiquer à l'utilisateur le résultat de l'analyse.

11 Évaluation et comparaison des tests de performance

11.1 Mesure des temps d'exécution du programme

Les mesures des temps d'exécution ont été effectuées le 15 Décembre 2022 sur un cluster de machines de la salle A104. Les processus qui ont été utilisés sont des cœurs de processeurs Intel i9 10900k ayant une fréquence de base de 3.70 GHz et pouvant monter jusqu'à 5.30 GHz en mode boost. Les calculs ont été effectués avec les valeurs de cœurs suivants: **1** (afin d'avoir le temps d'exécution

séquentiel du programme), **5**, **15**, **25** et **40** cœurs. Comme ce nombre correspond aux nombres de processus effectuant les tâches parallèles, cela revient donc à prendre, en ajoutant le processus 0 effectuant toute la partie séquentielle du programme, 2, 6, 16, 26 et 41 cœurs.
La mesure du temps d'exécution du programme a été réalisée sur les échantillons suivants:

- *taille de grille de cellules de dimension 40 (*$40 \times 40 \times 40$*)- seed 46 avec 100 générations maximales*;
- *taille de grille de cellules de dimension 60 - seed 6 avec 100 générations maximales*;
- *taille de grille de cellules de dimension 80 - seed 11 avec 100 générations maximales*;
- *taille de grille de cellules de dimension 100 - seed 2 avec 100 générations maximales*;
- *taille de grille de cellules de dimension 125 - seed 0 avec 100 générations maximales*;
- *taille de grille de cellules de dimension 150 - seed 0 avec 100 générations maximales.*

Les valeurs de seeds n'ont pas été prises au hasard. En effet, pour toutes ces seeds, le programme n'arrive pas à déterminer de configurations stable ou oscillatoire au bout des 100 générations autorisées. Cela permet d'avoir une cohérence des résultats en fonction des tailles de grille car on est sûr que le programme effectuera le calcul de 100 générations de cellules à chaque fois. Pour mesurer le temps d'exécution, deux *clock()* ont été utilisées sur le processus 0 car c'est lui qui arrête les autres processus utilisés dans le programme. La première balise a été placée juste après la fin de la configuration du jeu de la vie et la deuxième juste avant la fin de l'exécution du programme afin d'avoir un durée reflétant les temps de calculs. Les temps d'exécution obtenus, données en seconde et arrondies au millième, sont les suivants:

nombre de cœurs / dimension de la grille	1	5	15	25	40
40 x 40 x 40	0.379	0.151	0.246	0.395	0.583
60 x 60 x 60	1.608	0.403	0.499	0.651	0.854
80 x 80 x 80	6.326	1.049	0.980	1.165	1.561
100 x 100 x 100	19.032	2.338	1.808	2.091	2.510
125 x 125 x 125	55.774	7.019	3.834	4.329	4.940
150 x 150 x 150	133.001	18.739	7.103	8.219	8.645

On peut, à partir de ces résultats, représenter l'évolution du temps d'exécution du programme (en secondes) en fonction du nombre de processus utilisés. Afin d'avoir une représentation générale de la courbe d'évolution du temps d'exécution, on va prendre la moyenne des des temps d'exécution pour un processus donné:

On observe que le temps d'exécution diminue jusqu'à 15 cœurs. A partir de cette valeur, le temps de d'exécution du programme ne diminue plus.

11.2 Métrique de performance - accélération

L'accélération (ou *speedup*) va servir à observer le gain de temps d'exécution du programme en fonction de l'utilisation de 1 à 40 cœurs. L'accélération pour un nombre **P** de processus (que l'on note **S(P)**), est le rapport entre le temps d'exécution du programme en mode séquentielle et le temps d'exécution avec **P** processus soit la formule suivante:

$$S(P) = \frac{T(1)}{T(P)}$$

On va également comparer les valeurs d'accélération obtenues avec l'accélération idéale du programme. Pour calculer cette valeur, on prend la valeur du temps d'exécution séquentielle du

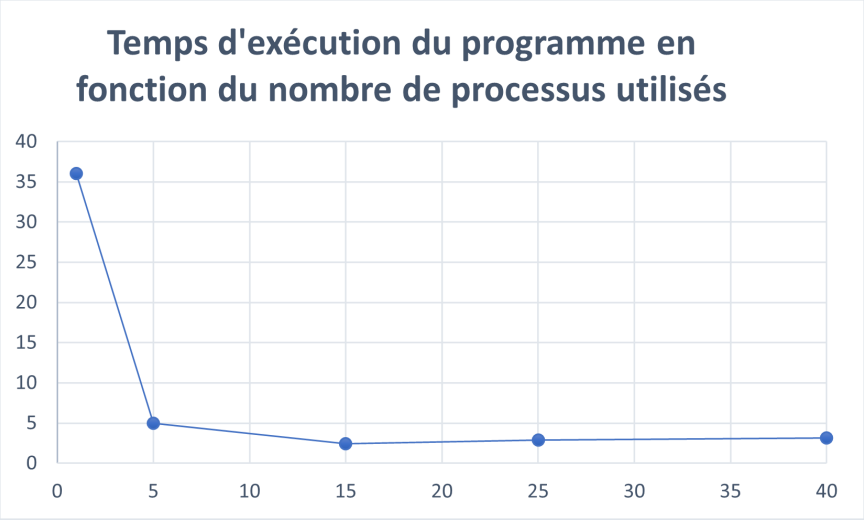


Figure 5: *Graphique du temps d'exécution moyen du programme (en secondes) pour des tailles de grilles de 40, 60, 80, 100, 125 et 150 en fonction du nombre de processus utilisés par l'utilisateur*

programme pour une taille de grille donnée (**T(1)**). Si l'accélération du programme est idéale, pour un nombre **P** de processus, l'accélération sera donc de **P x T(1)**. De plus, chaque point représentant la valeur de l'accélération en fonction du nombre de processus se fera par la moyenne de toutes les accélérations des différentes tailles de grille de cellules pour un nombre de processus donnés:

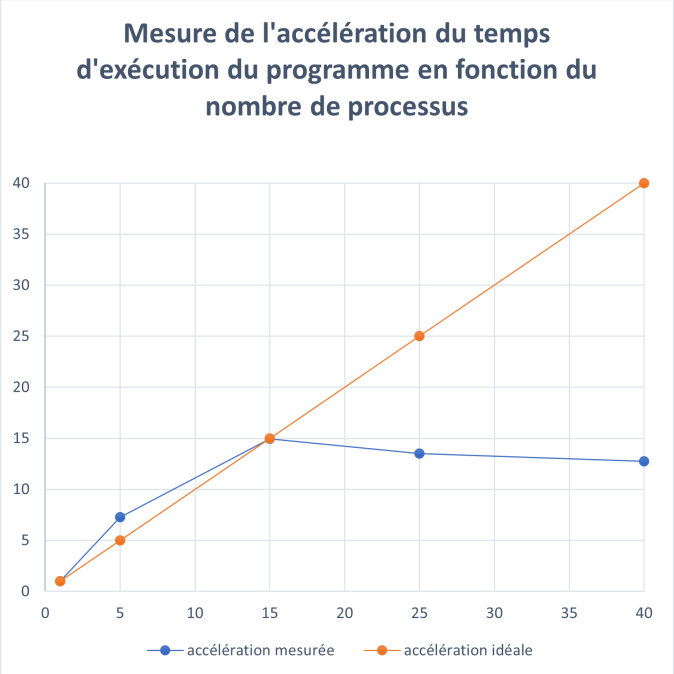


Figure 6: *Graphique représentant l'accélération théorique du programme (en orange) et l'accélération mesurée du temps d'exécution du programme (en bleu) en fonction du nombre de processus utilisés.*

On observe plusieurs résultats intéressants. Tout d'abord, on constate une hyper-accélération lors de l'utilisation de cinq cœurs (accélération mesurée qui dépasse l'accélération idéale qui est supposée maximale). Cela n'est pas normal mais peut s'expliquer par la complexité de l'algorithme permettant de déterminer la nouvelle génération de cellules (dans la fonction **Data__cellNextGen**. En effet, on avait parlé que la complexité de cet algorithme était corrélé au nombre de feuillets à analyser par le processus. Dans le cas d'une exécution séquentielle du programme, le processus effectuant les tests en parallèle va recevoir l'intégralité de la grille et va devoir déterminer la nouvelle génération de

cellules sur l'ensemble de la grille. Cela se répercutait par une complexité de l'algorithme de $O(n^3)$. En revanche, lorsqu'on utilise des processus effectuant des tâches en parallèle, on peut faire diminuer la valeur de la complexité de l'algorithme jusqu'à un minimum de $O(n^2)$ (chaque processus analyse un feuillet de la grille de cellules). En connaissant cela il est donc logique que le programme utilisant des tâches en parallèle puisse sur-performer par rapport à l'exécution séquentielle du programme. Et cela se vérifie si on représente sur un graphe l'évolution du temps d'exécution du programme en fonction des tailles de grilles de cellules utilisées et pour différents nombres de processus, et que l'on compare cela avec les courbes des fonctions $f(x) = x^2$ et $f(x) = x^3$:

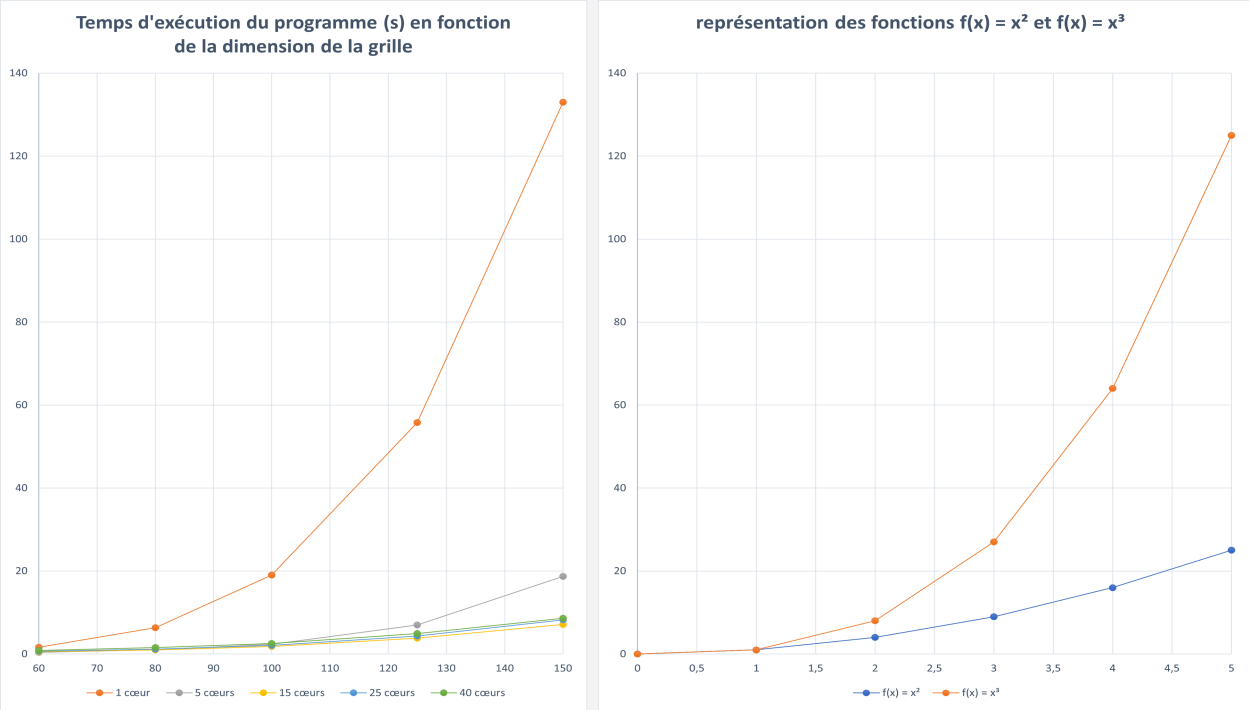


Figure 7: *Comparaison des courbes représentant l'évolution du temps d'exécution du programme (en secondes) en fonction de la taille de la grille de cellules pour différents nombres de processus (à gauche) avec la courbe représentant les fonctions x^2 et x^3 en fonction de x (à droite).*

On observe très clairement que la courbe pour 1 cœur (en orange sur le graphique de gauche) a une allure de forme cubique tandis que celles à plusieurs cœurs tendent plus à suivre une évolution quadratique. Et cela faussera les mesures d'accélération causant une d'hyper-accélération pour une exécution en parallèle.
On observe également un autre phénomène: après 15 cœurs, l'accélération freine brutalement voire diminue même pour une plus grande utilisation de cœurs. Cela peut être dû à des problèmes avec MPI comme par exemple un surcoût lié aux synchronisations ou aux communications entre les différents processus du cluster de machines ou alors un réseau d'interconnexion trop faible entre les différentes machines.

11.3 Métrique de performance - efficacité

L'efficacité correspond à la mesure du taux de ressources utilisées par le programme. Plus l'efficacité est grande, meilleure est l'utilisation des ressources. L'utilisation idéale des ressources est de 100%. L'efficacité, notée **e**, se calcule pour un nombre de processus **P** donné, par le rapport entre l'accélération pour ce nombre de processus sur le nombre de processus P soit:

$$e(P) = \frac{S(P)}{P}$$

A partir de cette formule, on obtient la courbe suivante:

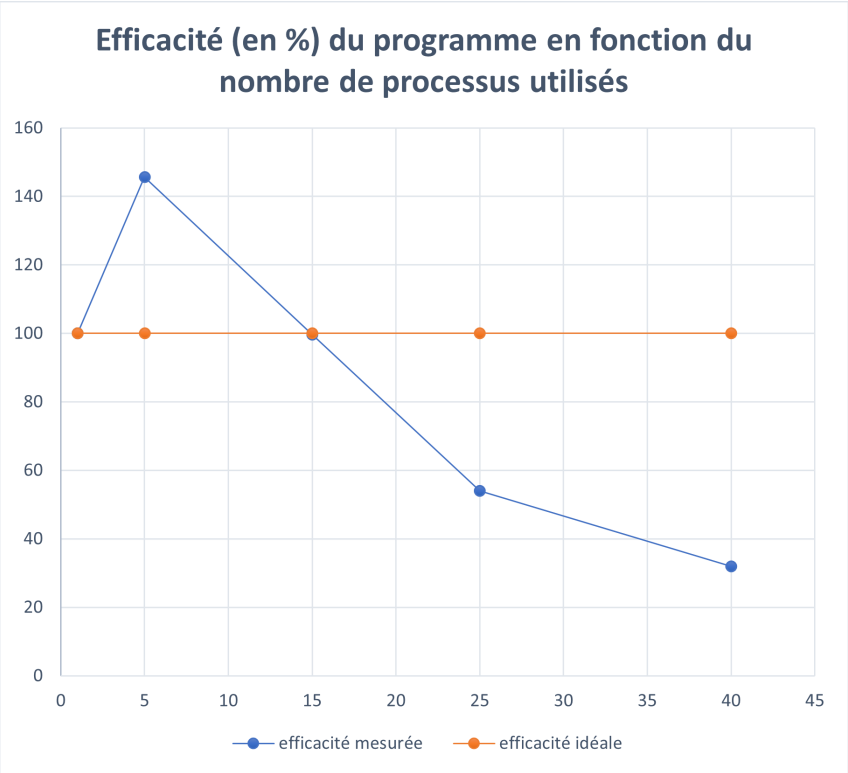


Figure 8: *Graphique de l'évolution de l'efficacité du programme (en %) en fonction du nombre de processus utilisés.*

Là encore, on se retrouve avec une hyper-efficacité du programme due, comme précisé précédemment, au changement de complexité en passant d'une exécution séquentielle à une exécution parallèle. Puis l'efficacité diminue de façon linéaire jusqu'à 25 cœurs. Et à partir de cette valeur, la perte d'efficacité commence à freiner.

Nous avons donc vu avec l'analyse des performances du programme du jeu de la vie, que le passage d'une exécution séquentielle à une exécution parallèle en utilisant MPI a eu une effet au-delà des attentes en changeant la complexité de l'algorithme de détermination de la nouvelle génération de cellules passant de $O(n^3)$ pour une exécution séquentielle et pouvant atteindre une complexité jusqu'à $O(n^2)$ pour une exécution parfaitement parallèle (pour un nombre de processus participant à la partie parallèle égal aux nombres de feuillets de la grille de cellules). De plus, avec les résultats obtenus, on observe qu'ajouter plus de processus n'améliore pas indéfiniment les performances du programme (après 15 cœurs, l'amélioration des performances est nulle). Cela peut paraitre contre intuitif car on a expliqué que plus le nombre de processus approchait le nombre de feuillets du cube, plus la complexité s'approchait de $O(n^2)$ et, par conséquent, que la vitesse d'exécution devrait être plus rapide. Cela peut s'expliquer par des problèmes de synchronisations et/ou de communication entre les différents processus présents sur différentes machines. On peut donc en déduire que pour les tailles de grille étudiées pour mesurer les performances du programme, l'exécution avec 15 processus fonctionnant en parallèle semble être le meilleur compromis en termes de performances énergétiques avec, au final, un bon temps d'exécution en général quelque soit la taille de la grille à analyser pour un coût relativement faible (seulement 15 cœurs). Si nous avions poussé les mesures sur des tailles de grille plus grandes, peut-être les résultats auraient été différents (quand on voit, par exemple, dans la *Figure 5* le temps d'exécution pour 5 cœurs qui commence à fortement augmenter entre des tailles de grille de cellules de 125 et de 150).