Le Jeu de la Vie

Rapport du Projet Tuteuré

par
Ghislain Hudry
Yves Perera
S4p1A”

22/03/2013

Sommaire :

Introduction :

Lors du quatrième semestre de notre DUT en informatique, nous avons eu un projet tuteuré (PT) à effectuer par binôme.
Le sujet sur lequel nous devions travailler était au choix et ce choix dépendait de notre classe et professeur.
Dans notre cas, les deux sujets proposées étaient le jeu de la vie et la réalisation d'un gestionnaire d'archive (dossier compressé).

Nous avions huit séances de deux heures encadrées et sept séances non encadrées pour réaliser le sujet choisi et rédiger un rapport.

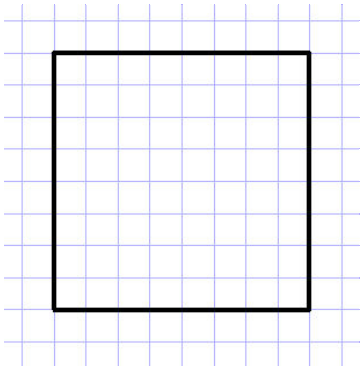
I) Présentation :

Notre binôme est formé de Ghislain Hudry et Yves Perera et nous avons choisi le sujet qui consiste à développer un "Jeu de la Vie".

Voici ce qu'est le Jeu de la Vie :

- Règles du jeu :

Pour commencer prenons un quadrillage :

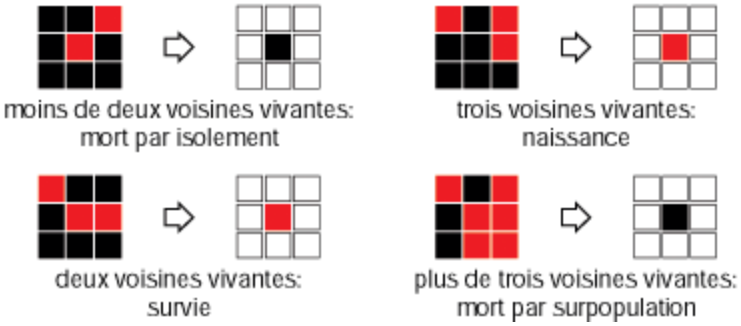


Nous allons d'abord nous intéresser à chaque cellules de ce quadrillage indépendemment.

Chacune de ces cellules ne peut avoir que deux états : elle peut être soit vivante ou morte.

Voyons maintenant les conditions de vie ou de mort d'une cellule :

Pour cela nous devons élargir notre zone d'étude aux cellules voisines de celle qui nous intéresse et considérer l'état de ces ces huit autres cases.
La vie ou la mort de notre cellule et alors fonction du nombre de cellules voisines qui vivent.
L'état change alors suivant ces lois :



L'état de chaque case de la grille du jeu au départ est déterminé aléatoirement et une fois le jeu lancé, il s'exécute tout seul jusqu'à ce que l'on décide de l'arréter.
Chaque instant fait l'objet de calculs qui re-déterminent l'état des cases.

Lorsque le jeu dure assez longtemps, la grille se stabilise et l'on voit alors apparaître des "structures stables" (ce sont des configurations qui permettent la survie des cellules) qu'il est

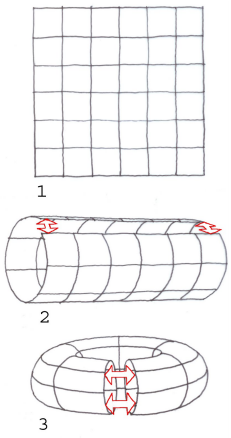
intéressant d'observer.

- Contraintes :

Le projet consiste donc à créer un Jeu de la Vie. Cependant il y a des contraintes.

- La première étant le choix du langage de programmation... Nous fûmes obligé de développer ce jeu en langage C ou C++.

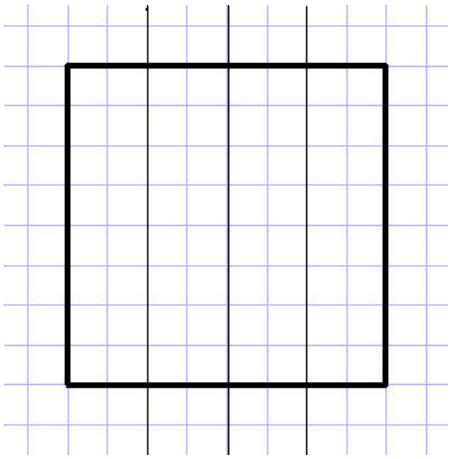
- Nous devions aussi considérer notre quadrillage comme un tore :



Cela signifie que l'on doit considérer les coté de notre carré comme joint deux à deux. Du coup les cellules de la rangée du haut de notre quadrillage prennent en compte l'état des cellules de la rangée du bas (et vice-versa). Idem pour les colonnes de droites et de gauche.

Pour réaliser ce tour de magie, nous avons défini une grille ayant deux colonnes et lignes supplémentaires que celle que nous affichons vraiment. Elles servent à copier les valeurs des lignes et colonnes qui leurs sont opposées.

- Parallélisation : Par essence, le jeu de la vie peut-être paralléliser, nous avons donc du multithreader l'application pour en augmenté l'efficacité.



Pour comprendre les threads dans le cas de ce projet, il faut imaginer notre grille séparé en quatre comme ci-contre.

A la base, on pourrais se dire que l'ordinateur prend le temps de parcourir toute la grille pour déterminer l'état de chaque case.

Cependant lorsque l'on utilise les thread, le calcul se fait différemment, chaque thread gère une partie de la

grille et les calculs de chaque quart du carré sont effectuées simultanément : Il faut considérer chaque thread comme un coeur.

- Et enfin, la stagnation: si rien ne se passe dans un thread, alors tant qu'il n'y a pas de perturbation (changement d'état sur les bords du thread en question), alors le thread n'a aucun calcule à faire. Le thread doit donc être capable de détecter si il y a des changement à l'intérieur de sa partie de la matrice ou sur ses bords, et en fonction, effectuer les calcules ou pas.

II) Implémentation :

- Version de base:

Tout d'abord, afin de répondre exactement au concept du jeu de la vie, nous nous sommes dit que il fallait obligatoirement deux matrices afin que le résultat du calcul globale de la matrice ne dépende pas du sens dans lequel le programme parcourt cette dernière.

Cependant, avoir deux matrices implique de faire une copie de l'une dans l'autre à un moment ou un autre de l'exécution. Nous avons donc pensé à, non pas à chaque fois effectuer les calcules sur la même matrice puis copier dans l'autre, mais plutôt inverser leur rôle à chaque itération.

Imaginons nous avons une matrice M1 et une matrice M2 à un état E0, à la première 'itération, on calcule l'état E1 à partir de M1 dans M2, puis à la deuxième itération, on calcule E2 à partir de M2 dans M1.

Donc en définitive, cette astuce a trois avantages:

- pas besoin de copier toute la matrice.
- pas besoin de créer de sémaphore pour chaque thread sur une partie de la matrice.
- on stock l'état précèdent de la matrice, ce qui sera très utile, nous le verrons par la suite, pour la stagnation.

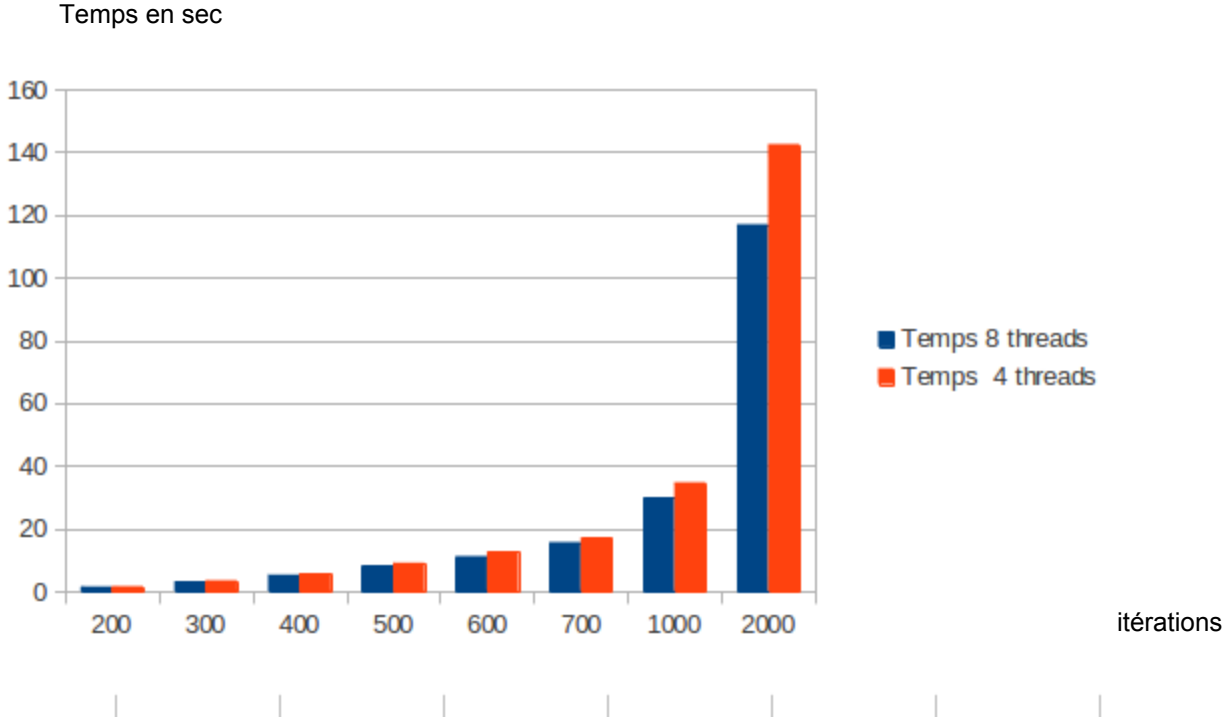
Le seul désavantage que l'on peut trouver et au niveau de la quantité de RAM utilisé, cependant, nous avons fait une structure qui permet de coder chaque cellule sur un seul bit, nous aurions donc pu mettre les deux matrices dans la même variable en s'aidant d'une sémaphore, mais le temps nous manquait et le programme aurait perdu un peu de performance

La simulation de la topologie torique de la matrice a été facilement implémenté.
Les bords de la matrice sont réservé pour l'émulation de cette topologie, à chaque itération le

bord droit est copié vers le bord gauche etc. puis les coins sont copié deux à deux haut gauche vers bas droit etc. En effet l'utilisation des deux matrices nous a permis d'implémenter facilement cette aspect du projet.

A cette étape du projet nous devions mettre en place la parallélisation. Une fois de plus l'utilisation d'une double matrice nous a facilité la tâche, les threads n'ont pas besoin de sémaphore étant donné qu'il n'accèdent en écriture qu'à leur propre partie(et non aux bords de leurs camarades de jeu).

Cependant pour que le calcule reste uniformément répartie dans le temps par rapport à l'application du tore et l'inversion des rôles des matrices, nous avons été contraint d'utiliser une "barrier". Nous avons tout de même obtenue des gains de performance non négligeables sur un ordinateur i7 (4 coeurs, 8 coeurs logiques):



- Raffinement:

Cette étape est sensé être une amélioration du code permettant de rendre l'éxécution plus rapide cependant (comme dit précédemment), l'architecture que nous avons choisi pour réaliser ce projet nous permet de ne pas avoir besoin de l'implémenter.

Le principe de cette amélioration est simple :

Tout d'abord il faut se rappelé le système des threads qui divisent notre quadrillage (en quatre ou plus) dans le but de pouvoir effectuer les calculs simultanément.

Essayons d'imaginer comment chaque thread calcule les cases qui lui sont affectées :
Le thread part d'en haut à gauche et parcours sa partie de grille normalement.

Cependant cela pose un problème...
Car chaque thread a besoin des informations de la bordure des threads qui l'entour. Le principe de l'étape de raffinement est donc de forcer les threads à calculer leurs bords avant leur centre car une fois les bords calculés chaque thread à toutes les informations qu'il lui faut pour continuer ses calculs tout seul.

Effectivement cette amélioration semble approprié à notre application mais comme je l'ai préciser plus haut, elle n'a pas lieu d'être.
Rappelez vous la manière dont nous passons d'une itération à l'autre :

- Nous remplissons une première matrices (aléatoirement) lors de la première itération.
- Puis nous effectuons les calculs par rapport à l'état des cellules de cette première matrice et copions les résultats de ces calculs dans une seconde matrice (pour ne pas écraser nos premières valeurs et risquer de fausser nos calculs).
- Une fois la seconde matrice remplie, on interverti les rôles et on réécrit par dessus les valeurs de la première matrice.

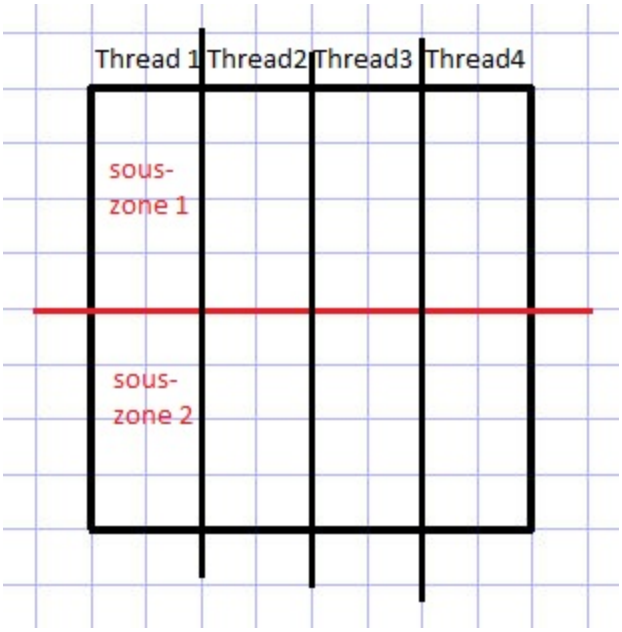
En procédant de cette manière, les threads peuvent parcourir normalement le bout de quadrillage qui leur est assigné, sans avoir à attendre de résultats venant des autres threads car on travail à chaque instant sur une matrice complète.

- Stagnation :

La stagnation est encore une forme d'amélioration et c'est la troisième étape de notre sujet. Son but est de faire économiser des calculs à notre application.
Pour ce faire, on va essayer d'anticiper l'itération suivante. On calcule toutes les cases autour de la zone sur laquelle on se concentre et si aucune des cases ne change d'état ni sur les bords ni dans la zone, on en déduit que notre case ne changera pas non plus d'état.

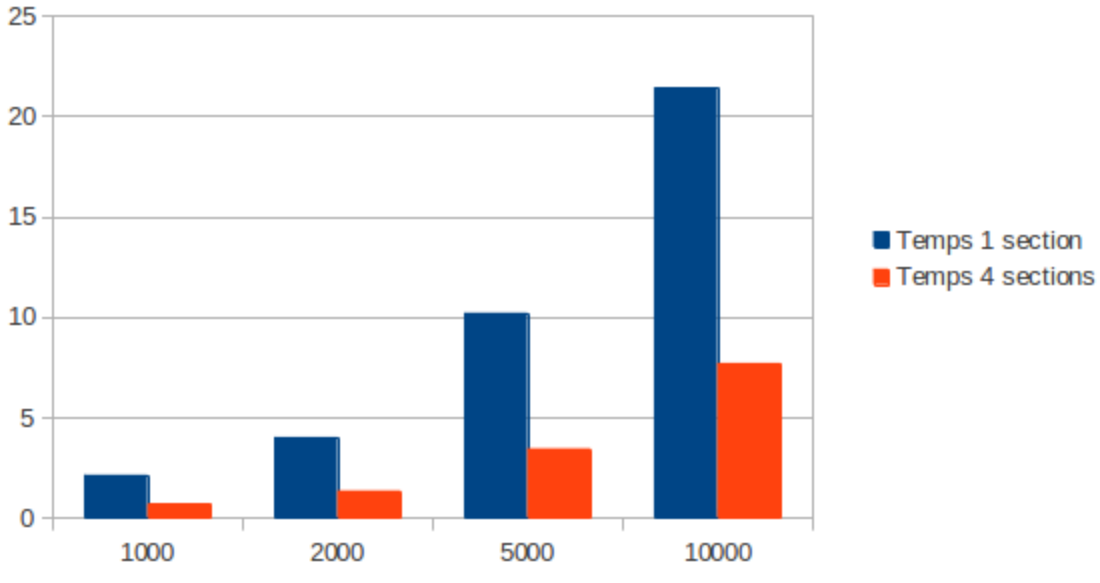
Au départ la stagnation devait être appliqué juste une fois à chaque thread, c'est à dire que, un thread a une zone de Stagnation. Cependant, cette solution n'entraîne que des gains minime sur des matrices lambda, ou alors de gros gains de temps sur des matrices totalement vides. En effet, pour que cela soit rentable il fallait que le thread qui effectue des calcules en moins soit un thread qui à plus de calcule à faire que les autres (matrice impaire etc...), car avec la barrier dans tous les cas on attend le thread le plus long.

Nous avons alors pensé à faire plusieurs sous zone de stagnation dans chaque thread, ici nous avons mit deux sous zones:



Dans ce cas, si chaque thread à une (ou plusieurs) sous zone inactive, alors le gain en vitesse est vite perceptible:

temps en sec



Tests effectués sur un ordinateur i7, matrice de 250x250, 4 threads à l'aide d'un pattern qui active uniquement 1 ou deux sections dans le cas de 4 sections, et active touts les sections si il n'y en a qu'une par thread. On peux donc voire que le temps l'exécution est presque divisé par 4 (presque, car il y a toujours le calcule des bords quoi qu'il se passe).

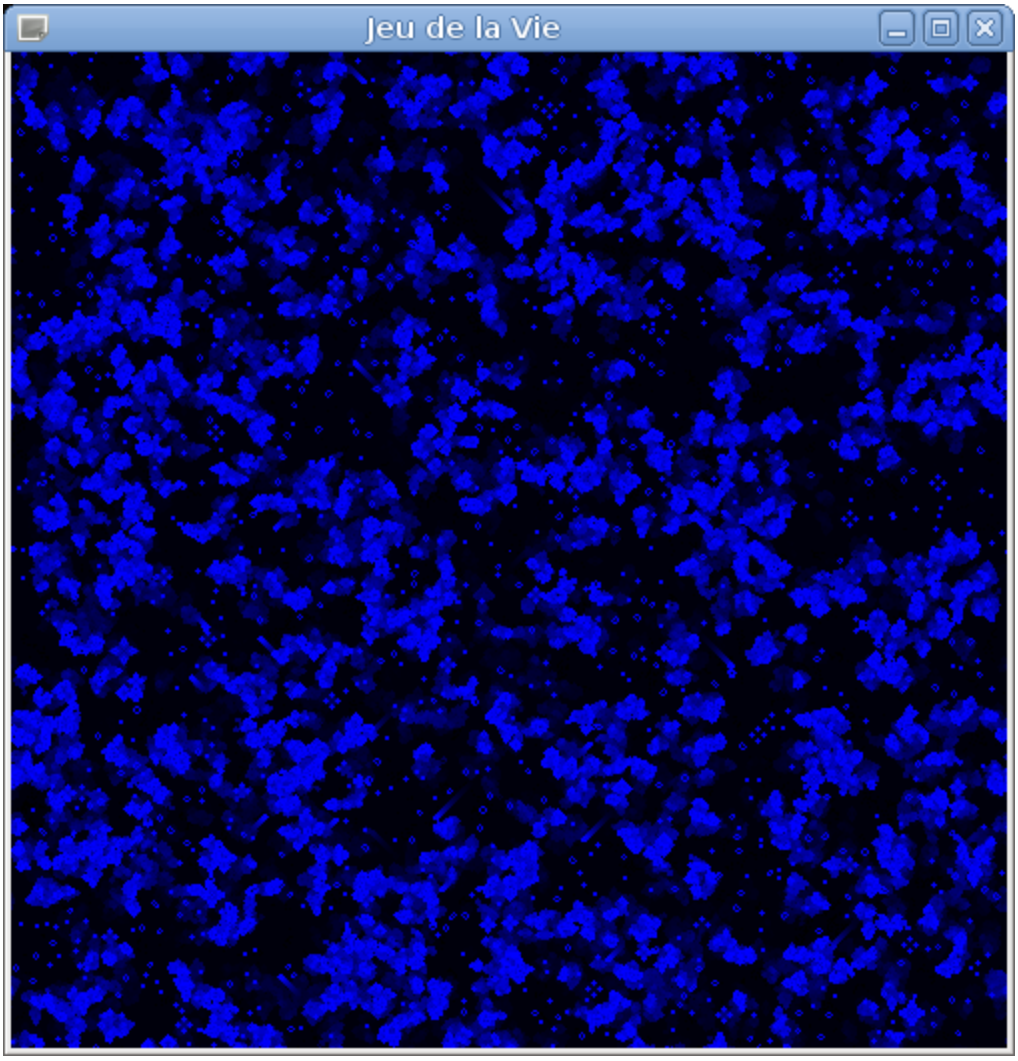
Selon la taille de la matrice il est donc intéressant de choisir un nombre de sous zone de stagnation en adéquation: Si il y en a trop peu, la plupart des threads feront tous les calcule et il y aura donc aucun gain de temps globalement, si il y en à trop, le gain de temps sera minime car il y auras plus de bords à calculer.

III) Décisions prises :

Notre décision principale fut de faire deux matrices pour gérer le passage d'une itération à une autre car le sujet ne nous orientait pas dans cette direction cependant cette solution nous a grandement aider tout au long de la réalisation de ce projet.

Nous avons aussi hésité lors de l'implémentation de la fonction tore car au départ nous ne voyions pas l'intérêt de prendre en compte les quatre coins extérieurs à l'affichage de notre jeu. Effectivement ce n'est pas évident de se représenter un tore à partir d'un quadrillage et nous étions du coup dans l'erreur. Nous nous en sommes aperçu en exécutant le programme et en notant de grosses incohérences sur les coins.
En fait la case extérieur en haut à gauche représente le pixel qui est en bas à droite sur l'affichage. Et donc c'est un pixel qui doit être pris en compte dans les calculs.
Dernière précision (qui a son importance) il faut faire attention à ne pas compter ces coins deux fois !

Nous avons aussi pris la décision de pousser le code de couleur un peu plus loin qu'un simple affichage ou il n'y a que deux couleurs (cellules vivantes / cellules mortes). En effet à chaque itération on ne remet pas à zéro l'affichage (fond opaque), on applique un fond noir partiellement transparent entraînant un effet de flou ressemblant à une persistance rétinienne.



IV) Résultats :

A la fin de ces 14 séances, nous avons donc un exécutable de notre propre Jeu de la Vie ! Il se compile dans un terminal via un makefile de trois façons différentes:

- On peut le compiler en mode graphique pour avoir un affichage réalisé en SFML.
- On peut le compiler en mode shell, utile pour calculer le temps d'exécution.
- Ou en mode debug.

Utilisation: make [option]
option: x : mode graphique
rien:mode shell
debug: mode debug

Ensuite notre programme peut être exécuté avec plusieurs arguments:
App rien: matrice par défaut, 250x250, 4 threads, 1000 itération, 2 sous zone.
App <numéro du pattern> → 0 pour une matrice aléatoire, 1 pour une matrice vide, 2 pour un pattern activant 4 threads, 3 pour un patterne 2 avec 4 sections de stagnation.
App <taille matrice> <nombre de threads> <nombres d'itérations>
App <taille matrice> <nombre de threads> <nombres d'itérations> <nombre de sous-zone par thread>

D'autres variables peuvent être modifiés dans le fichier de Constantes. Tel que la taille des cellule (en pixels), le temps d'attente entre deux affichages, la transparence afin d'avoir un effet de flou, tout cela en mode graphique bien sûre.

Conclusion :

Au fur et à mesure du projet, il y eu des moments ou l'on ne voyais pas quoi rajouter ou faire de plus mais l'on s'est rendu compte que l'on peut toujours améliorer et optimiser le programme, nous avons donc d'autres idées que nous n'avons pas eu le temps d'implémenter, tel que stocker directement dans la matrice le nombre de voisins de chaque cellule, et après chaque calcule, incrémenter ou décrémenter ce nombre suivant le changement. Ce qui aurais permis d'éviter beaucoup de calcule mais aurais utilisé beaucoup plus d'accès mémoire (cette solution n'aurais pas consommé plus de place dans la RAM grâce à notre structure Data qui a 7bits de libre, sachant qu'il ne peut y avoir que 8 voisins au maximum).

Nous sommes content de ce que nous rendons et nous en avons tiré de l'expérience à travers les problèmes qui sont survenu et les réponses que nous avons du y apporter.