Programmation Parallèle

Fiche *Projet* : Le jeu de la vie

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

Résumé

Ce projet porte sur la simulation parallèle du jeu de la vie, inventé par J. Conway en 1970, qui consiste en un ensemble de régles qui régissent l'évolution d'un automate cellulaire.

Cette simulation s'effectuera à l'aide de l'environnement EASYPAP, dans lequel une version séquentielle de la simulation vous est fournie.

1 Cadre du projet

Ce projet est à faire de préférence en binôme. La date prévu pour le rendu est le 20 mai. Votre rapport, sous forme de fichier au format PDF, contiendra les parties principales de votre code, la justification des optimisations réalisées, les conditions expérimentales et les graphiques obtenus accompagnés chacun d'un commentaire le décrivant et l'analysant.

Idéalement les machines utilisées pour faire des expériences doivent être des machines du CREMI dotées d'au moins 12 cœurs physiques (machines de la salle 008 et 203). Un guide de travail à distance au CREMI est disponible à cet endroit :

https://gforgeron.gitlab.io/ssh/guide.pdf

Attention à utiliser des machines non chargée, de manière à ne pas bruiter les expériences.

2 Description du modèle

Le jeu de la vie simule une forme très basique d'évolution d'un ensemble de cellules en partant de deux principes simples : pour pouvoir vivre, il faut avoir des voisins ; mais quand il y en a trop, on étouffe. Le monde est ici un grand damier de cellules vivantes ou mortes, chaque cellule étant entourée par huit voisines. Pour faire évoluer le monde on découpe le temps en étapes discrètes et, pour passer d'une étape à la suivante, on compte pour chaque cellule le nombre de cellules vivantes parmi ses huit voisines puis on applique les règles suivantes :

— Une cellule morte devient vivante si elle a exactement 3 cellules voisines vivantes — autrement elle reste morte.

— Une cellule vivante reste vivante si elle est entourée de 2 ou 3 cellules vivantes — autrement elle meurt.

Cet ensemble de règles est communément appelé « B3/S23 » (*BIRTH if 3 neighbors / SURVIVE if 2 or 3 neighbors*). La simulation est synchrone : à chaque étape, l'état d'une cellule dépend uniquement des états de ses voisines **à l'étape précédente**.

La figure 1 illustre le fonctionnement de ces règles sur des configurations très simples.

Vous pouvez avoir un aperçu du jeu de la vie en EASYPAP en visionnant ce diaporama.

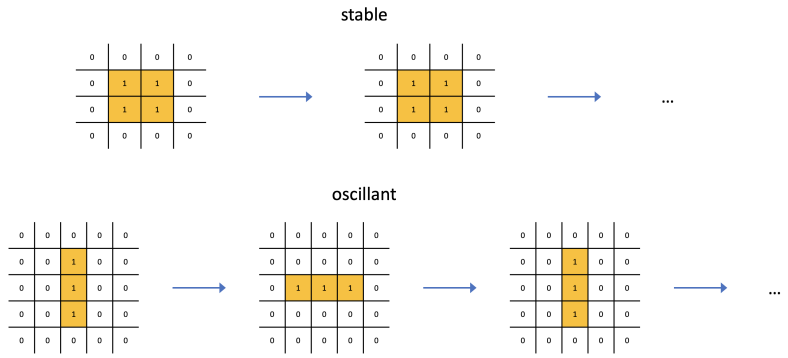

FIGURE 1 – Configurations simples et persistantes du jeu de la vie

3 Objectifs du projet

Le but de ce projet est de simuler le plus vite possible le jeu de la vie en utilisant les capacités parallèles du matériel.

3.1 Version OpenMP de base

Dans un premier temps, on se concentrera sur la parallélisation des calculs à l'aide d'OpenMP. S'il est assez immédiat d'obtenir une première version OpenMP opérationnelle, il vous est demandé de l'optimiser en explorant différentes tailles (ou même formes) de tuiles, différentes stratégies de répartition des blocs, etc.

Pour cela, vous pourrez avantageusement tirer parti des traces d'exécution que EASYPAP peut générer pour vous, et que EASYVIEW permet d'observer (lire la section 3.2 « *Post-mortem trace analysis* » de la documentation).

Pour vos expériences, vous pouvez commencer avec la configuration par défaut (4 lanceurs qui propulsent des glisseurs vers le centre), en choisissant la taille que vous souhaitez. Par exemple :

```
./run -k life -s 2048
```

Par la suite, vous pourrez expérimenter des configurations plus gourmandes :

```
./run -k life -s 2176 -a otca_off -r 10
```

Ou la terrible :

```
./run -k life -s 6208 -a meta3x3 -r 50
```

Notez que le paramètre `-r` (*refresh rate*) est optionnel et sert juste à ne pas trop ralentir la simulation avec l'affichage.

Dans cette première partie on se « contente » donc de produire une version OpenMP propre et efficace, en s'inspirant du noyau `blur` vu en TP.

3.2 Optimisations

Dans un second temps, il vous est demandé d'introduire des optimisations agressives dans le code. Vous pourrez, par exemple, explorer l'une des voies suivantes :

Évaluation paresseuse En remarquant que pour beaucoup de configurations, certaines régions du domaines restent stables (i.e. aucune cellule ne change d'état) durant plusieurs itérations, il est possible de mettre en place une stratégie paresseuse qui ne calculent pas les tuiles pour lesquelles on est sûr que leur contenu ne peut pas changer à l'itération suivante.

Vectorisation et empreinte mémoire réduite En notant qu'il n'est pas nécessaire d'utiliser des cellules codées sur 32 bits pour manipuler des booléens, un bon compromis est d'utiliser un octet (`char`) par cellule et d'utiliser des opérations de vectorisation explicites (*intrinsics*) pour manipuler des vecteurs de 32 cellules en AVX2 par exemple.

4 Travail d'expérimentation et d'analyse

Il est nécessaire de vérifier la correction de votre code avant tout travail d'expérimentation et d'analyse. Pour cela vous devez générer[1] des images de références (une fois pour toute, en utilisant la variante séquentielle fournie) et comparer pour chaque variante les images obtenues aux références (à l'aide de la commande `diff`). Nous vous conseillons d'utiliser pour références les configurations obtenues à partir de `otca_off` après 500, 1000 et 50000 itérations. Il peut être aussi intéressant d'utiliser des configurations très petites (64×64).

Votre rapport démontrera l'intérêt des optimisations apportées à vos variantes successives. N'oubliez pas de comparer les différentes stratégies que vous aurez essayées. Enfin, il s'agira de produire et d'analyser des graphiques montrant l'accélération obtenue en fonction du nombre de threads utilisés par rapport à la version séquentielle optimisée.

Pour toutes les expériences on utilisera de préférence les deux configurations suivantes :
`./run -k life -a random` : configuration pseudo-aléatoire ; on étudiera les cas où le paramètre `size` aura pour valeur 512, 1024, 2048, 4096 et 8192. Le nombre d'itérations, fonction de la surface de l'image, sera égale à $\frac{100 \times 8192^2}{size^2}$.
`./run -k life` : on étudiera les cas 960, 1920 et 3840 avec 8000 itérations.

5 Quelques indications

On trouvera des éléments méthodologiques dans le document https://gforgeron.gitlab.io/pap/td/easyplot/Sujet/tuto.pdf.

5.1 Éléments d'analyse

Il s'agit de raisonner à partir de la description des courbes et des traces, de formuler des hypothèses pour expliquer les comportements observés et, idéalement, de valider/infirmer les hypothèses à l'aide de nouvelles expériences. Voici quelques exemples d'observations et de pistes d'interprétations :

1. L'option `--dump` permet de sauvegarder la dernière image calculée sous forme d'image PNG.

— Une accélération supérieure au nombre de cœurs est constatée. Avez-vous utilisé le meilleur algorithme séquentiel ? Gagne-t-on grâce au non-déterminisme du parallélisme ? Bénéficie-t-on d'effets de cache favorables ou d'un meilleur débit mémoire ?
— On observe qu'une distribution dynamique est beaucoup moins performante qu'une distribution statique. Est-ce un problème de contention sur un mutex ? Est-ce un problème de taille de tuile mal calibrée ?
— On constate qu'une distribution statique est beaucoup moins performante qu'une distribution dynamique. La charge de travail est-elle bien équilibrée ? La machine est-elle occupée par d'autres processus ?
— On observe que la courbe d'accélération croit puis décroit. Y'a-t-il suffisamment de travail pour compenser le surcoût de la parallélisation ? Y-a-t-il un déséquilibre dû à l'hyperthreading ? Y a-t-il de la contention (synchronisation / cache / mémoire) ?

5.2 À propos des fonctions génériques

Dans le code fourni, certaines fonctions sont appelées de façon générique ; lors d'un appel

```
./run -k kernel -v version -a config
```

le programme cherchera à d'abord à utiliser la fonction `kernel_fun_version()` ou, à défaut, `kernel_fun()`. Voici les fonctions concernées :

kernel_init_version() pour allouer les structures de données ;

kernel_ft_version() pour appliquer une stratégie de placement de type *first touch* ;

kernel_draw_config() pour appliquer une configuration initiale ;

kernel_compute_version() pour calculer un certain nombre d'itérations ;

kernel_refresh_img_version() pour créer une image à partir d'une configuration ;

kernel_finalize_version() pour libérer les données allouées dynamiquement.

L'utilisation de ces fonctions est discutée dans la documentation (section 6.1 « Initialization Hooks »)