DM de PAP: Le jeu de la vie parallèle

À effectuer en binômes.

Le jeu de la vie, inventé par J. Conway en 1970, simule une forme très basique de vie 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 un grand damier torique (i.e. un carré de $N \times N$ cellules dont on considère que les bords se replient) qui contient des cellules vivantes ou mortes. Le temps est découpé en étapes : à chaque étape, pour chaque cellule c, on compte le nombre de cellules voisines (parmis les 8) qui sont vivantes et on en déduit si la cellule survit ou si elle meurt[1].

L'archive `life.tar.gz` disponible à l'adresse
`http://dept-info.labri.fr/ENSEIGNEMENT/pap/projet` contient deux répertoires. L'un contient les fichiers nécessaires à la compilation de la version `pthread`, l'autre est dédié à la version `MPI` du jeu de la vie. Chacun de ces répertoires contient une version séquentielle du jeu de la vie (fichier `life_seq.c`).

Le monde y est représenté par un grand tableau de booléens `int board[][]` de taille $N \times N$, chaque case valant 1 si une cellule y vit, 0 sinon. Pour chaque étape (le contenu de la grande boucle for), on calcule le nombre de voisins de chaque cellule dans le tableau `num_neigh`, puis l'on calcule le nouvel état des cellules vivantes. Pour simuler l'aspect thorique du monde, plutôt que traîner des `%N` partout, le tableau est indicé de `1` à `N`, et à chaque tour de boucle on copie la ligne 1 dans la ligne `N+1`, et la ligne N dans la ligne `0`, et de même pour les colones. Dans la version fournie, un petit tas de cellules est placé aléatoirement. En principe il se déplace de manière régulière. Cela vous permettra de vérifier que votre programme fonctionne bien.

Comme le jeu de la vie est par essence parallèle (chaque cellule devient vivante ou meurt en ne se souciant que de ses voisines proches), on se propose de le paralléliser pour accélérer son exécution sur des architectures parallèles.

Pour chaque partie de ce DM, on tracera une courbe de speedup pour comparer les performances obtenues.

Pour les mesures, il faudra bien sûr utiliser un N bien plus grand et désactiver l'affichage !

1 Version simple

Dans une version OpenMP, la directive `#pragma omp parallel for` effectue une barrière de synchronisation globale. C'est certes simple à implémenter, mais c'est abusif : les threads n'ont vraiment besoin de se synchroniser qu'avec les threads « voisins ». On atteint ici les limites de la facilité d'expressivité d'OpenMP.

Écrivez donc une version utilisant des `pthread`s. Il vous faudra déplacer le code de la boucle `for(loop)` depuis `main` vers une fonction exécutable par un thread. `main` se chargera désormais de créer un certain nombre de threads, chacun exécutant la boucle `for(loop)` sur une portion du tableau `board[][]`. Pour synchroniser les threads entre eux pour respecter les dépendances de données, vous pourrez utiliser des sémaphores (`sem_init`, `sem_post`, `sem_wait`).

[1] Pour information : si la cellule c était morte mais qu'exactement 3 cellules voisines étaient vivantes, la cellule c devient vivante, sinon elle reste morte. Si la cellule c était vivante : si seulement 0 ou 1 cellule voisine était vivante, la cellule c meurt d'isolement ; si 2 ou 3 cellules voisines étaient vivantes, la cellule c survit ; si 4 ou plus cellules voisines étaient vivantes, la cellule c meurt par étouffement.

2 Raffinement

En y regardant de plus près, on peut s'apercevoir que cette synchronisation pourrait être scindée en deux temps si chaque thread pouvait distinguer le calcul des cellules « intérieures » à sa zone du calcul des cellules sur les bords (qui devront par conséquent attendre des résultats des voisins et du centre). Ainsi, à chaque itération, chaque thread pourrait d'abord calculer les bords de sa zone, puis signaler qu'il terminé ce calcul, puis calculer l'intérieur et enfin attendre que ses voisins aient terminé leurs bordures respectives.

Quel est l'intérêt d'une telle décomposition de la synchronisation ? Modifiez votre code en conséquence.

3 Stagnation

Il arrive assez souvent que l'état de toute une zone reste inchangé après une étape de calcul (lorsqu'il n'y a aucune cellule vivante, par exemple). Cette zone ne changera donc plus tant que les bordures ne changeront pas non plus. Pour de telles zones, on peut économiser du temps de calcul en ne faisant simplement rien !

Est-ce que le temps de calcul global de la simulation va s'en trouver accéléré ? Discutez en fonction du nombre de threads par rapport au nombre de processeurs et du jeu de données.

4 Version MPI

Nous nous intéressons maintenant à l'implémentation d'une version MPI du jeu de la vie, pour tirer parti du cluster de machines Infini et de son réseau Infiniband. On pourra bien sûr garder sous les yeux les pages de man de ces fonctions MPI, mais sur `http://mpi.deino.net/mpi/mpi_functions/` la documentation est plus fournie et d'autres exemples de codes sont disponibles. Une documentation très détaillée est disponible sur `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html` .

4.1 Avant toutes choses

Pour pouvoir éviter de taper son mot de passe, générez une paire clé publique/privée :

```
ssh-keygen -t dsa
```

Validez autant de fois qu'il le faut. Utilisez une passphrase vide, à moins que vous sachiez gérer un agent ssh. Enfin, autorisez l'utilisation de la clé :

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Vérifiez avec un `ssh` que vous n'avez pas à taper de mot de passe. Il faudra tout de même taper yes pour confirmer l'identité de chaque machine la première fois que vous vous y connectez.

4.2 Version de base

Pour l'instant faisons simple : un seul thread par machine.

Chaque noeud va devoir traiter une portion du tableau `board[][]`. Les processus MPI voisins devront s'échanger les résultats obtenus sur les frontières de leurs portions avant de passer à l'étape suivante.

4.3 Et les cœurs ?

On se souvient maintenant que nos machines possèdent 8 cœurs, il serait sans doute intéressant d'en profiter !

Une solution simple est de ne pas modifier le programme ! Modifiez le Makefile pour donner faire varier la valeur affectée à l'option `-np` . On lance ainsi plusieurs nœuds MPI par machine, on profite ainsi des différents cœurs.

En particulier, évaluer l'impact de l'hyperthreading disponible sur les machines Infini en affectant deux processus par cœur.

4.4 Et l'équilibrage de charge ?

L'algorithme du jeu de la vie peut faire apparaître des irrégularité entre les temps de calcul des différentes portions de tableau. En particulier, si le contenu d'une portion ne change pas à l'étape i, calculer cette même portion à l'étape i+1 demandera peu d'effort car dans ce cas seules les frontières sont à recalculer. Certains cœurs peuvent donc se retrouver inactif alors que d'autres n'ont pas fini leur travail.

Discutez (sans coder) de l'élaboration d'un procédé pour répartir efficacement la charge sur le cluster de machines Infini.

5 Notes à propos du rendu

Pour le rendu du DM, il faudra rendre le code des différentes versions produites bien sûr, mais aussi un rapport. Ce rapport doit à la fois discuter des choix d'implémentation et montrer des courbes de **speedup**, mais aussi et surtout les commenter. De bonnes courbes non commentées ne valent pas grand chose. De mauvaises courbes bien commentées valent quelque chose. Commenter, cela veut non seulement dire l'allure général de la courbe (par exemple *« comportement linéaire jusqu'à un palier à partir de x »*), mais aussi donner votre avis : pourquoi la courbe a-t-elle cette forme, pourquoi c'est mieux que les précédentes, ou bien pourquoi elle est mauvaise.

6 Notes pour travailler

Pour pouvoir effectuer des mesures, venez travailler physiquement dans la salle 203 du CREMI pour disposer des mêmes machines que d'habitudes (8 cœurs). Au moment des mesures, vérifiez avec la commande `who` que vous êtes seul sur la machine, et avec la commande `top` que vous n'avez pas de processus fou qui traîne.