Examen Machine

Juvigny Xavier

21 Janvier 2010

1 Préambule

Les versions séquentielles des programmes à paralléliser peuvent se trouver sur Github. Avant de commencer l'examen, tapez dans une session shell la commande suivante :

```
git clone https://github.com/JuvignyEnsta/ExamenMachine2021
```

pour récupérer les codes sources ainsi que le makefile correspondant.

2 Introduction

- Quel est le nombre de cœurs physiques de votre machine ?
- Quel est le nombre de cœurs logiques de votre machine ?
- Quelle est la quantité de mémoire cache L2 et L3 de votre machine ?

3 Suite de Syracuse

La suite de Syracuse est une suite d'entiers naturels définie de la manière récurente :

- On choisit un nombre entier u_0 plus grand que zéro ;
- Si u_{n-1} est pair, $u_n = \frac{u_{n-1}}{2}$;
- Si u_{n-1} est impair, $u_n = 3u_{n-1} + 1$.

La suite de Syracuse possède un "cycle trivial". En effet, en prenant $u_0 = 4$, on obtient la suite : $1 \to 4 \to 2 \to 1 \to 4 \to 2 \to 1 \to 4 \to 2 \to 1 \to \ldots$.

La conjecture de Syracuse (donc encore non démontrée ou infirmée) stipule que quelque soit le nombre u_0 choisi, on retombe au bout d'un certain nombre d'itérations sur le "cycle trivial" $1 \to 4 \to 2 \to 1 \ldots$

À noter que cette conjecture n'est toujours pas démontrée et qu'on ne sait pas si il existe d'autres cycles non triviaux. Néanmoins, aucun contre exemple n'a été trouvé à ce jour par calcul sur ordinateur (en 2020, pas de contre-exemple $< 2^{68}$) et les récentes avancées sur ce problème tendent à montrer que cette conjecture est vraie (mais ne le montre pas !).

On définit alors :

- **Le temps ou longueur de vol** : C'est le plus petit nombre d'itérations n tel que $u_n = 1$;
- **Le temps de vol en altitude** : C'est le plus petit indice n tel que $u_{n+1} \leq u_0$;
- **L'altitude à l'itération** n : C'est la valeur que prend la suite à l'itération n.

L'idée pour tester la conjecture est de choisir pour valeur de u_0 tous les valeurs entières comprises entre cinq et un entier maximal fixé (pour $u_0 = 3$, on peut voir qu'on obtient la séquence $3 \to 10 \to 5$ et donc qu'on retombe ensuite sur la même suite que pour $u_0 = 5$).

Il est facile de voir qu'il est inutile de parcourir les nombres pairs puisqu'on divise dès la première itération la valeur de u_0 par deux et donc qu'on retombe sur une suite déjà testée. De même, si on trouve un n tel que $u_n < u_0$ pour un certain u_0, alors on retombe sur une suite déjà testée.

Puisqu'on parcourt les entiers impairs, ils sont de la forme $4k+1$ ou $4k+3$. Or, si on considère les entiers impairs de la forme $4k+1$, on a pour séquence en prenant $u_0 = 4k+1$: $4k+1 \to 12k+4 \to 6k+2 \to 3k+1$. Or $3k+1 < 4k+1$ ($k > 0$), donc on est sûr de retomber sur une suite déjà testée pour les nombres impairs de la forme $4k+1$.

On ne va donc parcourir que les entiers de la forme $4k+3$ pour initialiser u_0.

3.1 Syracuse simple

Dans le premier programme, on ne s'intéresse qu'au temps de vol moyen des suites considérées ainsi qu'à la moyenne des altitudes maximales de chaque suite itérée. Pour chaque suite testée, on arrête l'itération dès qu'on trouve un u_n valant 1.

Paralléliser le programme `syracuse_simple.cpp` à l'aide d'OpenMP et calculer le speed-up pour divers nombre de threads. Commenter et expliquer les résultats que vous avez obtenus pour le speed-up.

3.2 Syracuse avec orbite

Dans le second programme, on veut conserver les valeurs prises par les suites dans un tableau (pour diverses raisons même si dans le programme proposé on ne s'intéresse qu'à la valeur maximale atteinte par toutes les suites).

Paralléliser le programme `syracuse_orbite.cpp` à l'aide d'OpenMP et calculer le speed-up pour divers nombre de threads. Commenter et expliquer les résultats obtenus en particulier en comparant ces résultats avec les résultats obtenus avec la version précédente de Syracuse.

4 Automate cellulaire

Le but de ce programme est d'étudier des automates cellulaires très simples en **une** dimension.

Un automate cellulaire consiste en une grille régulière de cellule (ici en une dimension) pouvant avoir plusieurs "états" (ici deux états : allumée = 1 ou éteinte = 0) qui seront modifiés selon l'état des cellules voisines (immédiates ou non) en suivant des règles fixes à chaque itération.

On appelle **degré de voisinage** le nombre de cellule que l'on doit visiter avant d'atteindre une cellule spécifique. Ainsi, pour une cellule donnée, ses cellules adjacentes sont de degré de voisinage égal à un.

On appelle **range** le degré maximal de voisinage des cellules prises en compte pour le calcul de l'état suivant.

Ainsi, l'automate le plus simple sera celui où on ne prendra que les cellules adjacentes (et elle-même) pour le calcul de l'état suivant (allumée ou éteinte). Il existe donc huit "motifs" différents (trois cellules donc 2^3 configurations possibles) pour lesquels il faut fixer une règle (à savoir si la cellule sera allumée ou éteinte à la prochaine itération) :

Motif	111	110	101	100	011	010	001	000

Il y aura donc dans ce cas là $2^8 = 256$ configurations (ou règles) possibles qu'on représentera par un nombre binaire variant de 0 à 255.

Par exemple, la règle numéro trente (en binaire 30 = 00011110) sera :

Motif initial (t)	111	110	101	100	011	010	001	000
Valeur suivante cellule centrale (t+1)	0	0	0	1	1	1	1	0

En prenant les cellules adjacentes et les cellules de degré de voisinage égal à deux, on aura donc cinq cellules prises en compte dans le calcul d'une cellule (la cellule centrale dont on veut modifier la valeur, deux cellules à gauche et deux cellules à droite), ce qui nous fera $2^5 = 32$ motifs possibles pour lesquelles il faut établir une règle. Le nombre de règles possible sera alors de $2^{32} = 4294967296$ règles possibles !

Dans notre cas, on veut étudier l'influence de ces règles en regardant l'évolution d'une grille de cellule dont au départ toutes sont éteintes sauf une cellule au centre de la grille qui elle sera allumée.

On rajoute de plus une "condition limite" consistant à conserver deux rangées de cellules à gauche et à droite qui resteront toujours éteintes (et influeront sur le résultat obtenu pour l'automate).

On parcourt ensuite toutes les règles possibles pour générer pour chacune un diagramme "espace-temps" où chaque ligne i du diagramme représente un état de la grille au temps $t + i.\Delta t$.

Toujours avec la règle n° 30, les 3 premières itérations d'évolution de l'automate sont :

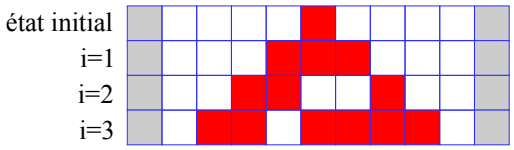

FIG. 1 : en blanc les cellules éteintes, en rouge les allumées, en gris les conditions aux limites (forcées à toujours éteintes).

Une fois que les diagrammes ont été calculés, on les transforme en image png qu'on sauvegarde.

Remarque : Si vous voulez gagner du temps à l'exécution ou bien essayer avec un degré de voisinage supérieur à un, vous pouvez modifier le code pour ne sauvegarder qu'une ou aucune image... Mais les images servent aussi à voir si vous n'avez pas introduit d'erreurs. À ce propos, pour vérifier que les fichiers images sont les mêmes qu'en séquentiel pour les valeurs de paramètres par défaut :

```
md5sum -c configs.md5sum
```

Enfin, pensez à conserver trois sources bien distincts :

1. Le premier fichier contenant uniquement la version OpenMP
2. Le deuxième fichier contenant uniquement la version MPI seule
3. Et si vous vous sentez d'attaque pour faire du MPI et de l'OpenMP, un troisième fichier contenant OpenMP et MPI (en bonus)

4.1 Parallélisation OpenMP

Paralléliser la boucle la plus externe à l'aide d'OpenMP. Calculer le speedup et commentez vos résultats.

4.2 Parallélisation MPI

Afin de pouvoir gérer des dimensions de grille et d'images plus importantes, on veut paralléliser le calcul des états de l'automate cellulaire ainsi que la constitution d'une image

(mais pas sa sauvegarde !).

Pour le calcul de l'état de l'automate cellulaire, on découpera la grille 1D en autant de blocs qu'il y aura de processeurs et en rajoutant une ou deux cellules (selon le degré de voisinage voulu) à gauche et à droite qui pourront servir soit de condition limite si les deux cellules correspondent à une condition limite de la grille originale, soit de zone d'échange de l'état des cellules à l'étape t (de la même manière que les cellules fantômes dans le projet).

En ce qui concerne l'image, il faudra simplement bien répartir l'image en plusieurs blocs correspondant aux cellules locales au processus.

Paralléliser le code selon cette méthodologie puis exécuter sur divers nombre de processus afin d'évaluer le speed-up pour le calcul de l'automate et la constitution de l'image, et conclure.

Bonus : Coder la parallélisation MPI *avec* la parallélisation OpenMP de la boucle externe, en utilisant au début du programme :

`int provided; MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);`

au lieu de `MPI_Init(&argc, &argv)`.