UNIVERSITÉ DE SHERBROOKE

25 avril 2019

MAITRISE INFORMATIQUE

Rapport du projet de recherche : Résolution de sudoku en parallèle [1]

Projet de session présenté à
Gabriel Girard
dans le cadre du cours
IFT 630 : Processus concurrents et parallélisme

Etudiant échange international

Julien DELAUNAY
18 155 167

1. Lien vers le git : `https://github.com/juliendelaunay35000/project_sudoku.git`

Table des matières

1 Introduction

Dans le cadre du cours IFT630 Processus concurrents et parallélisme, nous avons un projet de recherche en parallèle à réalisé. J'ai donc décidé de fabriquer un programme qui instancie des grilles de sudoku de manière aléatoire et qui cherche ensuite à remplir la grille. L'objectif de ce projet est de comparé le temps de résolution entre l'algorithme séquentiel et l'algorithme parallèle.

Pour la rédaction du rapport, j'ai utilisé LaTeX et ce projet est programmé en python avec la library *multiprocessing*[2] pour la mise en place du parallélisme. Ce projet se base dans la continuité du tp2 réalisé et la partie *brute force* en particulier qui à été programmé en python avec une library similaire.

2 Revue de la littérature

Le problème de la résolution de Sudoku est un défi connu à travers le monde de l'informatique. Certains chercheurs ont cherchés à classer par difficultés les grilles de Sudoku informatiquement [3]. D'autres, ont cherché à résoudre une grille de sudoku en se basant sur les algorithmes génétiques. [4] Le jeu du sudoku est devenu un phénomène mondial au cours de la dernière décennie. Il est également devenu un problème de test mathématique populaire afin de tester de nouvelles idées d'optimisation et de nouveaux algorithmes pour les problèmes combinatoires.

3 Mise en place sudoku

Pour la programmation du jeu de sudoku, ayant déjà codé pour un cours en France un jeu de mot fléchés en java, j'ai pu démarrer en ayant une idée de la réalisation de la grille. J'ai ensuite implémenté une technique de *backtracking search* pour résoudre la grille.

Pour le remplissage de la grille, j'ai utilisé la fonction *random.shuffle* afin de mélanger les valeurs et les points. Dans la fonction *populate()* (Annexe A), l'algorithme redémarre ensuite la grille à zéro et la remplis avec la fonction solve()(Annexe B.) Cette fonction est la partie du code qui est

2. `https://docs.python.org/2/library/multiprocessing.html`
3. `https://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS11/paper/viewPaper/2517`
4. `https://ieeexplore.ieee.org/abstract/document/7113148`

4 Utilisation de la library multiprocessing

Pour la mise en place du parallélisme dans ce programme, j'ai commencé par utiliser les fonctions *pool*[5] de cette library. Cependant, le problème de cette fonction est qu'il faut donner une liste à parcourir, j'ai donc implémentés les fonctions de la bibliothèques *threading*[6]. L'avantage de cette fonction est que le programme lance le nombre de threads qu'il désire, et il est ensuite possible que chaque thread lance une fonction différente.

Dans ce projet, le but de cette bibliothèque est de pouvoir tester avec des nombres différents de grilles à remplir ou un nombre différent de processus lancé si le temps d'éxecution avec le programme séquentiel est importante ou non.

5 Les différents résultats recherchés

Pour la comparaison des deux méthodes, j'ai lancé mon programme avec différents paramètres. Les principaux paramètres qui ont changés sont le nombre de processus lancés et donc le nombre de grilles remplis, ainsi que la taille de remplissage des grilles. Pour l'algorithme en parallèle, j'ai décidé de lancer chaque processus avec un nombre de grilles à remplir égal à 100.

Dans un premier temps, j'ai lancer le programme avec 500 grilles et le nombre par défaut de remplissage et 4 processeurs. J'ai ensuite lancé avec un nombre de remplissage de 20 ou 40 (au lieu de 36.) Pour les derniers tests, j'ai lancé le programme avec 16 processus, le nombre par défaut de remplissage et donc 1600 grilles à résoudre. Le dernier test effectué à été réalisé avec 16 processus et 10 grilles à remplir seulement. Ainsi il n'y a que 160 grilles à remplir.

6 Les résultats

Les résultats obtenus ont été écrit dans les fichiers .txt, et correspondent au temps mit par l'algorithme pour remplir le nombre prévu de grilles. Les documents commençant par *"time_parallel"*, correspondent aux temps de remplissage de l'algorithme en parallèle tandis que ceux commençant par *"time_sequentiel"* sont ceux correspondant à l'algorithme séquentiel.

Les fichiers n'ayant pas de suffixe sont les fichiers avec le résultat obtenu pour 4 fils et le nombre de remplissage de base. Les fichiers ayant 20 ou 40 pour suf-

5. `https://docs.python.org/fr/3/library/multiprocessing.html`
6. `https://docs.python.org/3/library/threading.html`

fixe sont ceux qui ont un nombre de remplissage de 20 ou 40 respectivement. Le fichier *"time_sequentiel_1600"* et *"time_parallel_16threads"* correspondent respectivement aux résultats pour les 16 threads avec le nombre de remplissage par défaut.

Nous pouvons voir que les résultats pour le nombre par défaut de remplissage est de 231 secondes pour le parallel et 267 pour le séquentiel.
La durée avec un remplissage de 20 pour le programme en parallèle est de 21366 secondes, ce qui correspond à 5,9 heures de calcul tandis que le programme en séquentiel à terminé son calcul au bout de 18688 secondes soit 5,2 heures.
Pour le programme avec un remplissage de 40, celui en parallèle à terminé en 104 secondes tandis que celui en séquentiel à terminé au bout de 117 secondes.
Le temps de calcul pour le programme avec 16 threads et un remplissage par défaut est de 1905 secondes soit 32 minutes tandis que celui en séquentiel à prit 4759 secondes soit 79 minutes.
Pour les derniers tests, le temps de calcul obtenu est de 309 secondes pour le séquentiel et 407 pour le programme en parallèle.

7 Conclusion

Au vu des résultats obtenus, je peux en tirer la conclusion que pour la résolution d'une grille Sudoku, si le nombre de processus est de 4, la durée du calcul n'est pas beaucoup plus rapide, ce qui est normal puisque le temps de lancer les processus et de faire le calcul ne récompense pas le temps de calcul pour le séquentiel.

Lorsque le nombre de remplissage est faible (20), le programme en parallèle à prit plus de temps à s'executer que le programme séquentiel. Je pense que ce problème est dû au fait que un processus à prit beaucoup de temps pour résoudre une grille, et les autres processus ont du l'attendre.

J'ai pu observer en modifiant le nombre de grilles à remplir avec les 16 processus qu'il est plus efficace de lancer le programme parallèle avec beaucoup de grilles à remplir (100) qu'avec un nombre faible (10).

L'utilisation d'un programme en parallèle pour la résolution d'une grille de Sudoku dépendent donc fortement du nombre de processus lancés et du nombre de grilles qui doivent être remplit.

A Fonction populate

Cette fonction permet de remplir la grille avec un nombre de chiffres à l'intérieur par défaut de 36.

```
def populate(self, n=36):
        """
        L'indice pour le nombre de valeurs a l'interieur de la grille est
        Vide le Sudoku, lance le solver en utilisant des valeurs aleatoire
        retire autant de valeur que possible.
        """
        # Mise en place du hasard dans la liste des points et des valeurs
        random.shuffle(self._points)
        random.shuffle(self._values)
        self.clear()
        self.solve()

        for point in self._points:
            if self[point] == 0:
                continue
            val = self[point]
            for v in self.candidates(point):
                if v == val:
                    continue
                self[point] = v
                if self.solve(True):
                    self[point] = val
                    break
            else:
                if (81 - sum(b._block.count(0) for b in self._field) < n):
                    self[point] = val
                    break
                self[point] = 0

        for point in self._points:
            self[point] *= -1
```

B Fonction solve

Cette fonction utilise la technique du backtracking search pour remplir la grille. En cas de blocage, le programme revient sur ses pas afin de chercher une nouvelle solution.

```
def solve(self, reset=False):
        """
        Resoud le Sudoku.
        Si 'reset' est True, regarde juste si le Sudoku est rempli.
        Retourne ensuite le sudoku identique par rapport a avant l'appel.
        """
        if self.is_solved():
            return True

        field = None
        for x in range(9):
            if field is not None:
                break
            for y in range(9):
                if self[x, y] == 0:
                    field = (x, y)
                    break

        if field is None:
            return True

        for new in self._values:
            try:
                self[field] = new
            except ValueError:
                continue
            else:
                if self.solve(reset):
                    if reset:
                        self[field] = 0
                    return True
                else:
                    self[field] = 0
```

C Utilisation de Git et LaTeX

J'ai choisis de mettre en place un github [7] afin de présenter mon travail. J'ai fais ce choix car je pense que Git est un outil de rendu de programme qui est très utile et qui permet une présentation soignée du travail réalisé. L'avantage de Git est la possibilité de retrouver le travail effectué et également de pouvoir le modifier.

J'ai également choisis d'écrire mon rapport en LaTeX car c'est un outil d'écriture qui est très utilisé dans la recherche et dont il faut s'habituer car la puissance et le soin qu'il donne à un rapport est très apprecié dans le monde professionnel. Ayant appris à utiliser LaTeX et GitHub récemment, j'ai pris ce projet comme un moyen de m'exercer dans ces deux outils.

7. `https://github.com/juliendelaunay35000/project_sudoku.git`