Introduction

Mission

Ce TER consiste en la distribution d'un calcul combinatoire sur plusieurs machines. Il s'agit de compter le nombre de semigroupes numériques par exploration d'arbre jusqu'à une profondeur donnée.

L'exploration consiste en un algorithme qui, à partir d'un semigroupe donné, génère tous les semigroupes fils induits par la suppression d'un nombre générateur.

Florent Hivert et Jean Fromentin ont proposé un algorithme optimisé pour résoudre ce problème et ont réussi à calculer le nombre de semigroupes jusqu'à la profondeur 67, ce qui a pris 18 jours.

Cela fait suite aux calculs effectués par Maria Bras-Amorós à l'aide d'un algorithme DFS performant mais non optimisé qu'on appellera par la suite "algorithme naïf".

L'algorithme optimisé est implémenté en `C++` et fait appel à de nombreuses techniques pour réduire le temps de calcul d'une solution engendrée à quelques dizaines de cycles processeur : vectorisation, parallélisation à l'aide de `Cilk++`, dérécursivation de l'exploration, optimisations des boucles et des accès mémoire ainsi qu'un choix rigoureux des types de données.

La principale limitation de cet algorithme est qu'il n'est pas possible de le distribuer facilement sur plusieurs machines. En effet, l'arbre d'exploration n'est pas équilibré et des problèmes de charge se posent rapidement (une seule machine fait tout le travail).

Une solution possible à ce déséquilibrage serait la mise en place d'un système de vol de tâche.

Le but de mon TER est donc de fournir une solution pour le calcul du nombre de semigroupes qui est performante, parallélisée et distribuée. L'objectif à terme sera de lancer le calcul final sur un cluster de machines performantes.

Problème posé

L'exploration de l'arbre d'un semigroupe numérique n'est pas triviale et est apparentée au problème de Frobenius.

Le mathématicien Georg Frobenius s'est posé la question suivante : quel est le montant maximal que l'on peut pas rendre en fonction de pièces de monnaie données ?

Ce montant se nomme "nombre de Frobenius" et est plus formellement décrit par le plus grand entier qu'il est impossible de calculer à partir de coefficients donnés.

On peut aussi visualiser ce problème en considérant les scores au Rugby : quels sont les scores que l'on ne peut pas obtenir avec le but (3 points), l'essai (5 points) et l'essai transformé (7 points) ?

La résolution des scores au Rugby est simple, on ne peut pas avoir 1, 2 ou 4 points ; tous les autres scores sont possibles. En effet, il est aisé de calculer le nombre de Frobenius (ici, 4) pour $n \leqslant 3$ où n est le nombre d'entiers possibles pour la combinaison. Ici, $n = 3$ pour $\{3, 5, 7\}$.

Organisation

Le travail sur ce TER a été effectué en autonomie avec un créneau d'une demi-journée par semaine au LRI pour faire le point sur les tâches à effectuer et résoudre les difficultés.

Le code s'organise autour d'un dépôt Git [1] qui contient tous les essais ainsi que la version finale destinée à être lancée sur un cluster de calcul.

Le dépôt contient plusieurs dossiers :

- `danse` : la version finale nommée DANSE pour *Distributed Array of Numerical Semigroup Explorations*, soit en français un réseau distribué d'exploration de semigroupes numériques ;
- `go-numeric-monoid` : les *bindings* Go de l'algorithme optimisé écrit en `C++` ;
- `naive-test` : l'implémentation de l'algorithme naïf, non parallélisé ;
- `native-parallelization` : un essai tout en Go qui implémente l'algorithme optimisé et qui est parallélisé, cependant il reste limité par certaines optimisations bas niveau impossibles à réaliser en Go.

Modélisation du problème

On veut calculer le nombre de semigroupes numériques (aussi appelé monoïdes) différents jusqu'à un genre maximum donné.

Un semigroupe numérique est composé de générateurs, il s'agit d'un ensemble minimal de nombres qu'on ne peut pas obtenir en combinant d'autres nombres appartenant au monoïde.

Ces combinaisons (au moins une addition) sont appelées décompositions, si pour un nombre du semigroupe il y a une et une seule décomposition alors ce nombre est un générateur.

Le genre d'un semigroupe numérique correspond à sa profondeur dans l'arbre et aussi au nombre de "trous". On appelle trou un générateur qui a été enlevé.

Le nombre de Frobenius d'un semigroupe est la valeur du plus grand trou.

Le conducteur est le nombre à partir duquel le nombre de décompositions est d'au moins 2.

Un exemple

Prenons un semigroupe quelconque noté S_X où X est le genre :

$$S_2 = \{0, \mathbf{3}, \mathbf{4}, \mathbf{5}, 6, 7, 8\} \cup [9; +\infty[$$

L'ensemble minimal de nombres générateurs a été mis en gras, on constate qu'on peut générer tous les autres nombres à partir de ceux-ci.

Le genre est égal à 2 car il y a deux trous, c'est à dire qu'on a retiré deux nombres au semigroupe, ici 1 et 2.

La multiplicité est de 1, soit le plus petit nombre appartenant au semigroupe avec 0 exclu.

Le nombre de Frobenius de ce semigroupe est égal à 5, soit le plus grand générateur.

1. `https://github.com/efournival/ter-lri`

Le conducteur est 6 (suivant le Frobenius), tous les nombres à partir de celui-ci peuvent être générés d'au moins deux manières :

— l'ajout de zéro au nombre, soit $0+6$ (ou $6+0$, mais on va ignorer la commutativité)

— la somme de deux nombres générateurs, ici $3+3$

Ce semigroupe va engendrer le semigroupe suivant si on retire, par exemple, 3 :

$$S_3 = \{0, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, 9, 10\} \cup [11; +\infty[$$

Algorithme naïf

Décrit par Maria Bras-Amorós en 20XX, cet algorithme *Breadth-First Search* permet de déterminer les générateurs d'un semigroupe numérique.

L'algorithme naïf utilise une structure de données comportant un tableau de $3 \cdot G - 1$ cases (avec G le genre maximum à calculer) de booléens indiquant pour chaque nombre si il possède ou non au moins une décomposition.

Étant donné un générateur à retirer, l'algorithme va calculer toutes les combinaisons possibles en effectuant un ET logique entre deux cases du tableau.

Algorithme optimisé

L'algorithme dit optimisé a été implémenté par Jean Fromentin et Florent Hivert avec l'objectif de calculer le monoïde engendré par la suppression d'un nombre générateur.

Il s'agit d'un algorithme de type *Depth-First Search* soit un parcours en profondeur, à la différence de l'algorithme naïf qui est un parcours en largeur.

La structure de données mise en place ici comporte :

— le genre du monoïde ;

— la multiplicité ;

— le nombre de Frobenius ;

— un tableau de $3 \cdot G - 1$ cases d'entiers représentant, pour chaque nombre appartenant au semigroupe, le nombre de combinaisons (additions) permettant de générer ce même nombre à partir des autres.

Le but de cet algorithme est de calculer le nombre de combinaisons.

Pour chaque nombre, si il possède au moins une combinaison, alors il est générateur du semigroupe.

Déroulement

Jeudi 12 janvier 2017

Installation au LRI, explications :

— parcours de l'arbre de semigroupes numériques

— structures de données utilisées, notamment dans `NumericMonoid`

— algorithme “naïf” : tableau de booléens (le nombre est-il dans le semigroupe ?) de taille $B = 3 \cdot G - 1$ où G est le genre soit la profondeur dans l'arbre et le nombre de trous

pour g de dernierEnlevé + 1 à B **faire**
  si $T[g]$ **alors**
    pour i de 1 à $\lfloor \frac{g}{2} \rfloor$ **faire**
      si $T[i]$ et $T[g-i]$ **alors**
        g est un nombre de décomposition

— algorithme optimisé facilitant la vectorisation : tableau d'entiers modélisant le nombre de paires ($i, j \in$ SN tel que $i + j = g$ et $i \leqslant j$), tous les nombres avec exactement une paire sont des générateurs

Autonomie

— première implémentation de l'algorithme naïf ; difficultés : initialisation, confusion décomposition/générateur

— première implémentation de l'algorithme optimisé

— organisation du dépôt Git

Lundi 16 janvier 2017

— mise au point et correction de l'algorithme naïf

— débogage et tests

Autonomie

— mise en place de tests unitaires

— amélioration de l'algorithme naïf ; difficulté : les tests unitaires sur des semigroupes connus passent, mais le résultat final (nombre de semigroupes calculés, par genre) n'est pas bon

— correction de l'algorithme optimisé : il fonctionne

— correction de l'algorithme naïf en rajoutant l'attribut `m`

— parallélisation à l'aide d'un verrou partagé sur la pile (`sync.RWMutex`) et d'une incrémentation atomique sur les cellules du tableau de résultat (packages `sync/atomic` et `unsafe`)

— difficultés : critère d'arrêt pour un genre donné, profiler le code car la charge processeur reste faible (verrous ?), types de données, stratégies de parallélisation

Jeudi 26 janvier 2017

— présentation du profiling avec la commande `perf`, elle est effective au niveau hardware donc ne dégrade pas les performances ; permet d'associer code machine et symboles de débogage au temps relatif passé par le processeur ; commandes `perf record`, `perf report` et `sudo perf top` pour du temps réel

— pour tester `NumericMonoid`, on ferra attention à se positionner dans le répertoire `src/Cilk++`

— il faut vérifier la bonne parallélisation en testant différents nombres de threads (modification de `runtime.GOMAXPROCS`) ; une bonne parallélisation donnera le résultat deux fois plus rapidement si on double le nombre de threads (jusqu'au nombre de cœurs, mais attention à l'HyperThreading : tester avec 2 fois le nombre de cœurs physiques) ; seul 5 à 10% du temps de calcul total est destiné à la communication/synchronisation

— pour remplacer l'utilisation de `sync/atomic`, on peut utiliser un stockage local au thread ; implémenté dans Sage en Python ; en Go, on pourra utiliser des *channels* (*pipes* en Python)

— pour s'arrêter lorsque les calculs sont parallélisés : compter le nombre de processus actifs (entre le pop de la pile et après les push) ; on a fini lorsque la pile est vide et qu'il n'y a plus aucun processus actif ; fait appel à `sync/atomic`

Jeudi 2 février 2017

— profiling en utilisant `perf` : mise en évidence de délais supplémentaires induits par les communications et la synchronisation

— mise en place d'un arrêt "propre" avec comptage du nombre de routines actives et de l'état de la pile

— les threads de l'ordonnanceur interne du Go (autant de threads que de cœurs logiques) ne prennent 100% de la charge processeur : mise en évidence (de trop) d'attente de verrous

Autonomie

— réflexion sur une stratégie de parallélisation qui ne nécessite pas de verrous et qui permettra à terme de mettre en place le calcul distribué

— rédaction du rapport

— difficultés sur la multiplicité, le conducteur, la méthode pour compter les combinaisons dans l'algorithme optimisé

Mercredi 8 février 2017

— éclaircissements sur les nombres de décomposition, les générateurs, le conducteur, la multiplicité, …

— discussion sur la stratégie à adopter pour la distribution : faire appel au code `C++` multithreadé à l'aide de `cgo` et gérer la distribution ainsi que le vol de tâche à l'aide d'un code Go natif

— code à étudier : l'itérateur dans `treewalk.cpp` de `NumericMonoid` ainsi que la méthode `walk_children_stack`

— pistes pour l'implémentation du vol de tâches : documentation automatique du code Python ainsi que le code source du projet Sage dans `src/sage/parallel/mapreduce.py`

Autonomie

— rédaction du rapport : algorithmes, exemple de semigroupe et avancement

— documentation et réflexion sur l'utilisation de `cgo`

— découverte de `go build -gcflags='-m'` qui indique si des fonctions ont été ou pas inlinées, et pour quelles raisons

— mise en place d'un package Go permettant de faire le lien entre le code optimisé de `NumericMonoid` et un code écrit en Go : implémentation d'une interface `C` ⇔ `C++` pour pouvoir utiliser `cgo`

— problèmes rencontrés : `NumericMonoid` fait une erreur de segmentation pour des valeurs de `MAX_GENUS` inférieure à 16 ; sous GCC 6, les résultats (nombre de semigroupes par genre) diffèrent (!), résolu en forçant l'utilisation de GCC 5

Résultats

Conclusion