Introduction

Mission

Ce TER consiste en l'exploration de l'arbre de semigroupes numériques dans le but de compter le nombre de combinaisons possibles jusqu'à une profondeur donnée.

L'exploration consiste en un algorithme DFS (*depth first search*) 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 $\{3, 5, 7\}$.

Déroulement

Jeudi 12 janvier 2017

Installation au LRI, explications :

— parcours de l'arbre de semi-groupes numériques

— structures de données utilisées, notamment dans `NumericMonoid`

— algorithme "naïf" : tableau de booléens (le nombre est-il dans le semi-groupe ?) de taille $B =$ 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 ($i, j \in \mathrm{SN}$ 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 en lecture/écriture 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

Résultats

Conclusion