## TER individuel

# Distribution d'un calcul combinatoire

**Edgar Fournival**

Encadrant : Florent Hivert

# Table des matières

# Introduction

Dans le cadre du deuxième semestre du M1 Informatique de Paris-Saclay se déroulant à l'Université Paris-Sud, une unité d'enseignement de TER est proposé aux étudiants.

L'acronyme TER signifie Travail d'Études et de Recherche, il peut être réalisé de manière individuelle au sein d'un laboratoire de Paris-Saclay ou en groupe à travers les différents sujets proposés en début de semestre.

Aucun des sujets proposés ne m'intéressait réellement. De plus, j'ai trouvé qu'un TER individuel était une bonne opportunité d'approfondir l'aspect recherche de ma formation.

J'ai donc contacté Florent Hivert, qui était responsable du cours d'algorithmique l'année dernière en L3 MIAGE apprentissage, afin de savoir si un encadrement individuel était possible. Sa réponse fût positive et nous nous sommes rencontrés afin de discuter de la mission et des modalités.

C'est comme cela que j'ai effectué mon TER au LRI au sein de l'équipe GALaC. Le sujet porte sur la distribution d'un calcul combinatoire, plus précisément l'exploration de l'arbre de semigroupes numériques.

J'ai ainsi été confronté à un problème combinatoire s'inscrivant tout à fait dans le programme du M1 Informatique, notamment en continuité de la matière Algorithmique Avancée.

Dans ce rapport, je vais tout d'abord présenter le contexte de ce TER puis dans un deuxième temps expliquer plus en détails le problème posé et la solution existantes. J'évoquerai enfin les développements effectués pour le distribuer et les résultats obtenus.

# Partie I

# Contexte

## I.1 Laboratoire et équipe de recherche

J'ai réalisé ce TER au Laboratoire de Recherche en Informatique (LRI) présent sur le campus d'Orsay. Il s'agit d'une d'une unité mixte de recherche (UMR 8623) menée conjointement par l'Université Paris-Sud et le CNRS.

Ma mission s'est déroulée dans le bâtiment 650 Ada Lovelace, qui héberge aussi des enseignants chercheurs de l'Inria.

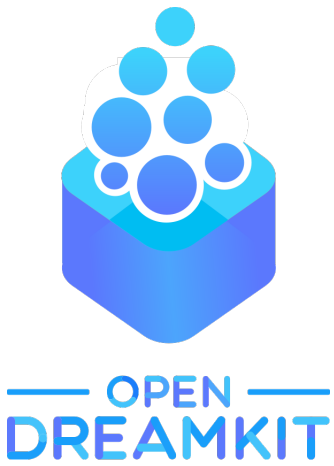


FIGURE 1 – Le logo d'OpenDreamKit

Plus précisément, j'ai été accueilli au sein de l'équipe GALaC (Graphes, Algorithmes et Combinatoire) à laquelle appartient mon enseignant encadrant. Dirigée par Johanne Cohen, cette équipe de recherche travaille sur des problématiques de calcul algébrique, d'algorithmique, de théorie des graphes

ou encore sur la spécification théorique de systèmes en réseau.

L'équipe GALaC coordonne et participe au projet européen OpenDreamKit [1] qui développe notamment le logiciel Sage. Il s'agit d'un ensemble de bibliothèques avec une interface en Python permettant de fournir des outils communs aux mathématiciens.

Cet effort de mutualisation des développements est commun à une dizaine d'universités européennes. Il permet de concentrer les efforts de 50 personnes réparties dans 16 sites sur la maintenance de librairies utilisées par Sage. Le budget total du projet est de 7.6 millions d'euros et une majeure partie sert à employer 11 chercheurs (en moyenne) à temps plein pour travailler uniquement sur OpenDreamKit.

## I.2 Mission

Mon travail a consisté à distribuer 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.

Cette mission s'inscrit dans le livrable 5.6 de la proposition OpenDreamKit cadrant notamment l'infrastructure et les besoins liés aux calculs efficaces en combinatoire. En effet, cela permet notamment d'approfondir et de développer des techniques d'exploration d'arbres fortement déséquilibrés.

L'algorithme mis en œuvre dans cet TER a été proposé et implémenté par Florent Hivert et Jean Fromentin, de l'Université du Littoral Côte d'Opale. Il est implémenté en `C++` et fait appel à de nombreuses techniques pour réduire le temps de calcul : vectorisation, parallélisation à l'aide de la bibliothèque `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. L'arbre d'exploration n'étant pas équilibré, des problèmes de charge se posent rapidement. En effet, au bout d'un certain temps, une seule machine voire un seul cœur fini par réaliser tout le travail. L'équilibrage entre plusieurs cœurs à été mis en place très simplement et de manière performante à l'aide de la librairie Intel Cilk Plus [2] qui travaille en mémoire partagée. Une solution possible à ce déséquilibrage serait la mise en place d'un système de vol de tâches, non plus limité à l'échelle du processeur, mais étendu à l'échelle d'une multitude de machines.

1. `http://opendreamkit.org/`
2. `https://www.cilkplus.org/`

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

## I.3 Organisation

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

Les horaires sont assez flexibles et je constate que la stratégie mise en place ici se fonde d'avantage sur le travail effectué que sur des heures fixes à respecter. Le télétravail est tout à fait intégré au fonctionnement de l'équipe. Il s'agit d'un mode de travail que j'ai déjà expérimenté lorsque j'étais apprenti et j'en étais très satisfait.

Tout au long de ce TER, j'ai travaillé de manière graduelle en testant et en expérimentant divers algorithmes ou techniques qui ont finalement permis de réaliser le programme final. Par exemple, j'ai commencé par implémenter dans le langage cible l'algorithme de calcul que nous souhaitions distribuer. Après nous être assuré des performances et de son fonctionnement, j'ai effectué des tests de concurrence avant de continuer.

J'ai noté mon avancement au fur et à mesure et un compte-rendu par période est visible en annexe I.

Au final, l'objectif durant ce TER était de réaliser un programme permettant de distribuer le calcul et de le lancer avant la fin de ma mission. Ayant eu beaucoup de travail le dernier mois de mon année universitaire, cela a pris du retard et j'espère pouvoir démarrer cette expérimentation avant la soutenance.

# Partie II

# Problème posé

## II.1 Modélisation

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 en effet 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 $\{3, 5, 7\}$.

Dans notre 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.

## II.2 Un exemple

Prenons un semigroupe numérique 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. Si on place ce semigroupe numérique dans un arbre, il aurait une profondeur de 2 également.

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.

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 monoïde va en engendrer un autre si on retire, par exemple, 3 :

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

Dans la suite du rapport, le semigroupe numérique présenté ci-dessus pourra être noté comme ceci :

$$\langle \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7} \rangle$$

## II.3 Premier algorithme

Décrit par Maria Bras-Amorós en 2005 [3], cet algorithme *Breadth-First Search* permet de déterminer les générateurs d'un semigroupe numérique.

Il 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.

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

FIGURE 2 – Approche naïve pour déterminer les nombres de décomposition d'un semigroupe numérique

L'approche de Maria Bras-Amorós se base donc sur l'algorithme donné en figure 2. Implémenté en Go dans le cadre de ce TER, il nécessite une modification car seuls les nombres générateurs nous intéressent. On va donc considérer uniquement les nombres partant du générateur que l'on vient d'enlever jusqu'à $3 \cdot G - 1$. Partant du principe qu'un nombre $n$ appartenant au semigroupe numérique est un générateur, il sera noté comme tel uniquement si l'algorithme donné ci-dessus ne détermine pas que $n$ est un nombre de décomposition.

## II.4 Algorithme optimisé

L'algorithme dit optimisé a été proposé et implémenté par Jean Fromentin et Florent Hivert [5] avec l'objectif de calculer le monoïde engendré par la suppression d'un nombre générateur. Il réalise cette tâche de manière très rapide en utilisant les fonctionnalités SIMD du processeur. Le parcours de l'arbre a été mis en place sous la forme d'un algorithme *Depth-First Search* soit un parcours en profondeur, à la différence de l'algorithme vu précédemment qui est un parcours en largeur. Il est donc bien plus optimal car il n'est plus nécessaire de stocker tous les monoïdes de genre $g$ avant de parcourir ceux de $g + 1$.

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.

Plus formellement, avec $S$ un semigroupe numérique quelconque, $S_g$ son genre et $i, j$ deux entiers appartenant à $\mathbb{N}$ :

$$i, j \in S \text{ si et seulement si } i + j = S_g \wedge i \leqslant j$$

## II.5 Mise en œuvre parallélisée

Nous avons vu dans la partie précédente qu'un algorithme optimisé avait été utilisé pour effectuer le calcul de manière efficace. Dans les faits, celui-ci repose sur différentes stratégies que nous allons modifier afin de pouvoir le distribuer sur plusieurs machines.

L'architecture existante est présentée en figure 3. Jusqu'à une certaine profondeur, on va effectuer un parcours de type DFS en comptant le nombre d'appels récursifs lancés, cela est réalisé par la fonction `walk_children()` du code original de `NumericMonoid`. Ensuite, à partir d'une profondeur définie par la constante `STACK_BOUND`, une méthode utilisant fortement des optimisations bas niveau et d'autres astuces de programmation comme un *Duff's device*[1] est impliquée. Celle-ci va compter le nombre de semigroupes numériques à l'aide d'itérateurs sur les nombres générateurs des monoïdes. Elle s'exécutera jusqu'à atteindre la profondeur `MAX_GENUS` qui est le genre maximal des semigroupes numériques à compter.

Cette dernière fonction prend quelques secondes à s'exécuter mais ne fait travailler qu'un seul cœur du processeur. Afin d'être le plus efficace possible, l'étage au-dessus va devoir la paralléliser, c'est à dire dans notre cas lancer ce code de manière concurrente dans l'objectif d'utiliser la totalité de la puissance de calcul de la machine.

Un problème se pose cependant : il est très difficile de paralléliser cette exploration car l'arbre est fortement déséquilibré. Afin de pouvoir calculer efficacement, la librairie Intel Cilk a été utilisée. Celle-ci repose sur le vol de tâches ainsi que sur des algorithmes de synchronisation très performants [2] fonctionnant en mémoire partagée.

Le vol de tâche décrit par Blumofe et Leiserson [1] est une technique

1. Un *Duff's device* est une manière de dérouler une boucle à l'aide d'une structure `switch` afin de s'affranchir des délais induits par le branchement conditionnel. Cela est nécessaire car GCC avec toutes les optimisations activées (`-O3`) ne déroule pas la boucle de la bonne façon. Plus d'infos sur `https://en.wikipedia.org/wiki/Duff's_device`

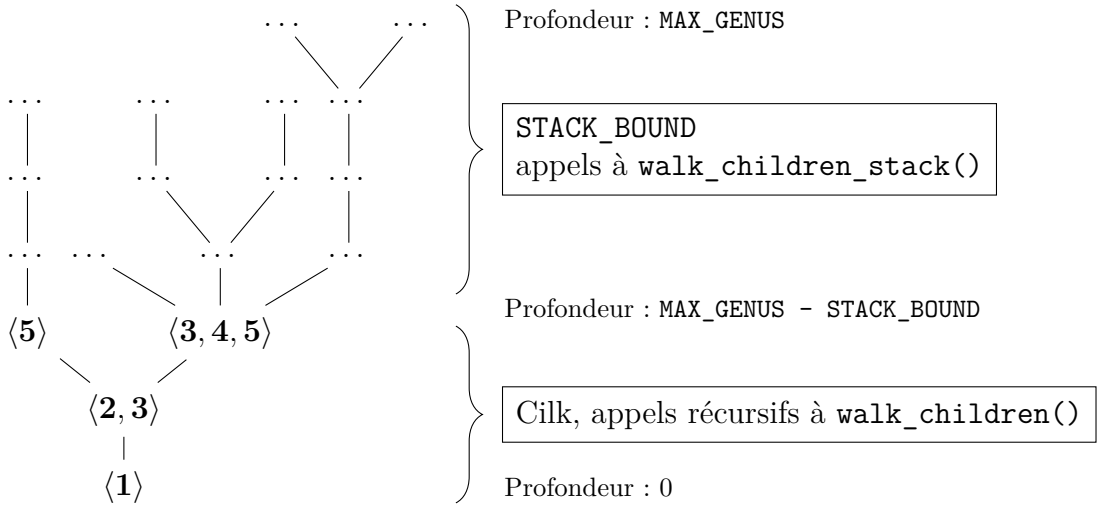


FIGURE 3 – Exploration parallélisée de l'arbre par `NumericMonoid`

permettant à un processus (dans notre cas, un *thread* géré par Cilk) de voler du travail à un autre lorsqu'il n'a plus rien à faire. Dans notre cas, elle permet d'exploiter 100% du processeur en ordonnançant automatiquement les tâches à réaliser. En effet, pour une profondeur $p$ quelconque de l'arbre, il a été démontré que seulement 1% des nœuds contenaient 99% des nœuds à la profondeur $p + 10$.

# Partie III

# Résolution et développements

## III.1 Technologies utilisées

Ce TER avait pour objectif initial d'implémenter l'exploration d'arbres de semigroupes numériques en Python afin d'utiliser les algorithmes de MapReduce et de vol de tâches mis en place dans Sage [4]. Cependant, j'ai proposé de le développer plutôt en Go, un langage qui me tient à cœur et qui est, à mon avis, plein d'avenir. De plus, il est parfaitement indiqué à notre problématique car :

— on peut appeler facilement du code C ;

— la partie réseau de la librairie standard est de très bonne qualité ;

— les fonctionnalités de parallélisation et d'asynchronisme du langage sont basées sur l'algèbre de processus *Communicating sequential processes* décrit par C. A. R. Hoare [6] et facilitent notamment l'implémentation du vol de tâches ;

— il est compilé et performant, le code machine généré est ainsi presque aussi rapide que celui généré par GCC.

Pour la transmission des informations, j'ai choisi d'utiliser le protocole TCP/IP qui, grâce à sa fiabilité et sa gestion de l'ordre des messages, est adapté aux calculs intensifs qui nécessitent une certaine sûreté. Cependant, il repose sur le compromis suivant : les communications ne comporteront pas d'erreurs mais sont susceptibles d'être incomplètes. Il faudra donc rajouter une couche supplémentaire à ce protocole afin de ne pas perdre de messages et donc fausser complètement le calcul.

Afin d'analyser les performances de mes programmes, je me suis appuyés sur les commandes issus du monde Linux suivantes : `perf` à laquelle j'ai

été initié durant ce TER, `time` et `top`. La plus utile et pertinente, `perf`, est particulièrement indiquée pour déterminer les parties du code qui ralentissent l'exécution d'un programme.

```
Samples: 44  of event 'cycles:u', Event count (approx.): 2366779
Overhead  Command          Shared Object          Symbol
  38,34%  simple-implemen  simple-implementation  [.] main.NaiveMonoid.Son
  18,93%  simple-implemen  simple-implementation  [.] runtime.mallocgc
  17,03%  simple-implemen  simple-implementation  [.] runtime.resolveTypeOff
   7,83%  simple-implemen  simple-implementation  [.] runtime.park_m
   5,27%  simple-implemen  simple-implementation  [.] runtime.runqget
   5,10%  simple-implemen  simple-implementation  [.] runtime.casp
   3,02%  simple-implemen  simple-implementation  [.] runtime.lock
   2,53%  simple-implemen  simple-implementation  [.] runtime.minitSignalMask
   0,80%  simple-implemen  simple-implementation  [.] runtime.clone
   0,60%  simple-implemen  simple-implementation  [.] runtime.sysmon
   0,51%  simple-implemen  [kernel.vmlinux]       [k] page_fault
   0,03%  simple-implemen  simple-implementation  [.] runtime.usleep
   0,01%  simple-implemen  simple-implementation  [.] _rt0_amd64_linux
```

FIGURE 4 – Informations présentées par l'écran principal de `perf`

Les statistiques de charge et le *monitoring* des systèmes de calculs seront effectuées grâce au projet libre `netdata`[1]. Cette application associe en effet des informations pertinentes (charge CPU, RAM, réseau, …) visualisable en temps réel à un impact sur la machine très faible. Elle sera utilisée pour contrôler les différentes données d'utilisation du système en fonction du temps et ainsi vérifier facilement le bon fonctionnement de la distribution.

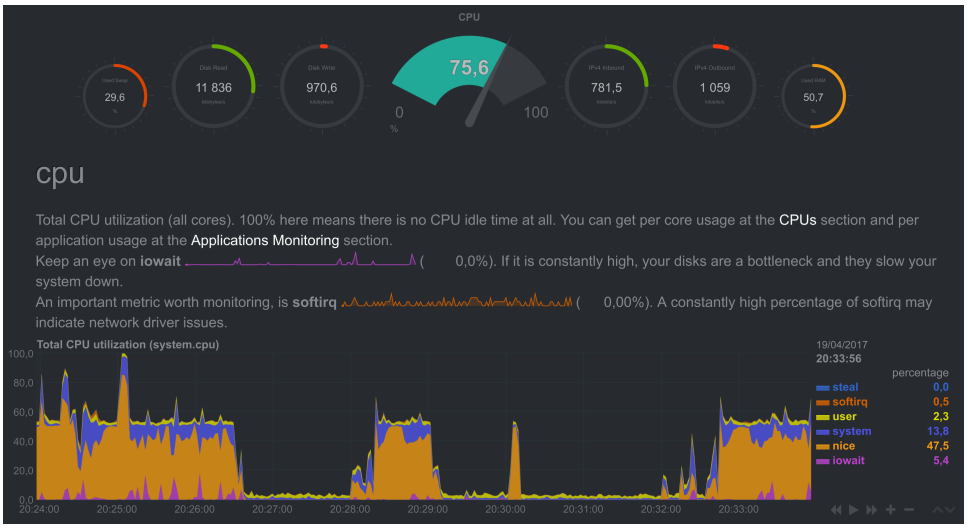


FIGURE 5 – Capture d'écran d'un panneau de contrôle de `netdata`

1. `http://my-netdata.io/`

## III.2 Idée générale

Pour résoudre et distribuer le problème, nous avons opté pour le rajout d'un "étage" dans l'architecture, lui aussi basé sur le vol de tâches.

Jusqu'à présent, Cilk effectuait un équilibrage local à la machine jusqu'à la profondeur définie par `STACK_BOUND`. Dans la solution mise en œuvre, le programme de distribution effectue un vol de tâches mais cette fois ci sur le réseau, jusqu'à la profondeur `CILK_BOUND` avant d'appeler `walk_children()`.

Cette dernière va ensuite équilibrer localement jusqu'à `STACK_BOUND` tandis que `walk_children_stack()` s'occupera des profondeurs supérieures, jusqu'à `MAX_GENUS`.

La mise en place de cette solution a donc nécessiter de pouvoir appeler les fonctions déjà implémentées en `C++`. Le langage Go n'étant pas prévu pour effectuer des optimisations bas niveau, j'ai pris la décision de réaliser des *bindings*[2] Go $\Leftrightarrow$ `C++`.

Ce développement a pris un certain temps dans le déroulement de ce TER car nous avons été confrontés à un bug de GCC qui sera abordé par la suite. De plus, j'ai fait le choix de mettre en place des tests unitaires[3] afin de ne pas me heurter à des incohérences de résultat par la suite. La mise en place de ces tests a également pris du temps.

L'intégration de la distribution par vol de tâches en Go a été plutôt simple à réaliser. En effet, le langage propose, notamment via les *channels*[4], différentes fonctionnalités très utiles à l'implémentation de notre système.

Je n'ai pas rencontré de difficultés non plus à implémenter la partie réseau, c'est à dire le code gérant les échanges de messages et les entités clients et serveurs. Le langage Go étant par design et grâce à sa librairie standard adapté à la mise en place de protocoles de transfert ou de microservices[5].

Cependant, la détection de la fin du calcul a été un challenge intéressant au niveau algorithmique. À l'heure où j'écris ce rapport, cela n'est pas encore au point et devrait être terminé avant de lancer les calculs définitifs.

2. Interface de programmation permettant d'appeler du code existant depuis un autre langage.

3. Tests automatisés portant sur une partie du code d'un programme et permettant notamment de vérifier, dans notre cas, que des résultats corrects sont bien retournés ou que des erreurs sont bien générées.

4. `https://gobyexample.com/select`

5. `https://fr.wikipedia.org/wiki/Microservices`

## III.3 Organisation

Le code s'articule autour d'un dépôt Git [6] 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++` ;

— `simple-implementation` : implémentation propre avec tests unitaires de l'algorithme naïf et de l'algorithme optimisé, non parallélisée ;

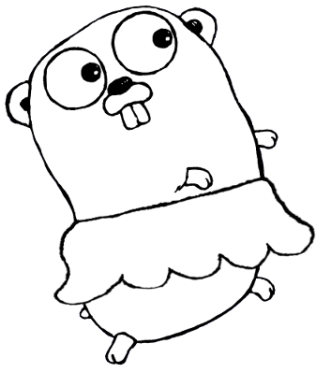

FIGURE 6 – Logo du programme de distribution `danse`, réalisé à partir du logo original du langage Go de Renee French

L'objectif est de lancer les calculs jusqu'au genre 70 sur des machines performantes. Ces résultats n'ont jamais été obtenus auparavant. Pour comparaison, le calcul jusqu'au genre 67 sur une machine à 32 cœurs AMD a pris 18 jours [5].

## III.4 Bug GCC Cilk n°80038

Certaines méthodes de l'implémentation originale réalisée par Florent Hivert et Jean Fromentin font appel à du code spécifique très optimisé que seul certains compilateurs, ici GCC pour le langage C++, peuvent générer. C'est pour cela que des *bindings* Go ont dû être implémentés afin de faire le lien

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

entre le code généré par GCC et les fonctionnalités plus haut niveau (notamment côté réseau) du langage Go.

Ces *bindings* ont été mis en place conjointement avec des tests unitaires. Ces derniers sont, en plus d'être fortement recommandés en général, omniprésents dans les développements réalisés en Go. En effet, le langage propose divers outils pour réaliser très simplement et rapidement ce genre de tests : la commande `go test`, le package `testing`, etc.

C'est donc dans ce cadre que je me suis rendu compte que les résultats (le comptage du nombre de semigroupes) renvoyés par les différentes explorations de l'arbre étaient incorrects, bien qu'aucune erreur ne soit remontée. Ces résultats incohérents sont également rencontrés avec le code original de `NumericMonoid`.

À la suite de nombreux tests, le problème a été identifié comme lié à la version du compilateur GCC. Ma machine de travail personnelle étant sous ArchLinux, je dispose de la dernière version stable (GCC 6) qui engendre ces résultats incorrects. Le développement de `NumericMonoid` ayant été réalisé à l'aide de GCC 5, les valeurs retournées étaient correctes jusque là.

Nous avons donc publié un rapport de bug avec un exemple réduit de code posant problème [7]. Le code minimal est visible en annexe II.

Il s'est avéré que GCC, depuis l'introduction d'un correctif concernant les arguments des lambda fonctions avec Cilk, appelait les destructeurs des arguments concernés au mauvais moment.

Mon encadrant, Florent Hivert, a rapidement trouvé une solution à ce problème une fois qu'il fut identifié : passer les semigroupes numériques par valeur et pas par référence. Cela implique une copie et grève les performances de l'ordre de 5%. Heureusement, le bug a été résolu par Xi Ruoyao et le correctif sera mis en place dans la prochaine version mineure de GCC.

7. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80038`

# Conclusion

À l'heure où j'écris ce rapport, nous n'avons aucune certitude concernant la fiabilité et l'efficacité du programme que j'ai développé. Celui-ci doit distribuer l'exploration de l'arbre des semigroupes numériques de manière correcte, c'est à dire en retournant le bon résultat, et performante afin de garder un bon facteur d'accélération. Tout cela sera vérifié le jeudi précédant la soutenance et nous devrions obtenir les résultats deux ou trois semaines plus tard.

Cependant, cela n'enlève rien au fait que ce TER a été très réussi en termes de compétences acquises et d'immersion dans le monde de la recherche. En effet, j'ai pu acquérir des connaissances dans des domaines variés comme l'optimisation, l'algorithmique, la parallélisation, etc.

Les différentes personnes que j'ai rencontré au LRI étaient toutes sympathiques et passionnée. J'ai apprécié cet aperçu de la partie recherche du métier d'enseignant-chercheur qui sévit à la faculté de sciences. Je remercie Florent Hivert pour avoir accepté de m'encadrer durant ce TER.

# Bibliographie


[1] Robert D. Blumofe et Charles E. Leiserson. « Scheduling Multithreaded Computations by Work Stealing ». In : *J. ACM* 46.5 (sept. 1999), p. 720–748 (cf. p. 9).

[2] Robert D. Blumofe et al. « Cilk : An Efficient Multithreaded Runtime System ». In : *SIGPLAN Not.* 30.8 (août 1995), p. 207–216 (cf. p. 9).

[3] M. Bras-Amorós. « Addition behavior of a numerical semigroup ». In : *Arithmetic, geometry and coding theory (AGCT 2003)*. T. 11. Sémin. Congr. Soc. Math. France, Paris, 2005, p. 21–28 (cf. p. 7).

[4] Florent Hivert. « HPC in Combinatorics : Application of Work-Stealing ». 2015 (cf. p. 11).

[5] Jean Fromentin et Florent Hivert. « Exploring the tree of numerical semigroups ». 14 pages. Mai 2013 (cf. p. 8, 14).

[6] C. A. R. Hoare. « Communicating Sequential Processes ». In : *Commun. ACM* 21.8 (août 1978), p. 666–677 (cf. p. 11).

# Annexes

## I 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` $\Leftrightarrow$ `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

## Jeudi 2 mars 2017

— il est "normal" que le programme fasse une erreur de segmentation pour une valeur de `MAX_GENUS` $\leqslant 11$ car il s'agit d'une constante codée en dur et utilisée pour indiquer quelles explorations doivent être explorées par Cilk ou par dérécursivation (ligne 48 de `treewalk.cpp`)

— un bug a sûrement été introduit par la version 6 de GCC au niveau de la vérification des indices d'accès aux tableaux : des avertissements apparaissent à la compilation lorsque l'indice est non signé (type `ind_t`), il n'y a plus d'avertissement si on change le `typedef` de `ind_t` en quelque chose de signé

— on veut vérifier que le Go supporte bien le modèle fork-join récursif, pour cela il faut implémenter l'exemple de Wikipédia (page anglaise) qui effectue un tri fusion

— la suite du travail consistera à traduire en Go la boucle principale du code source original de `NumericMonoid` soit `walk_children` ligne 49 de `treewalk.cpp`

## Autonomie

— nettoyage du dépôt et mise au propre des versions naïves et optimisées afin d'uniformiser les tests

— mise en place d'une VM et lancement de la compilation sous GCC 7 afin de déterminer si le problème de résultats faux était toujours présent : c'est le cas

— implémentation en Go de la boucle principale

## Jeudi 9 mars

— débogage du bug affectant les versions de GCC $\geqslant 6$, découverte d'un correctif qui consiste à passer le monoïde à explorer par valeur plutôt que par référence, isolation du bug

— suite de l'implémentation de la boucle principale : ajout de *bindings* supplémentaires

## Autonomie

— le bug [8] est remonté à l'équipe de GCC, à première vue il s'agit d'un mauvais appel des destructeurs dans la fonction parent ce qui conduit à un accès sur pointeur nul dans la fonction fille (lancée via `cilk_spawn`) [9]

— mise en place du *workflow* Go ainsi que du code haut niveau dans `danse`

— suite de l'implémentation de la boucle principale : cela prend du temps à cause de la mise en place des *bindings* Go, de la traduction du typage et des tests unitaires

— finalisation de l'intégration continue

## Jeudi 23 mars

— présentation [10] du problème de *race condition* dans Cilk

— explications sur les différents niveaux de l'exploration de l'arbre : Cilk, instruction `popcount`, etc.

— confirmation que l'ordonnanceur de Go est adapté à notre problème : pas d'explosion de la mémoire allouée ou de ralentissement quand un grand nombre de goroutines sont lancées

— évaluation des performances de l'ordonnanceur de Go et des channels : moins bien que Cilk et du C++ optimisé mais satisfaisant

— suite des opérations : distribuer le lancement des goroutines

## Autonomie

— réflexion sur la possible utilisation d'un algorithme de *machine learning* pour ajuster et optimiser les différents paramètres au fur et à mesure du calcul

8. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80038`
9. `https://gcc.gnu.org/ml/gcc-help/2017-03/msg00037.html`
10. `http://stackoverflow.com/questions/42816920/data-race-in-cilk-home-made-reducer`

— implémentation du système distribué

— *refactoring* du problème pour utiliser les fonctionnalités de synchronisation de Go : l'utilisation des *channels* afin d'obtenir un calcul complètement asynchrone donne de bonnes performances et permet de très facilement mettre en place le vol de tâche et la couche réseau

## Jeudi 6 avril

— pas d'intérêt particulier à utiliser un algorithme de machine learning, on insistera par contre sur le choix des paramètres initiaux

— attention à l'architecture basée entièrement sur des channels, les communications pour éviter les erreurs liées à la concurrence sont couteuses

— mise au point sur les différents étages de l'arbre, on va tester dans un premier temps une constante `CILK_BOUND`

— informations sur le rendu du rapport : vers mi Avril, bien introduire le problème, mettre un dessin des différents étages de parallélisation/distribution

## Autonomie

— dévellopement du programme de distribution final (DANSE)

— mise en place sur la machine du centre de calcul de l'université `gocilk`

## Jeudi 20 avril

— relecture et correction du rapport

— configuration de `gocilk` afin d'ouvrir les ports nécessaires à `netdata` et à `danse`

# II Code minimal causant le bug Cilk

```
#include <vector>
#include <cilk/cilk.h>

void walk(std::vector<int> v, unsigned size) {
  if (v.size() < size)
    for (int i=0; i<8; i++) {
      std::vector<int> vnew(v);
      vnew.push_back(i);
      cilk_spawn walk(vnew, size);
    }
}

int main(int argc, char **argv) {
  std::vector<int> v{};
  walk(v, 5);
}
```

À compiler à l'aide de :

```
g++-6 -std=c++11 -fcilkplus -g -Wall -lcilkrts red2.cpp -o red2
```