Projet de Systèmes Synchrones

Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates

Vincent Delaitre

vincent.delaitre@ens-lyon.fr

1 Introduction

Ce projet propose une implémentation d'une librairie pour la manipulation de très grands entiers. Ceux-ci sont décomposés récursivement sous la forme $n = g + x_p d$ avec $x_p = 2^{2^p}$, $0 \leq g < x_p$ et $0 < d < x_p$. Cela correspond en fait à une représentation sous forme de graphe dirigé acyclique et permet notamment de regrouper les noeuds de même valeur. Ainsi, contrairement aux représentations sous forme de tableau de bits, la taille de la structure nécessaire à stocker un entier n n'est pas une fonction croissante de n. Il existe d'ailleurs de rares nombres dit éparses dont la représentation sous forme d'Integer Decision Diagram IDD ne prend que très peu de place par rapport aux tableaux de bits.

J'ai implémenté les opérations présentées dans l'article et regroupé le tout dans une petite librairie C++. J'ai d'autre part ajouté quelques opérations que je décris dans la première partie. Je développe ensuite dans la deuxième section les difficultés que j'ai rencontrées et les solutions adoptées, ainsi qu'une petite démonstration d'un programme utilisant ma librairie. Enfin, je présente en dernière section mes résultats en comparaison avec une librairie Linux pour la manipulation d'entier.

2 Opérations sur les IDD

Dans cette partie, je m'attacherai à décrire les définitions récursives de mes propres opérations à partir de celles présentées dans l'article (je choisis les mêmes notations pour les opérateurs : `D` pour décrémenter, la fonction $\texttt{x'}(q) = x_q - 1$, etc...). Dans la suite, on aura toujours $n = g + x_p d$.

2.1 Complément à deux

Le complément à deux apparaît dans l'article pour calculer la soustraction. Je le défini de la manière suivante.

Soit $q > p$, on appelle complément à deux de n par rapport à x_q le nombre $x_q - n - 1$ et on le note $\overline{n}^q$. Remarquons que l'on a :

- Si $q = p + 1$: $\overline{n}^q = (x_p - g - 1) + x_p(x_p - d - 1) = \overline{g}^p + x_p\overline{d}^p$
- Sinon : $\overline{n}^q = x_{q-1} - n - 1 + x_{q-1}(x_{q-1} - 1) = \overline{n}^{q-1} + x_{q-1}\texttt{x'}(q-1)$

On en déduit donc l'algorithme suivant en $O(2^{n.p} + q)$ pour calculer $\texttt{NEG}(n, q)$ le complément à deux de n par rapport à x_q :

$$\begin{aligned}\texttt{NEG}(n,q) = (q = 0)\ ?\ (n = 0\ ?\ 1 : 0)\ :&\\ (n.p = q')\ ?\ \texttt{T}(\texttt{NEG}(n.0, n.p), n.p, \texttt{NEG}(n.1, n.p))\ :&\\ \texttt{T}(\texttt{NEG}(n, q'), q', \texttt{x'}(q'))&\\ \{q' \leftarrow \texttt{D}(q)\}&\end{aligned}$$

2.2 Exponentielle

L'exponentielle $\texttt{POW}(n,m) = n^m$ a été implémentée par l'algorithme d'exponentiation rapide. Il utilise l'opérateur $\texttt{HALF}(n) = (q,r)$ tels que $n = 2q + r$ avec $r < 2$ (division euclidienne par 2). En négligeant la complexité de `HALF` par rapport à celle de l'opérateur produit `P`, on trouve une complexité en $O(m2^{n.p})$.

$$\begin{aligned}\texttt{POW}(n,m) = (m=0)\ ?\ 1\ :&\\ (m=1)\ ?\ n\ :&\\ (r=0)\ ?\ \texttt{P}(x,x)\ :&\\ \texttt{P}(x,\texttt{P}(x,x))&\\ \{(q,r) \leftarrow \texttt{HALF}(m)\ , x \leftarrow \texttt{POW}(n,q)\}&\end{aligned}$$

L'opérateur division euclidienne par deux est calculé comme suit. Il utilise l'opérateur $\texttt{Y}(q) = x_q/2$ qui se définit facilement par récurrence : $\texttt{Y}(0) = 1$ et $\texttt{Y}(q) = \texttt{T}(0, q-1, \texttt{Y}(q-1))$. `Y` est déclaré en tant que fonction de mémorisation locale.

$$\begin{aligned}\texttt{HALF}(n) = (n=0)\ ?\ (0,\ 0)\ :&\\ (n=1)\ ?\ (0,\ 1)\ :&\\ (r_1=0)\ ?\ (\texttt{T}(q_0, n.p, q_1),\ r_0)\ :&\\ (\texttt{C}(\texttt{A}(q_0, \texttt{Y}(n.p)), n.p, q_1),\ r_0)&\\ \{(q_0,r_0) \leftarrow \texttt{HALF}(n.0)\ , (q_1,r_1) \leftarrow \texttt{HALF}(n.1)\}&\end{aligned}$$

2.3 Décalage à gauche

On souhaite calculer $n \ll m$. Posons $(e,l) = \texttt{R}_\texttt{M}(m)$ avec $e < 2^l$, on a alors :
$n \ll m = x_l 2^e n = 2^m g + x_p 2^m d$. On distingue alors deux cas selon que $p < l$ ou $p \geq l$ et on justifie l'utilisation du constructeur `C` :

- Si $p < l$ alors $n < x_{p+1} \leq x_l$ d'où $2^e n < x_l^2 = x_{l+1}$. On peut ainsi utiliser l'opérateur `C` pour calculer $n \ll m = \texttt{C}(0, l, n \ll e)$.
- Sinon $l \leq p$ donc $2^m = 2^e x_l < x_{l+1} \leq x_p$ et on peut utiliser `C` pour calculer $n \ll m = \texttt{C}(g \ll m, p, d \ll m)$.

On en déduit donc l'algorithme suivant en $O(\log_2(m) + 2^{n.p})$ pour calculer $\texttt{LSH}(n,m) = n \ll m$:

$$\begin{aligned}\texttt{LSH}(n,m) = (m=0)\ ?\ n\ :&\\ (n=0)\ ?\ 0\ :&\\ (n=1)\ ?\ \texttt{A}_\texttt{M}(0,n)\ :&\\ (l > n.p)\ ?\ \texttt{C}(0, l, \texttt{LSH}(n,e))\ :&\\ \texttt{C}(\texttt{LSH}(n.0, m), n.p, \texttt{LSH}(n.1, m))&\\ \{(e,l) \leftarrow \texttt{R}_\texttt{M}(m)\}&\end{aligned}$$

2.4 Décalage à droite

L'algorithme de décalage à droite est un peu différent et utilise la soustraction. En effet :

$$n \gg m = \lfloor 2^{-m} g + 2^{2^p - m} d \rfloor = \begin{cases} g \gg m + d \ll (2^p - m) & \text{si } 2^p \geq m \\ d \gg (m - 2^p) & \text{sinon} \end{cases}$$

On calcule donc le décalage à droite $\texttt{RSH}(n,m) = n \gg m$ en $O(2^{n.p+m.p})$ de la manière suivante (`A` et `S` sont respectivement les opérateurs d'addition et de soustraction) :

$$\begin{aligned}\texttt{RSH}(n,m) = (m=0)\ ?\ n\ :&\\ (n=0)\ ?\ 0\ :&\\ (n=1)\ ?\ 0\ :&\\ (k \geq m)\ ?\ \texttt{A}(\texttt{RSH}(g,m), \texttt{LSH}(d, \texttt{S}(k,m)))\ :&\\ \texttt{RSH}(d, \texttt{S}(m,k))&\\ \{k \leftarrow \texttt{A}_\texttt{M}(0, n.p)\}&\end{aligned}$$

3 Implémentation

J'ai implémenté une mini-librairie de manipulation de grands entiers en C++ à partir de mes opérations ainsi que celles décrites dans l'article. Je détaille ci-après mes choix d'implémentation pour les fonctions mémo ainsi que pour effectuer l'optimisation qui utilise les mots machines plutôt que les définitions récursives lorsque c'est possible. Je souligne au passage que les définitions de `C` et `P` dans l'article m'auront posé quelques difficultés puisqu'elles ne sont pas tout à fait correctes.

En effet, pour `C`, il faut commencer par le test $p = d.p$ et non $p = g.p$, sans quoi il arrive parfois que la condition $max(g, d) < x_{p+1}$ soit violée. Par exemple, en suivant la définition de l'article, le calcul de $\texttt{C}(3, 0, 3) = T(1, 0, 4)$ mène à un résultat corrompu.

Pour `P`, il arrive que les produits récursifs $\texttt{P}(a, b.0)$ et $\texttt{P}(a, b.1)$ violent encore une fois la condition $max(g, d) < x_{p+1}$ pour le constructeur `C` ce qui mène à nouveau à un résultat corrompu. Il faut donc rajouter des cas pour traiter ces débordements.

Enfin, la gestion de la mémoire aura aussi soulevé quelques questions puisque je ne dispose pas de garbage collector.

3.1 Fonctions-mémo

Pour éviter de calculer deux fois la même valeur, certains opérateurs mémorisent les anciens calculs en mémoire et retournent directement le résultat si une opération déjà rencontrée survient à nouveau. C'est notamment le cas de `T` et `x'` qui se comportent comme des fonctions-mémo globales partagées entre tous les entiers. Les opérateurs `TWICE`, `HALF`, $\texttt{A}_M$, `P` et `Y` sont quant à eux des fonctions-mémo locales qui sont effacées à la fin des différents appels récursifs engendrés par une opération entre deux entiers.

Pour réaliser les fonctions mémo, les opérateurs disposent d'une table de hachage ce qui permet d'effectuer les recherches en temps constant. J'ai utilisé ma propre implémentation des tables de hachage : les buckets sont des arbres rouges et noirs, la taille de la table est multipliée par deux (et amenée au plus proche nombre premier) dès que la table a autant d'éléments que de buckets. L'utilisation d'arbres rouges et noirs au lieu de listes était peut-être un peu abusive puisque ma fonction de hachage est assez bonne et que les buckets contiennent la plupart du temps 0, 1 ou 2 éléments pour une table de 2^{16} buckets remplie.

3.2 Utilisation des mots-machine

Les opérateurs sont définis de manière récursive mais dès que les entiers manipulés tiennent sur un mot machine on utilise plutôt le processeur. Afin de conserver une certaine homogénéité dans les structures de donnée, les mots-machine sont représentés comme les autres noeuds des IDD à l'aide de trois pointeurs $n.0$, $n.p$ et $n.1$ vers les fils. Dans le cas d'un mot machine W, $n.p$ est $NULL$ et $n.0/n.1$ sont utilisés pour stocker le mot 32 ou 64 bits si bien que $\texttt{T}(W\&\texttt{0xffffff}, NULL, W \gg 32)$ revoie l'adresse du noeud correspondant au mot machine W. On conserve de cette manière la propriété fondamentale pour l'égalité : deux entiers sont égaux s'ils sont stockés à la même adresse.

3.3 Gestion de la mémoire

Deux problèmes se présentent en ce qui concerne la gestion de la mémoire.

D'une part, les tables temporaires deviennent parfois trop grosses lorsque les opérandes possèdent un grand nombre de bits. Pour résoudre ce problème, un nettoyage de la table est effectué avant le redimensionnement (qui se produit lorsque la table est pleine) : on supprime les noeuds orphelins. Si le nombre de noeuds stockés après nettoyage est inférieur à la moitié de la taille de la table, le redimensionnement est annulé.

Pour réaliser le nettoyage, on tient à jour pour chaque noeud un compteur du nombre de parents. Lors du nettoyage, les noeuds sans parents sont supprimés. Il faut cependant veiller tout particulièrement à déclarer les entiers utilisés par les calculs en cours comme vivant, même s'ils sont au sommet de l'IDD et donc sans parent. Pour cela, j'ai utilisé une classe C++ qui encapsule les entiers de base et qui tient à jour les compteurs.

3.4 Aperçu de la librairie

La librairie se trouve dans les fichiers *iddlib.cpp* et *iddlib.h*. Il y a deux types principaux : `uIDD` et `sIDD` pour représenter les entiers signés et non signés. Tout a été fait pour faciliter la lecture et les entiers de la librairie se manipulent comme des entiers normaux. Il y a de plus quelques fonctions supplémentaires :

- `(u/s)IDD idd_pow((u/s)IDD n, uIDD m)` élève n à la puissance m.
- `(unsigned/signed) int idd_to_int((u/s)IDD n)` convertit vers le type int. En cas de dépassement, renvoie MAX_INT.
- `void idd_print((u/s)IDD n)` affiche n sur stdout.

Ci-dessous un petit exemple pour calculer factorielle 1000, le $10000^{\text{ème}}$ terme de la suite de Fibonacci ainsi que Ackermann(4,2) (cf le fichier example.cpp) :

```
#include<stdio.h>
#include"iddlib.h"

uIDD ackermann(uIDD m, uIDD n) ;

int main(int, char**)
{
uIDD fact = 1 ;
for(int i=2 ;i<=1000 ;i++)
fact = i*fact ;

printf("Factorielle 1000 :\n") ; idd_print(fact) ; printf("\n\n") ;

uIDD fibo1 = 0 ;
uIDD fibo2 = 1 ;
for(int i=2 ;i<=10000 ;i++)
{
uIDD tmp(fibo2) ;
fibo2 = fibo1+fibo2 ;
fibo1 = tmp ;
}

printf("Fibonacci 10000 :\n") ; idd_print(fibo2) ; printf("\n\n") ;

printf("Ackermann(4,2) :\n") ; idd_print(ackermann(4,2)) ; printf("\n") ;

return 0 ;
}

uIDD ackermann(uIDD m, uIDD n)
{
if(m == 0) return n+1 ;
else if(m == 1) return n+2 ;
else if(m == 2) return (2*n)+3 ;
else if(m == 3) return (1<<(n+3))-3 ;
else if(n == 0) return ackermann(m-1, 1) ;
else return ackermann(m-1, ackermann(m, n-1)) ;
}
```

4 Résultats

J'ai comparé ma librairie à la GNU MP Bignum Library (GMP) au niveau temps de calcul et mémoire consommée pour les opérations d'addition, de multiplication et d'élévation à la puissance

100. La GMP est entre 10 et 10000 fois meilleure pour le temps de calcul. L'espace mémoire utilisé par ma librairie est quant à lui sensiblement équivalent à celui de la GMP pour l'addition et les petits nombres, mais croît dramatiquement pour l'exponentielle. Les résultats sont rassemblés dans les tableaux et graphiques ci-dessous. Le fait que l'on obtienne à peu près des droites sur les graphes montre que l'on obtient expérimentalement une complexité polynomiale. Par exemple, pour la multiplication IDD, le coefficient de la droite est 2.3 d'où une complexité de $O(n^{2.3})$.

Tab. 1. Nombre d'additions par seconde et occupation mémoire pour différentes tailles d'entiers.

Bits	32	128	256	512	1024	8192	65536
#/s Add IDD	$13 \cdot 10^6$	$169 \cdot 10^3$	$73 \cdot 10^3$	$24 \cdot 10^3$	$17 \cdot 10^3$	10^3	150
#/s Add GMP	$28 \cdot 10^6$	$18 \cdot 10^6$	$23 \cdot 10^6$	$20 \cdot 10^6$	$14 \cdot 10^6$	$3 \cdot 10^6$	$377 \cdot 10^3$
#/s Ratio	2,13	108	319	841	815	1577	2517
Mem IDD (Ko)	124	124	128	136	140	224	160
Mem GMP (Ko)	140	128	128	128	128	140	128

Tab. 2. Nombre de multiplications par seconde et occupation mémoire pour différentes tailles d'entiers.

Bits	32	128	256	512	1024	8192	65536
#/s Mult IDD	$142 \cdot 10^3$	$13 \cdot 10^3$	$3 \cdot 10^3$	697	155	0,73	0,01
#/s Mult GMP	$40 \cdot 10^6$	$11 \cdot 10^6$	$5 \cdot 10^6$	10^6	$445 \cdot 10^3$	$15 \cdot 10^3$	571
#/s Ratio	282	824	1633	2122	2871	$20 \cdot 10^3$	$68 \cdot 10^3$
Mem IDD (Ko)	124	140	192	412	1232	6196	10888
Mem GMP (Ko)	140	132	132	132	140	160	228

Tab. 3. Nombre d'élévation à la puissance 100 par seconde et occupation mémoire pour différentes tailles d'entiers.

Bits	32	128	256	512	1024	8192	65536
#/s Pow100 IDD	85	1,65	0,32				
#/s Pow100 GMP	224768	9140	6660	2270	779	49	4
#/s Ratio	2644	$11 \cdot 10^3$	$20 \cdot 10^3$				
Mem IDD (Ko)	2036	9448	37700				
Mem GMP (Ko)	168	152	128	176	200	560	1152

5 Conclusion

Les tests précédents semblent donner un avantage aux tableaux de bits, d'autant plus qu'il s'avère que les grands nombres testés ne sont pas très denses : les IDD devraient en théorie être plus efficaces.

Il y a cependant de nombreux facteurs qui peuvent dégrader les performances. En effet, bien que mes tables de hachage soient honorablement efficaces, je ne doute pas que l'on puisse faire bien mieux. Le nettoyage trop fréquent des tables peut aussi ralentir l'exécution en provoquant du recalcul. Enfin, la grande quantité de mémoire utilisée pour les grands nombres empêche les tables de hachage de tenir dans le cache processeur.

Il y a encore beaucoup de travail pour arriver à une librairie réelle mais pour conclure, il semble que le résultat soit en fait plutôt prometteur pour un premier jet d'une librairie, comparée à la GMP développée depuis longtemps et optimisée en assembleur.

Fig. 1. Nombre d'opérations par secondes et ratio pour l'addition (échelles logarithmiques)

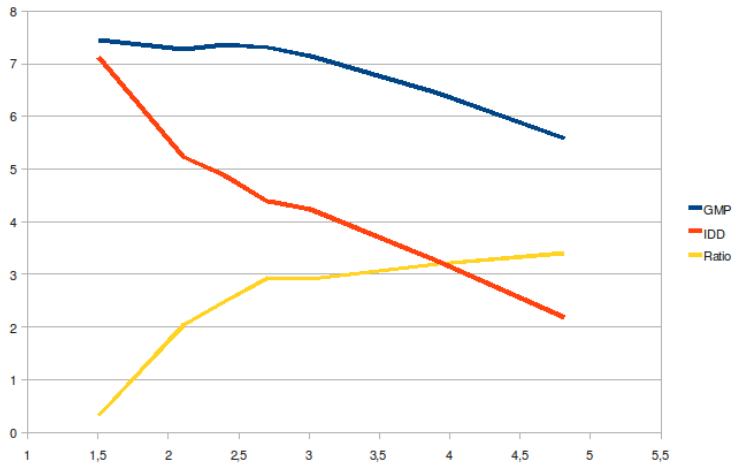

Fig. 2. Nombre d'opérations par secondes et ratio pour la multiplication (échelles logarithmiques)

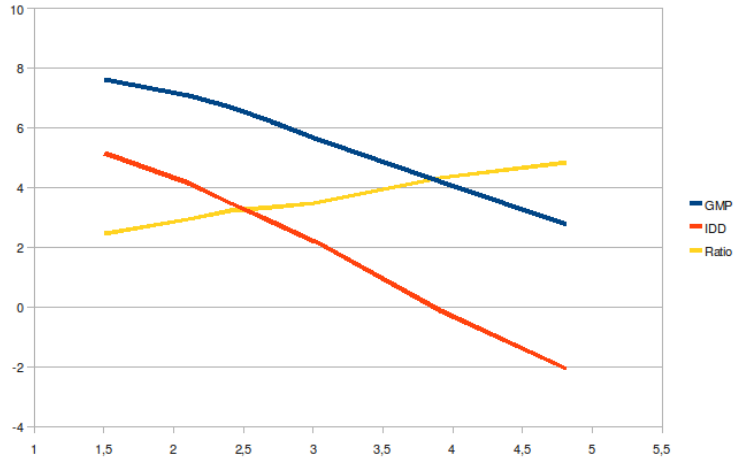

Fig. 3. Nombre d'opérations par secondes et ratio pour la puissance 100 (échelles logarithmiques)