L'AUTEUR

Jean-Paul DELAHAYE
est professeur à l'Université de Lille et chercheur au Laboratoire d'informatique fondamentale de Lille (LIFL).

qui peut être un ensemble de nombres, ou de lettres ou de codes numériques chargés de représenter informatiquement des objets réels. On se donne aussi un nombre entier n et une application F de S^n dans S^n, c'est-à-dire une transformation qui à tout n-uplet d'éléments de S en associe un autre. Dans l'exemple de la permutation circulaire, l'application F de S^3 dans S^3 est celle qui, à tout triplet de valeurs (A, B, C), associe (B, C, A).

Calcul *in situ* général

Quand on considère des variables booléennes, cela signifie que chaque élément de S est 0 ou 1. En plus du cas booléen, nous envisagerons deux autres cas finis. Si nous calculons avec des nombres entiers écrits en binaire avec k chiffres alors les éléments de S sont 0, 1, 2, ..., $2^k - 1$; si nous calculons modulo p (nous remplaçons chaque nombre par son reste lors de la division par p), les éléments de S sont 0, 1, ..., $p - 1$.

Dans le cas général, un calcul *in situ* pour une application F est une suite d'affectations analogues à celles que nous avons utilisées dans les exemples. Ces affectations sont envisagées une à une et chacune attribue une valeur nouvelle à l'une des variables dont l'ancienne valeur est donc perdue... c'est tout le problème ! La valeur nouvelle ne dépend que des valeurs disponibles à cet instant du calcul et nous nous imposons, dans le calcul *in situ*, de n'utiliser aucune variable supplémentaire.

Pour la permutation circulaire de n variables proposée dans l'encadré 1, nous considérons l'opération :

$F : (A_1, A_2, \ldots, A_n) \rightarrow (A_2, \ldots, A_n, A_1)$.

La méthode proposée montre que $n + 1$ affectations suffisent pour réaliser cette permutation circulaire, sans que nous n'utilisions aucune autre variable que celles données, et sans que nous ayons à stocker dans les variables des nombres plus grands que $2^k - 1$ dans le cas des nombres en binaire, ou que $p - 1$ dans le cas du calcul modulo p. L'encadré 2 explique que lorsqu'on se donne des variables où l'on peut mémoriser des entiers sans limitation de taille (ce qui est irréaliste en informatique), alors toute affectation F de S^n dans S^n se calcule *in situ* en au plus $n + 1$ affectations.

Cette valeur de $n + 1$, valable pour les permutations circulaires de n variables et pour les transformations F quelconques quand nous utilisons des variables non bornées, n'est pas générale. Pour la transformation $F(A_1, A_2, A_3, A_4) = (A_2, A_1, A_4, A_3)$, on ne sait pas faire mieux que six affectations pour un calcul *in situ* avec des variables bornées.

Les programmes *in situ*, surtout s'ils sont courts, devraient aider à concevoir des puces électroniques moins gourmandes en mémoire, ce qui augmentera la rapidité du calcul. En effet, calculer «localement» est souvent plus efficace pour un processeur que de faire des requêtes aux mémoires externes qu'on doit associer à l'utilisation de variables auxiliaires.

Cette économie de mémoire entraîne un allongement de la taille des programmes. C'est là un nouvel exemple d'une règle générale rencontrée en maintes occasions en informatique : pour économiser de la mémoire, il faut parfois complexifier le calcul. Les algorithmes de compression

2. Calcul *in situ* dans le cas infini

Au départ, nous avons un n-uplet : $(A_1, A_2, ..., A_n)$. Les variables $A_1, A_2, ..., A_n$, sont choisies dans un ensemble S non borné, par exemple l'ensemble des entiers.

Une application F attribue de nouvelles valeurs aux variables $A_1, A_2, ..., A_n$ qui deviennent $B_1, B_2, ..., B_n$.

Utilisons des variables où nous pouvons mémoriser des entiers sans limitation de taille et voyons comment il a été démontré, dans ce cas particulier, que toute application F peut se calculer *in situ* en au plus $n + 1$ affectations. Pour que le calcul soit *in situ*, nous devons tenir compte du fait que lorsque nous mettrons une valeur dans A_i, celle-ci écrasera l'ancienne valeur dès lors indisponible. Pas question donc d'écrire un programme du type :

$A_1 := E_1(A_1, A_2, ..., A_n), A_2 := E_2(A_1, A_2, ..., A_n), ..., A_n := E_n(A_1, A_2, ..., A_n)$.

L'idée du calcul est de remplacer A_1 par $A' = 2^{A_1} 3^{A_2} ... p_n^{A_n}$, où la suite p_i désigne la suite des nombres premiers (2, 3, 5, 7, ...). Cette étape réunit dans A_1 toute l'information contenue dans les variables $A_1, A_2, ..., A_n$. Pour récupérer A_i à partir de A', on calculera le nombre de fois que A' est divisible par p_i. Le fait de placer B_2 dans la variable A_2 ne fait pas perdre l'information A_2, car celle-ci reste présente, mêlée aux autres valeurs dans A'. On peut alors calculer tous les nouveaux A_i en récupérant toutes les valeurs des anciens A_i dans A', puis en utilisant l'application adéquate. À l'étape $n + 1$, on calcule A_1 en fonction de A'.

Comme nous avons regroupé et codé (avec la décomposition en facteurs premiers) toutes les valeurs de $A_1, A_2, ..., A_n$ dans A', nous n'avons pas besoin de mémoire supplémentaire et nous gardons à portée de main durant le calcul toutes les valeurs initiales des variables mises «au chaud». Bien sûr, cette astuce ne fonctionne pas en informatique où chaque variable ne peut prendre qu'un nombre fini de valeurs.

de données fonctionnent aussi selon cette maxime : en calculant les versions compressées d'images, de musiques ou de vidéos que nous voulons conserver, nous économisons de l'espace. Aujourd'hui, mener des calculs est devenu plus facile et l'algorithmique *in situ* est appelée à se développer et à intervenir au plus profond des puces électroniques. Un brevet a été déposé dans cette optique par les chercheurs du domaine.

Serge Burckel, aujourd'hui enseignant à l'Université de La Réunion, est l'initiateur des travaux de recherche sur l'algorithmique *in situ* et l'un des principaux auteurs des découvertes sur le sujet. Il pense que le principe du calcul *in situ* a également un sens physique.

L'information est conservée

Pour S. Burckel, l'importance des calculs *in situ* provient de ce que tout système physique isolé qui se modifie mène une sorte de calcul *in situ*. Le fait d'être isolé impose qu'aucune information n'est copiée à l'extérieur du système durant sa transformation. C'est donc que les variables internes qui le décrivent gèrent l'évolution du système d'étape en étape sans que rien d'autre ne soit utilisé. Les lois de la physique ont probablement des formes particulières qui autorisent cela et même qui facilitent l'obtention de ces calculs *in situ*. Comprendre ce qui est faisable par des algorithmes sans mémoire est donc important non seulement pour les applications qu'on devrait en tirer dans la conception des programmes et des puces électroniques, mais aussi pour saisir les principes généraux de la physique.

S. Burckel explique que lorsqu'il a commencé à étudier cette algorithmique de l'économie de mémoire, il a cherché à identifier les transformations d'une suite de n variables booléennes calculables par un programme *in situ*. Prenant des transformations de ce type au hasard, il a découvert, à sa grande surprise, que chacune d'elles se laissait programmer *in situ*. Ce succès sur des exemples était certainement la conséquence d'un théorème général qu'il a fini par obtenir : toute transformation de l'ensemble $\{0,1\}^n$ (les n-uplets de 0 et de 1) dans lui-même se programme *in situ*.

Une longue quête a alors commencé pour, selon les différents types de transformations, élaborer les meilleures listes d'affectations possibles constituant les programmes *in situ*. Nous allons présenter quelques-uns des résultats obtenus.

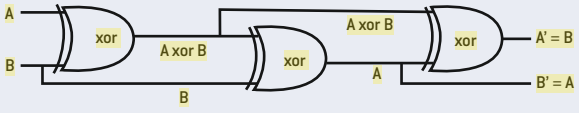
Les meilleurs résultats connus

Après les permutations circulaires de variables, les transformations les plus simples sont les permutations quelconques de variables, par exemple : $(A,B,C,D,\mathcal{E}) \rightarrow (B,\mathcal{E},D,C,A)$. Un peu d'attention montre que la transformation se décompose en 2 cycles : $A \rightarrow B \rightarrow \mathcal{E} \rightarrow A$ et $C \rightarrow D \rightarrow C$.

Autrement dit, on est ramené à deux permutations circulaires, l'une portant sur trois éléments, l'autre sur deux. Puisque chacune se décompose en suites d'affectations (respectivement quatre et trois affectations), ce sera le cas de la transformation qui réunit les deux cycles. Elle pourra donc être obtenue *in situ* en sept affectations.

Nous savons depuis des siècles que toute permutation se décompose en cycles. La méthode de décomposition proposée pour notre exemple se généralise donc sans difficulté : toute permutation possède un programme *in situ*.

3. Calculer sans croiser de fils



Le croisement de deux fils où circulent des informations est impossible si l'on interdit à un fil de passer sur l'autre (si deux fils se touchent, les informations vont se mélanger). L'utilisation de trois portes logiques *xor* permet l'équivalent d'un croisement de fils. On exploite la permutation *in situ* en trois affectations :
$A := A \text{ xor } B$, $B := ((A \text{ xor } B) \text{ xor } B) = A$; $A := ((A \text{ xor } B) \text{ xor } A = B$.

Plus généralement, si des informations arrivent sur n fils dans un certain ordre et doivent repartir dans un autre (c'est-à-dire une permutation quelconque de n fils, ce que l'on désigne parfois sous le nom de tresse) sans aucun croisement de fils, la programmation *in situ* de la permutation de variables donnera une solution utilisant au plus $3n/2$ portes logiques *xor*.

✔ BIBLIOGRAPHIE

S. Burckel, E. Gioan et E. Thomé, **Mapping computation with no memory**, dans *Unconventional Computation*, *8th International Conference Proceedings,* Lecture Notes in Computer Science n° 5715, pp. 85-97, Springer, 2009.

S. Burckel et E. Gioan, ***In situ* design of register operations**, *Proceedings of ISVLSI 2008*, IEEE Computer Society Annual Symposium on Very-Large-Scale Integration, 2008.

S. Burckel et M. Morillon, **Quadratic sequential computations**, *Theory of Computing Systems,* vol. 37(4), pp. 519-525, 2004.

S. Burckel et M. Morillon, **Sequential computation of linear boolean mappings**, *Theoretical Computer Science*, vol. 314, pp. 287-292, 2004.

S. Burckel, **Closed iterative calculus**, *Theoretical Computer Science*, vol. 158, pp. 371-378, 1996.