LOGIQUE & CALCUL

Le calculateur amnésique

Un calculateur sans mémoire est-il sérieusement limité ? Les résultats de l'algorithmique in situ *montrent que non : avec un peu d'astuce, il s'en sortira toujours.*

Jean-Paul DELAHAYE

J'ai la mémoire qui flanche
Je me souviens plus très bien
De quelle couleur étaient ses yeux
Je crois pas qu'ils étaient bleus
Étaient-ils verts ? Étaient-ils gris ?
Étaient-ils vert-de-gris ?
Ou changeaient-ils tout le temps de couleur
Pour un non, pour un oui ...
Jeanne Moreau, Cyrus Bassiak
et François Rauber

Si vous avez déjà écrit des programmes informatiques, vous avez nécessairement été confronté au problème suivant : deux nombres sont stockés dans les variables A et B, par exemple $A = 44$ et $B = 13$, et vous voulez échanger le contenu de A et B pour obtenir $A = 13$ et $B = 44$. La méthode classique, que tout le monde réinvente, consiste à utiliser une variable C et à écrire la séquence de trois instructions : $C := A$; $A := B$; $B := C$.

Le signe := est celui de l'affectation. L'instruction X := Y signifie que nous lisons le contenu de Y et que nous le mettons dans X, ce qui efface le contenu de X sans changer celui de Y. Cette séquence de trois affectations pour l'échange de deux valeurs exige l'utilisation d'une variable C supplémentaire. Or nous pouvons échapper à l'introduction de ce complément de mémoire par la séquence suivante : $A := A + B$; $B := A - B$; $A := A - B$. Le déroulement du calcul donne :

	A	B
Au départ :	44	13
Après $A := A + B$:	57	13
Après $B := A - B$:	57	44
Après $A := A - B$:	13	44

La suite d'instructions est valable avec n'importe quelles valeurs à la place de 44 et 13. Elle fonctionne aussi avec des valeurs booléennes où les signes + et – sont remplacés par *xor* qui désigne l'opération « ou exclusif » (0 xor 0 = 0 ; 0 xor 1 = 1 ; 1 xor 0 = 1 ; 1 xor 1 = 0). La suite d'instructions est :

$A := A$ xor B ; $B := A$ xor B ; $A := A$ xor B, où A et B valent 0 ou 1.

Vérifions que la séquence permute bien les valeurs de A et B en examinant l'effet de chaque opération. La première affectation met (A xor B) dans A. La deuxième met donc ((A xor B) xor B) dans B ; mais, d'après l'associativité du xor, cela vaut (A xor (B xor B)) c'est-à-dire A, puisque (B xor B) = 0 et que A xor 0 = A. La troisième affectation met donc (A xor B) xor A dans A ; comme (A xor B) xor A = (A xor (B xor A)), cela donne, par commutativité, (A xor (A xor B)) = ((A xor A) xor B) = (0 xor B) = B. Il y a bien échange.

Ce type de calcul est dénommé calcul *in situ* ou *calcul sans mémoire*. L'idée est d'opérer les calculs sans faire appel à des variables de stockage autres que celles qui communiquent les données au programme, que l'on utilise pour calculer et pour placer les résultats une fois les opérations terminées. L'étude de ce calcul *in situ* a pour fonction d'identifier ce que peut calculer un amnésique qui, ne disposant de rien d'autre que l'espace mémoire des données du calcul, le réutilise car il n'a en propre aucune mémoire à sa disposition.

Avant de donner des détails sur les résultats obtenus par les chercheurs qui étudient cette algorithmique *in situ*, et pour bien en mesurer la difficulté, posons-nous une petite énigme.

Considérons une permutation circulaire de trois données A, B, C, où l'on cherche à passer de $A = 3$, $B = 50$, $C = 700$ à $A = 50$, $B = 700$, $C = 3$. Nous voulons pour ce faire utiliser un programme *in situ*, c'est-à-dire n'utilisant aucune autre variable que A, B et C, et bien sûr nous voulons qu'il fonctionne pour tout jeu de données.

Permutation circulaire de trois valeurs

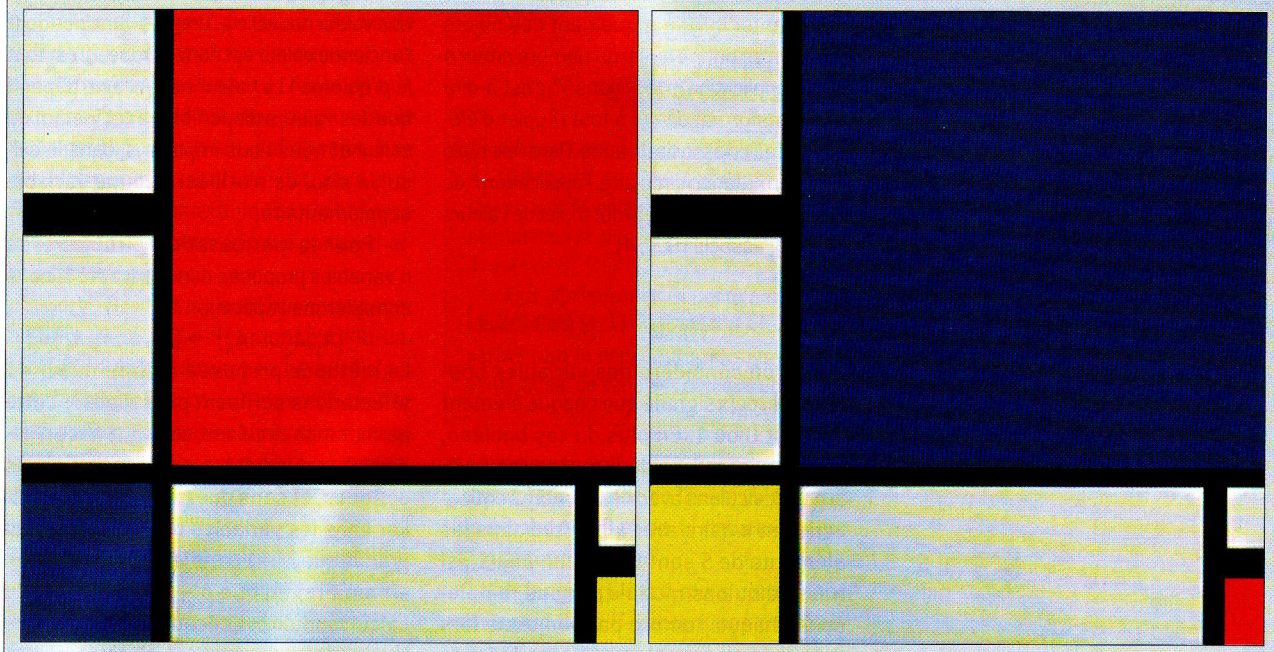
En voici un, avec le détail de l'état de chacune des variables lors du calcul :

	A	B	C
	3	50	700
$A := A + B$	53	50	700
$B := B + C$	53	750	700
$C := A - B + C$	53	750	3
$B := B - A + C$	53	700	3
$A := A - C$	50	700	3

En remplaçant les + et les – par des *xor*, on obtient, comme précédemment, un programme pour variables booléennes. Une autre méthode aurait consisté à échanger les valeurs de A et B, puis à échanger les valeurs de B et C en utilisant deux fois la méthode vue précédemment. Cela aurait nécessité six affectations alors que notre proposition n'en utilise que cinq. Peut-on faire mieux que cinq affectations ? La solution est donnée dans l'encadré ci-contre.

Expliquons maintenant avec plus de détails ce que les chercheurs dénomment calcul *in situ*. Un ensemble fini S est donné

1. La permutation circulaire



La question posée est : quel est le minimum d'affectations dont on a besoin dans un calcul *in situ* pour échanger circulairement les contenus de trois variables A, B, C, (A, B, C) devenant (B, C, A) ? Dans le tableau de Mondrian et son faux, trois couleurs ont été ainsi permutées. Petit exercice : quel est le vrai, quel est le faux ?

Notons d'abord que trois affectations ne suffiront pas. Il faudrait en effet, dès la première affectation, placer une bonne valeur là où on fait l'affectation, ce qui ferait disparaître la valeur qui y est stockée, dès lors irrécupérable.

En revanche, réaliser la permutation circulaire avec quatre affectations est possible, donc 4 est le nombre minimum d'affections nécessaires à la permutation circulaire de trois nombres. Voici la solution en quatre affectations :

	A	B	C
Au départ	A	B	C
A := A + B + C	A+B+C	B	C
C := A − B − C	A+B+C	B	A
B := A − B − C	A+B+C	C	A
A := A − B − C	B	C	A

Le procédé se généralise pour opérer une permutation circulaire entre n mémoires, c'est-à-dire pour passer de $(A_1A_2A_3...A_{n-1}A_n)$ à $(A_2A_3...A_{n-1}A_nA_1)$.

À la première étape, on place dans A_1 la somme $A_1 + A_2 + A_3 + ... + A_{n-1} + A_n$, puis, à chaque étape, toutes les valeurs initiales des variables sont présentes dans $A_2, A_3, ... A_{n-1}, A_n$ sauf une, ce qui permet, grâce à A_1, de la placer au bon endroit.

	A_1	A_2	A_3	.....	A_{n-1}	A_n
Au départ	A_1	A_2	A_3	.....	A_{n-1}	A_n
$A_1 := A_1 + A_2 + ... + A_n$	$A_1 + A_2 + ... + A_n$	A_2	A_3	.....	A_{n-1}	A_n
$A_n := A_1 - A_2 - ... - A_n$	$A_1 + A_2 + ... + A_n$	A_2	A_3	.....	A_{n-1}	A_1
$A_{n-1} := A_1 - A_2 - ... - A_n$	$A_1 + A_2 + ... + A_n$	A_2	A_3	.....	A_n	A_1
.....	.....	...	...	.....	...	...
$A_1 := A_1 - A_2 - ... - A_n$	A_2	A_3	A_4		A_n	A_1

L'affectation opérée pour A_n place donc bien la valeur initiale de A_1 en A_n, puis l'affectation opérée pour A_{n-1} place donc bien la valeur initiale de A_2 en A_{n-1}, etc. Le calcul est présenté avec des + et des −, et fonctionne avec des nombres entiers, réels ou même complexes.

Le calcul s'adapte aussi avec des nombres calculés modulo k, ou aux variables booléennes (en remplaçant les + et les − par *xor*, ce qui correspond d'ailleurs au + quand on calcule modulo 2). Cette remarque est importante car elle permet de répondre à une objection naturelle, mais essentielle, à la méthode proposée pour opérer une permutation circulaire de n entiers.

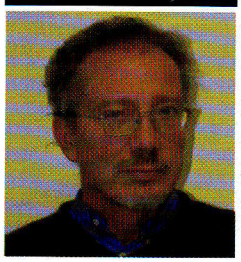
Voici l'objection : quand on calcule avec des entiers, la méthode de calcul *in situ* proposée triche, car elle stocke dans A_1 la somme des valeurs de toutes les variables, ce qui exige un espace mémoire plus grand pour la variable A_1 : s'il y a par exemple 10 variables variant entre 0 et 10, il faut pour stocker A_1 une variable capable de recevoir toutes les valeurs entre 0 et 100.

Voici la réponse à l'objection. Oui, cela est vrai pour la méthode avec des + et des −, c'est-à-dire avec les opérations arithmétiques habituelles entre nombres entiers. Cependant, puisqu'on peut mener un calcul modulo k (où k est un entier fixé), on dispose dans ce cas d'une méthode qui n'utilise pas plus d'espace pour A_1.

Une autre idée pour répondre à l'objection consiste à écrire les nombres entiers en base 2 et, au lieu de faire des additions ou des soustractions entre deux nombres X et Y, on calcule le nombre dont les chiffres binaires sont ceux obtenus en faisant le *xor* entre les chiffres binaires de X et de Y (exemple 1001 xor 0111 = 1110, autrement dit, 9 xor 7 = 14) et l'on retombe bien sur ses pieds à la fin.

L'espace nécessaire pour stocker A_1 reste encore identique à celui nécessaire pour stocker chaque nombre (p chiffres binaires par exemple si on manipule des nombres de p chiffres binaires). Les procédés décrits (traduits avec des calculs modulo k ou des *xor*) réalisent donc un calcul *in situ* sans avoir à considérer un espace mémoire particulier pour A_1.

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.

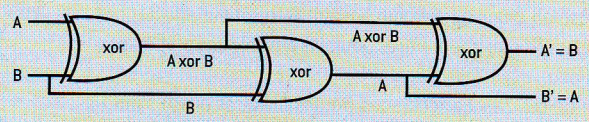
Les meilleurs résultats connus

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

4. Nouvelle décomposition de matrices pour le calcul *in situ*

La décomposition d'une application linéaire F en série d'affectations pour en tirer un programme *in situ* est toujours possible en $2n-1$ étapes, ce qui est très peu.

La lecture de ces $2n-1$ affectations (toutes linéaires aussi) s'interprète comme la décomposition de la matrice de départ en un produit de matrices simples : chacune est obtenue en modifiant une unique ligne à la matrice identité. C'est là un nouveau type de décomposition d'une matrice en produit de matrices simples.

Donnons un exemple avec un calcul modulo 2.

Nous voulons aboutir à l'affectation finale :

$F(A, B, C, D) = (A + D, A + B + C, A + C + D, A + C)$.

Comme toute application linéaire, on peut l'écrire sous la forme du produit d'une matrice et d'un vecteur colonne.

Multiplication de matrices

$$F = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C \\ A+C+D \\ A+C \end{pmatrix}$$

L'application est définie par le vecteur colonne indiqué. Toutefois, le calcul ne peut se faire *in situ* d'un seul coup car, dès que l'on a remplacé A par $A + D$, il faut réaménager le calcul en fonction de cette nouvelle valeur de A, et ainsi de suite pour toutes les variables. Cela peut se faire en utilisant des pseudomatrices unités où une seule ligne est transformée. Ainsi, la matrice F de la transformation est décomposée en un produit des matrices J, K, L et M, explicitées ci-dessus. On obtient avec ces matrices :

$F(A, B, C, D) = MLKJ(A, B, C, D) = (A + D, A + B + C, A + C + D, A + C)$.

Multiplication de matrices

$$J = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ B \\ C \\ D \end{pmatrix}$$

Multiplication de matrices *Calcul modulo 2*

$$K = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A+D \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C+2D \\ C \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C \\ C \\ D \end{pmatrix}$$

Multiplication de matrices

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A+D \\ A+B+C \\ C \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C \\ A+C+D \\ D \end{pmatrix}$$

Multiplication de matrices *Calcul modulo 2*

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} A+D \\ A+B+C \\ A+C+D \\ D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C \\ A+C+D \\ A+C+2D \end{pmatrix} = \begin{pmatrix} A+D \\ A+B+C \\ A+C+D \\ A+C \end{pmatrix}$$

Puisqu'il faut $n + 1$ affectations pour une permutation circulaire entre n variables, il faut dans le pire cas (quand tous les cycles sont des cycles d'ordre 2) $3n/2$ affectations si n est pair, et $(3n - 1)/2$ affectations si n est impair. Dans le cas d'une permutation se décomposant en k cycles non réduits à un point, $c_1, c_2, ..., c_k$ de longueurs $|c_1|, |c_2|, ..., |c_k|$, le programme *in situ*, résultat de l'idée indiquée au-dessus, est composé de $k + |c_1| + |c_2| + ... + |c_k|$ affectations. Ce nombre est le meilleur qu'on puisse obtenir. La décomposition d'une permutation quelconque de variables en un programme *in situ* de $3n/2$ opérations au plus permet d'échanger de l'information entre n fils qui ne peuvent pas se croiser (car placés sur un même plan), en utilisant $3n/2$ portes logiques *xor* au plus *(voir l'encadré 3)*.

D'autres résultats ont été obtenus pour les applications linéaires. Tout d'abord sur des variables booléennes et plus généralement modulo p avec p premier (S. Burckel et Marianne Morillon) et par la suite sur des entiers modulo k, pour tout entier k fixé (Emmanuel Thomé). Un exemple est donné dans l'encadré ci-dessus.

Les résultats obtenus indiquent que toute transformation linéaire peut s'obtenir par un programme *in situ* comportant au plus $2n-1$ affectations, chacune correspondant à une opération elle-même linéaire (comme dans l'exemple). De plus, on peut imposer l'ordre dans lequel se feront les affectations : 1, 2, ..., n, $n-1$, ..., 2, 1 (affectation de la première variable, de la deuxième, etc.)

Trouver le simple est compliqué

Toute transformation n'est pas linéaire, aussi fallait-il étudier le cas général. Dans un premier temps, il a été établi que toutes les transformations bijectives (linéaires ou non) F de S^n dans S^n possédaient des programmes *in situ* et que $2n - 1$ affectations suffisent. S. Burckel et Emeric Gioan ont ensuite montré que $5n-4$ affectations suffisent toujours, quelle que soit la complexité de F et cela même quand F n'est pas bijective. Lorsque le nombre d'éléments de S est une puissance de 2, on peut même descendre jusqu'à $4n-3$.

Ces résultats récents améliorent un résultat précédent obtenu par S. Burckel et M. Morillon qui indiquait que n^2 affectations suffisaient dans le cas booléen. Le gain de n^2 à $4n - 3$ est considérable. Malheureusement, le calcul des décompositions est parfois très difficile et lorsque le nombre de variables est grand, on risque de se heurter à une impossibilité pratique. Autrement dit : la complexité des algorithmes pour calculer les décompositions les plus simples est grande. On a là, une fois encore, une illustration de la maxime « Trouver le simple est compliqué ».

Ces résultats ne donnent pas nécessairement les décompositions optimales, qui sont encore plus difficiles à trouver. Les explorations exhaustives sont en théorie envisageables, puisque nous sommes dans un cas fini. Elles garantissent de trouver les programmes *in situ* optimaux, mais elles sont si coûteuses en calcul qu'on ne peut les envisager que si n est vraiment petit (inférieur à 10). De nombreux résultats sont donc encore attendus dans ce domaine à la fois combinatoire, algébrique et algorithmique. Il est à parier qu'à mesure des perfectionnements apportés aux méthodes pratiques pour calculer les décompositions en programmes *in situ*, leur intérêt concret apparaîtra et leur utilisation se généralisera.