APL-CAM Journal, BACUS, 10, 2, 316-323, avril 1988.
Reproduit des Comptes Rendus des Journées d'Etude, organisées par le Groupe de Travail APL de l'AFCET, Chatenay Malabry, France, 2223 Avril 1982; Editeurs: G. Martin et B. Mailhol; Publié par l'Association Française pour la Cybernétique Economique et Technique, AFCET, Paris, France, 1982; pp. 141-148.

Pourquoi utiliser préférentiellement les vecteurs dans une programmation en APL

Gérard A. Langlet
Animateur APL de l'AFCET
Délégué par l'AFNOR pour la normalisation d'APL
CEA/IRDI/DESICP/SCM
Centre d'Etudes Nucléaires de Saclay
F-91191 Gif-sur-Yvette

Résumé

Ce papier montre que l'utilisation préférentielle des vecteurs conduit à une programmation très modulaire, utilisant peu ou pas d'étiquettes, ni de variables locales. D'autre part, APL étant évidemment le meilleur outil de test d'algorithmes et de modélisation pour gros systèmes, la transcription dans les langages traditionnels ou dans les langages-machine est grandement facilitée. L'inroduction des processeurs vectoriels (CRAY ONE par exemple), renforce cette tendance. Enfin une telle pratique améliore significativement les performances.

1. Introduction

Dès son avènement, APL se présentait comme le meilleur langage de traitement de données structurées en tableaux et on peut raisonnablement affirmer qu'il l'est encore aujourd'hui. Il existe en fait maintenant deux types de programmeurs APL : celui qui est parvenu à utiliser ce langage après avoir suivi les voies classiques (et les affres) de l'informatique, et celui qui a, un jour, découvert APL et l'a immédiatement appliqué à la résolution du problème qui le motivait à ce moment-là, puis s'est aperçu qu'il devenait informaticien comme M. Jourdain faisait de la prose.

Lequel a priori développe les meilleurs programmes ? Combien d'années fault-il pour devenir un Docteur ès-APL ? Peut-on se passer de toute méthodologie ? Doit-on connaître tout APL pour devenir compétent ? Le programme le plus court est-il souvent le meilleur ?

Ce sont autant de questions que se posent aussi bien les débutants que les programmeurs chevronnés, et auxquelles il ne semble pas si simple de répondre d'une manière définitive.

Il faut d'abord prendre conscience qu'APL est, entre autres et *ab origine*, une notation si puissante qu'elle recouvre et dépasse souvent les possibilités de la notation mathématique conventionelle élaborée au fil des siècles par quelques grands esprits. APL n'a acquis et mérité son nom qu'une fois la preuve établie que cette notation permettait aussi de programmer en prime divers ordinateurs existants et de rivaliser ainsi brillamment avec quelques concurrents, ce qui ne fut pas toujours bien accueilli.

Grâce à un noyau relativement restreint de gens opiniâtres et parfois réprouvés, APL fonctionne aujourd'hui pratiquement chez tous les constructeurs de gros, moyens et petits matériels, et lorsque l'effort de normalisation actuellement entrepris dans le monde, et en France particulièrement, aura porté ses fruits, on ose espérer que l'enseignement de ce langage et l'enseignement à l'aide de ce langage recueilleront des suffrages quasi-unanimes...

Pour nous utilisateurs, APL se juge par la façon dont tournent les applications. Les gens qui souvent émettent des jugements ne se rendent pas compte de l'apport d'APL au niveau de la souplesse, qualité nécessaire dès qu'on aborde des domaines de modélisation de plus en plus complexes, par exemple en Conception Assistée par Ordinateur, avec constitution d'importantes Bases de Données, et utilisation de logiciels graphiques tridimensionnels.

On a maintenant tellement pris l'habitude de manipuler des tableaux à 2 dimensions, puis 3, puis 4 ou plus, depuis qu'au fil des ans les limitations des espaces de travail reculent de tous côtés, qu'on finit par oublier un point caractéristique de la manière dont APL opère, et que les habitués des autres langages appliquent, hélas pour eux, par pure obligation :

"Tout travail mérite salaire"

Autrement dit : la facilité apparente que procure APL se paie toujours quelque part dans la chaine d'exécution entre l'interprétation d'une instruction géniale et la restitution finale des résultats attendus. Le fait de se dispenser de déclarer type et dimensions implique nécessairement que ces derniers attributs seront testés sur chacun des arguments de chacune des primitives, et ce de façon plus que redondante, dans toute instruction apparemment triviale.

L'accumulation de ces tests réduit globalement les performances de rapidité, et comme le programmeur APL, se permet le luxe suprème d'ignorer les boucles, le pauvre interpréteur doit les reconstituer, puis les exécuter sans brancher…

Toutefois, il se comporte selon que ce qu'on lui a appris lors de sa mise au monde, et tous les cas possibles sont encore bien loin de la solution idéale pour laquelle il irait chercher le meilleur algorithme infaillible en langage-machine.

Pour obtenir l'appellation APL, il ne devrait plus suffire, de nos jours, de mettre sur le marché des interpréteurs capables de digérer à peu près la plupart des expressions courantes. On doit requérir un minimum de performances en précision, en temps total et en gestion dynamique de la mémoire pour éviter au passage le remplissage de celle-ci par des tableaux temporaires inutiles ou de type mal optimisé (par exemple les nombres binaires traités comme des entiers).

Pour l'instant néanmoins, si on se contente de ce qui existe et rend des services non négligeables, une certaine méthodologie de la part du programmeur parvient à améliorer notablement le comportement parfois étrange des systèmes disponibles.

Certains constructeurs, fiers de leur produit vont jusqu'à distribuer des tableaux comparatifs pour les temps d'exécution des primitives de leur APL et de celles de leurs concurrents. Nous allons un peu les imiter et en tirer quelques conclusions.

2. In medio stat virtus

Le vecteur APL se situe dans le juste milieu entre le scalaire et les tableaux de rang supérieur. Cette position priviligiée l'est à un tel point que les primitives scalaires convertissent les arguments scalaires en vecteurs (ou en tableaux) assez fréquemment comme chacun sait, du moins en théorie. Il n'est pas souhaitable en effet que les interpréteurs effectuent réellement cette besogne dans tous les cas pour des raisons évidentes d'encombrement. Mais que dire par exemple du système APLSV qui, dans le cas d'une expression comme :

```
V∘.+100⍴0
```
(V étant un vecteur d'entiers)

trouve le moyen de convertir chaque composante du vecteur binaire `100⍴0` en un entier chaque fois qu'il en a besoin dans le produit extérieur?

`V∘.+100⍴+0` consomme un temps inférieur de 30 % pour `50=⍴V`, et de 50 % pour `100=⍴V`, car le + monadique supplémentaire effectue la conversion une fois pour toutes… Noter que `⍉(100,⍴V)⍴V`, expression moins élégante, s'avère beaucoup plus rapide.

Tout le monde connait la lenteur de la factorielle, assez peu utilisée du reste, et s'est rendu compte qu'il valait souvent mieux recourir à une fonction définie qu'à la fonction primitive. Il en est souvent ainsi pour l'appartenance et l'index (`⍳` dyadique), et aussi pour les primitives `⊤` et `⊥`.

À propos de l'appartenance, nous allons essayer de présenter un exemple précis qui montre au passage que cette primitive devrait être étendue pour accepter des spécifications d'axe(s). Si V1 et V2 sont deux tableaux possédant le même nombre de lignes, mais un nombre en général différent de colonnes, on veut fabriquer un tableau binaire R exprimant si chaque élément de V2 appartient ou non à la ligne correspondante de V1. Un fort en thème écrit immédiatement l'expression :

```
R←1 2 1 ⍉ ∨/ V2∘.=V1
```

qui va surtout exécuter un travail en partie inutile, tandis qu'un informaticien débutant en APL, mais féru de boucles, attellera ces dernières à la charrue pour tracer ses sillons. Il ne faut surtout pas se moquer, car, comme le prouvent les mesures présentées sur la Figure 1, il y a des cas où les mains plaines appartiennent bien aux innocents.

Le troisième exemple ne peut fonctionner que pour des tableaux numériques, et malgré le soin apporté à éviter les pièges de la conversion de 0 binaire mentionnés ci-dessus, n'a comme avantage que l'absence d'étiquettes et de variables locales.

La dernière solution proposée n'est que la réduction dimensionelle de la première solution, c'est-à-dire la recherche et l'élimination de toute opération inutile que l'on a tendance à effectuer en APL grâce à la facilité apparente de traiter des tableaux parfois gigantesques en ignorant les boucles.

Pour tenter de résumer tout ceci, et pour élargir la signification du titre de cet article, nous érigerons cette constation en règle de déontologie.

Temps Consommé

	IBM 5110 Secondes	VSPC (370-168) Ms

```
    ρV2
27 6
    ρV3
27 8
    ∇ R←A AP1 B
[1] R←(⎕IO+0 1 0)⍉∨/A∘.=B
    ∇
    R1←V3 AP1 V2                          170        138
    ∇ R←A AP2 B;L;⎕IO
[1] ⎕IO←1
[2] R←(ρA)ρ1←0
[3] L←1↑ρA
[4] E:R[I;]←A[I;]∊B[I←I+1;]
[5] →(L>I)/E
    ∇
    R2←V3 AP2 V2                           13         38
    ∇ R←A AP3 B;⎕IO
[1] ⎕IO←O
[2] R←(⍳1↑ρA)×1+⌈/(,A),,B
[3] R←(A+R∘.+(1↓ρA)ρ+0)∊,B+R∘.+(1↓ρB)ρ+0
    ∇
    R3←V3 AP3 V2                           47         36
    ∇ R←A AP4 B;D
[1] D←ρA
[2] R←ρB
[3] R←⌽ρB←((¯1↑D),R)ρB
[4] A←⍉RρA
[5] R←⍉∨/A=B
    ∇
    R4←V3 AP4 V2                            7         21
    ∧/,R1=R2
1
    ∧/,R1=R3
1
    ∧/,R1=R4
1
```

La fonction AP3 a été incluse ici pour montrer que sur un APL moderne la programmation en tableau parvient à battre de peu la programmation en boucle, ce qui n'est pas vrai sur les systèmes plus anciens.

(V2 et V3 sont des entiers)

Figure 1

Un aspect peut-être assez peu connu des comparaisons de performances au sein d'un même APL est celui de la vitesse de réduction d'un tableau selon des axes différents.

Soit par exemple une matrice carrée binaire symétrique T que nous réduirons selon les lignes, puis selon les colonnes sachant que le résultat est identique dans les deux cas.

```
T←Tv⍉T←(N,N)⍴1 0 0 1 0 1 1 0 1 0 1
                APLSV (IBM 5110) VSPC (370-168)
                N=100 (Secondes)   N=500 (Ms)
R←∨/T                    4               5
R←∨≠T                   59               4
R←∨/⍉T                  32             439
R←∧/T                    2               6
R←∧≠T                   46               4
R←∧/⍉T                  37             454
```

Figure 2

Les résultats de la Figure 2 font apparaitre un écart important en défaveur de la réduction selon la première dimension, à tel point qu'il devient plus avantageux, sous APLSV, d'effectuer en premier une transposition. Il s'agit pourtant d'un cas trivial.

0n subodore que VSPC, plus récent, doit présenter de meilleurs atouts. C'est effectivement le cas, puisque la réduction selon la première dimension semble cette fois plus rapide que sa collègue. Remarquer que la transposition consomme encore un temps considérable; espérons que dans les futurs systèmes, la transposition ne sera plus physiquement pratiquée, mais simplement prise en compte dans la représentation interne des variables.

```
VSPC:   A←1,500000⍴0
        B←(500000⍴0),1
        C←~A
                        Ms
+/A                     67
∨/A                      2
∧/A                      1
+/B                     67
∨/B                     66
∧/C                      2
∨/C                      1
```

Figure 3

La Figure 3 est un tableau de résultats de mesures de temps pour la réduction sur des vecteurs binaires.

Le nombre de composantes est le double de celui du tableau T considéré dans la deuxième colonne de la Figure 2. Le temps-machine consommé atteint des valeurs relativement faibles; ceci montre que l'interpréteur est bien optimisé pour des vecteurs et ceci reste vrai pour la plupart des implémentations, même anciennes, si on compare à ce qui se passe pour des tableaux, surtout de rang élevé.

On objectera qu'il est impossible de tout calculer en vecteurs. Eh bien a priori oui, mais en y réfléchissant un peu mieux, toute une série de "superprimitives", c'est-à-dire de fonctions définies susceptibles de constituer une bibliothèque d'outils vraiment efficaces, en gestion aussi bien que pour le calcul scientifique, peut s'écrire selon ce principe. C'est ce que nous avons pratiqué depuis déjà assez longtemps pour pouvoir en clamer la rentabilité.

En général, la programmation de ces superprimitives obéit aux règles suivantes :

1) Les arguments ont comme noms : R pour le résultat, A pour l'argument gauche et B pour l'argument droit.
2) La programmation doit rester indépendante de ⎕IO ; sinon, ce dernier paramètre-système est déclaré local.
3) Le nombre de variables locales est minimal, et de préférence 0.
4) Le nombre d'étiquettes est nul ; le seul branchement permis est →0 (on tolèrera à la rigueur →⎕LC). S'il s'avère impossible d'éviter la présence d'étiquettes, après bien des essais toutefois, c'est que la fonction ne peut constituer une vraie superprimitive, mais doit se décomposer en deux ou plusieurs nouvelles fonctions. On parvient ainsi à concevoir différents niveaux de superprimitives, au besoin récursives.
5) Le traitement algorithmique est pensé vectoriellement. Si le résultat doit être un tableau, il est simplement restructuré dans la dernière instruction de la superprimitive.
6) Le nombre de lignes de la fonction se limitera à un nombre très petit, de sorte que le cerveau humain puisse la comprendre sans dépense abusive de phosphore.

3. Exemples

1. La fonction SEL, comme son nom l'indique, sélectionne A[i] fois la composante B[i] en la répétant au besoin. Dans une expression comme R←A SEL B, les vecteurs A et B doivent avoir la même dimension et A[i] doit être un entier non négatif. Si A ne contient que 0 ou 1, l'expression équivaut à R←A/B.

Ainsi 2 0 4 SEL 5 6 7 a pour résultat 5 5 7 7 7 7 ; on regrette souvent que tous les systèmes APL ne proposent pas encore cette possibilité comme extension du langage. La Figure 4 montre la liste d'une des fonctions que l'on peut écrire pour pallier cet inconvénient, et donne quelques mesures de temps-machine dans un cas bien particulier.

```
    V←¯50000+?1000⍴99999          V est entier
    L3←1∧L2←L1←?1000⍴2×⎕I0←0      L3 est binaire
    L1[327 453 235]←4 5 6         L1 et L2 sont entiers
                                       5110        VSPC
    R1←L1 SEL V                     12 sec        20 Ms
    R2←L2/V                       6.5 sec        12 Ms
    R3←L3/V                         2 sec         4 Ms
    ∇ R←A SEL B;⎕I0           Remarque
[1] ⎕IO←1                     L1 SEL V est accepté directe-
[2] ⍎(0=⍴,A)⌈⍴,B)/'→R←O⍴B'    ment dans certaines implémen-
[3] ⍎(1≥⌈/A)/'→0,⍴R←A/B'      tations sous la forme étendue
[4] ⍎(0=⍴⍴A)/'A←(⍴B)⍴A'       de la réduction: L1/V
[5] B←(R←A≠0)/B
[6] R←(+/A←R/A)⍴0              Avec les mêmes vecteurs, nous
[7] R[+\1,¯1↓A]←1             obtenons (sur IP Sharp) 90 MS
[8] R←B[+\R]                  pour SEL contre 30 MS pour /
    ∇
```

Figure 4

Sur 1000 composantes, le vecteur L2 n'en possède que trois différentes de 0 et de 1, de telle sorte que la comparaison avec la réduction pure par un vecteur de même taille

devienne possible. On se rend compte dans ces conditions, que l'utilisation de SEL consomme moins du double de temps que la réduction par un vecteur binaire codé en entiers ! Nous remarquons au passage que cette dernière consomme par contre trois fois plus de temps que la réduction binaire normale, toujours à cause des problèmes de conversion et nous en concluons que la fonction SEL constitue une superprimitive valable. Noter que les trois instructions débutant par ⍋ traitent les effets de bords (arguments vides ou scalaires et argument gauche binaire) et participent allègrement à la consommation des millisecondes.

2. Une superprimitive assez importante est le tri croissant (ou décroissant) à l'intérieur de chaque ligne (ou colonne) d'un tableau numérique. En effet, lorsqu'un tableau se trouve agencé de cette manière, il devient aisé à l'aide de R=1⌽R, de compter les occurrences multiples dans chacune des lignes, évidemment sans boucle.

La Figure 5 ci-après présente la fonction TRIL, qui effectue une opération de tri croissant dans chaque ligne d'un tableau, mais aussi dans chaque sous-tableau lorsque T est un tableau de rang n, alors considéré comme un vecteur de sous-tableaux de rang n-1. C'est d'ailleurs la raison pour laquelle la ligne [4] se trouve programmée de cette façon et génère momentanément un objet autre qu'un vecteur… On a encore pris la précaution de forcer la conversion binaire en numérique grâce au + monadique.

La dépense de temps-machine reste tout-à-fait acceptable lorsqu'on la compare à celle d'un simple tri sur un vecteur de longueur égale à celle du tableau T.

```
    ∇ FAIRET;⎕RL;⎕IO              Fabrique un tableau T
[1] ⎕IO←0                         N=50  sur  IBM 5110
[2] ⎕RL←7⋆5                       N=500 sous VSPC
[3] T←¯50000+?(N,20)⍴99999
    ∇V
    ∇ R←TRIL B;D                  Temps pour R1 TRIL T
[1] D←⍴B                          5110       VSPC
[21 B←,B                          27 sec     475 Ms
[31 R←1+(⌈/B)-⌊/B
[4] R←,(R×⍳1↑D)∘.+(1↓D)⍴←0       Temps pour un tri simple
[5] B←R+B                         sur le vecteur TT←,T
[6] B←B[⍋B]                       R2←TT[⍋TT]
[7] R←D⍴B-R                       5110       VSPC
    ∇                             22 sec     430 Ms
```

Figure 5

On n'est point parvenu à supprimer la seule variable locale qui subsiste, mais malgré la présence de l, cette fonction délivre des résultats identiques en origine 1 et en origine 0, si bien que la déclaration de ⎕IO en local a pu devenir facultative.

4. Conséquences

De petits programmes modulaires et lisibles, sans renvois et sans pléthore de variables, se transcrivent très facilement en Fortran ou en Assembleur(s). Comme ils ont été développés en APL et mis au point contre la montre, ils fonctionnent alors tout de suite et d'une façon optimale la plupart du temps. Tandis qu'un programme Fortran écrit même et je dirais surtout à partir d'un organigramme, ne se comporte jamais comme la théorie le prévoit, et exige un effort de révision considérable pour des modifications parfois bénignes, les dérivés des superprimitives ont des performances supérieures, à

cause de leur simplicité. Ils ne contiennent pas de boucles imbriquées, se substituent les uns aux autres de façon modulaire et exigent peu de documentation pour se faire comprendre.

La programmation vectorielle, née en pratique avec APL, a été redécouverte assez récemment, peut-on dire, et fait des progrès considérables avec la mise en service de CRAY-1 et, dans l'avenir, de CRAY-2… Le niveau interne des boucles du Fortran disparait en fait totalement dans le code, et la transcription des superprimitives devient encore plus facile, à tel point qu'on peut envisager de la rendre complètement automatique.

Il va de soi que les superprimitives constituent aussi des modèles pour d'éventuelles extensions d'APL et surtout pour l'écriture de nouveaux macro-langages destinés à des domaines particuliers du traitement de données, par exemple dans l'enseignement.