Texte dans la série "La Programmation en Vecteurs" et "APLl vs APL2". L'auteur remarque qu'il serait intéressant de compléter les tests sur d'autres machines et sous l'APL2 Version 3 sur 3090 avec coprocesseur vectoriel. Les lecteurs qui seraient intéressés sont priés d'envoyer leurs résultats à l'adresse de notre association.

What is a Pernicious Loop?

Gérard A. Langlet
CEA/IRDI/DESICP/DPC
C.E.N. Saclay
F-91191 Gif-sur-Yvette France

Pernicious Loops

A loop is said to be "pernicious" when it does not appear in the notation that is used to write an expression, but strongly increases the CPU consumption because it will be generated underneath by the current implementation. This is the case for the "each" operator in APL2 and APL2-like implementations. Pernicious loops will also be enco~tered with some extensions of other operators such as "reduction" and "scan" to mixed and user-defined functions.

Spectacular examples are given in Table I with `∈⍳¨V`, an expression that apnarently looks nice to any APL2 fan, and with `⊃,/⍳¨V`, wich is much worse, since it contains "`,/`" as well as "each", and is an ersatz which can be used in some extenderl APL implementations when they do not sunport "enlist" yet.

This is one of the most exciting benchmarks I have ever made, because so much information came out from some simple comparisons.

Everybody has felt for a long time the need for a IOTA function that could work on a vector so that:

```
      IOTA 3 2 0 5 7 1                          returns:
1 2 3 1 2 1 2 3 4 5 1 2 3 4 5 6 7 1             in origin 1.
```

The above expressions are easy to imagine with extended implementations and run all right ... on short vectors. But when you have an ISO-APLl interpreter, what should you do to write your IOTA function?

The worst solution, that we do not consider in the bench, would be to define a function with an explicit loop that would concatenate every `⍳V[I]` successively, which is just what, in fact, the "each" operator does within a vector of vectors. An important improvement results from the rule that states: "do not use catenate for large arrays within loops"; first try to know the shape of the result vector, generate it with fills of the same type as the supposed type of the result; if the result is too big for your workspace, the WS FULL message will appear immediately; in the opposite favourable case, fill the result by successive indeexing. In fact, you do so when programming in Fortran, Cobol or C, since catenate is just (is it really?) a "priviliege" of APL. As far as I am concerned, I always discard "catenate" from everyday-programming in APL except for very short vectors.

Another bunch of pernicious loops is also provided by the "execute" primitive which, in addition, allows splendid pornography that however stricto senso conforms perfectly to

the present ISO standard. See expression 2). Of course, it can help you in provisory versions of your applications if you have not yet found any other means of obtaining your result. This expression also combines a strong use of "catenate" on large vectors. It cannot be anything but slow and difficult to maintain. Moreover you sometimes encounter system limits, generally the maximum number of characters that is admitted in the argument of "execute", or' the maximum size of the execute buffer, and, in some cases the symbol table overflows... Even with N=100, you cannot get any result with 2) in APL90 (⍎ admits 256 characters as a maximum in my version), and it is perhaps a good thing so far. Curiously, IBM APL on PC allovs about 8K of argument for "execute", and IBM APL2 on PC only admits 4K... Even with the 64K limit of APL.68000, the problem still exists, as shown in Table 1. Programminp expressions such as 2) is encouraged only by the system accoutant!

Since expression 0") is the shortest and the one whicll is taught in APL2 manuals, it should work fast on large vectors. Then, why do we get so quickly a WS FULL with a 800K ws? A good question to the implementers. Because of the short-integer coding, this problem arises to a lesser extent in a PC... Note that you must not be in a hurry anyway.

Expression 0') is also short and surely has a dazzling effect on the reader especially if he does not practice extended APL yet. Our benchmark shows that it is as catastrophic as expression 2) which has been murdered herebove.

As I have frequently repeated this assertion in the past, just try to think in simple vectors and you will find solutions for which APL is appropriate. Expression 1) does not respect the ISO-APLl standard, because it uses the "replicate" extension of "compression". However, this extension exists in most small implementations, and is generally well-programmed. Moreover, a unarnimous consensus will push it to be included soon into the next standard. Good programming in a low-cost, (sometimes free) APL is possible, although some implementations mav be very slow for a reason of double interpretation. Another remark is the following: small APL interpreters execute ISO-APL expressions much faster than extentded APL ones, because their data structures are more simple and shorter; they also have less branches in their internal code.

In general, simple and frequently-used primitives or combinations such as +/ or +\ have been optimized by the implementers. Try to use a good subset of APL, avoiding "encode" "decode" "execute" "catenate" and some newly-intoduced operators which are only nice on the paper and lead to endless discussions among "specialists". APL is a laxist language. Try to discover its tricks as well as its traps: only practice can help you.

Table 1 - Comparative Benchmark of Expressions:

O') ⊃,/⍳¨V
O") ∈⍳¨V (in APL2 - using "enlist")
1) (⍳+/V)+V/V-+\V
2) ⍎1⌽,')',',',',(',',⍳',⍕V∘.+,⌊O
with V←N⍴10 and ~⎕IO←1.
All measures are averaged in milliseconds.
~ means: expression not admitted in this APL.

With N=10, all these expressions return:
1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 1 2 3 4 5 6 1 2 3 4 5 6 7 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 10

	N =	100	1000	5000	1 0000
PC AT 640K 8 MHz (same machine for all checks)					
IBM APL2	O')	900	16600	346000	WS FULL
IBM APL2	O")	500	6000	60000	203000
APL*PLUS PC 7.0	1)	80	610	3000	WS FULL
IBM APL V 2	1)	240	5600	30400	WS FULL
IBM APL2	1)	220	1820	9010	WS FULL
SHARP APL (freeware)	1)	900	5350	25270	WS FULL
I-APL (freeware)	1)	4200	WS FULL	…….	
APL*PLUS PC 7.0	2)	660	WS FULL	…….	
IBM APL V 2	2)	1000	WS FULL	…….	
IBM APL2	2)	1100	SYSTEM LIMIT (due to ⍎) (22640 ms for limit N=844)		
ATARI 1040 & MEGA ST (68000)					
APL.68000 V 6.05 (runtime is freeware)	1)	60	540	2680	5240
SUN 3-160 (6802(» (same machine with 1 MB ws for all checks)					
APL.68000 V 6.09	O')	~	~	~	~
APL90	O')	250	4200	54000	189000
DYALOG APL 5.0 rel 6	O')	140	3800	77000	320000
APL.68000 V 6.09	1)	40	180	740	1380
DYALOG APL 5.0 rel 6	1)	60	540	3680	5400
APL90	1)	60	560	3500	7000
APL90	2)	LIMIT ERROR (due to ⍎)			……..
DYALOG APL 5.0 rel 6	2)	250	4200	85000	302000
APL.68000 V 6.09	2)	360	12200	300000	LIMIT ERROR (due to ⍎)
IBM 3090 with APL2 1.0 (⎕WA=810628 in a clear ws)					
APL2 1.0	0")	3	19	160	WS FULL
APL2 1.0	0')	10	200	5200	WS FULL
APL2 1.0	1)	2	15	80	156
APL2 1.0	2)	10	233	4946	25980