APL Optimization Techniques
For Commercial Applications

Eugene Ying
2013
Topics

1. Table Look Up
2. Arithmetic Optimization
3. Scalar Extension
Assumptions

- Dyalog APL Version 13.2.17248.0 64 bit Classic
- Using defaults that come with V13.2
- Running on IBM power7 processor (3.1Ghz)
- In AIX 6.1
Topic #1
Table Look Up
Vector Membership Function

Subject Argument $s$

\[
\begin{array}{cccc}
18 & 65 & 87 & \ldots \\
\end{array}
\]

Principal Argument $P$

\[
\begin{array}{cccccc}
20 & 37 & 65 & 81 & 98 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & \ldots \\
\end{array}
\]
Tables in Commercial Data Processing are not always numeric

Ticker Symbols
Customer Names
User IDs
Machine Names
Model IDs
Product Codes
...

Commercial Applications

Your Customers
- Exxon Mobil
- Chrysler Group
- Apple
- RadioShack

Subject Argument

Fortune 500 List
- Wal-Mart Stores
- Exxon Mobil
- Chevron
- Phillips 66
- Berkshire Hathaway
- Apple
- ...

Principal Argument

Answer is 1 0 1 0
Matrix Membership Function

Inner Product

\( v/\text{subject}^\land.=\varnothing \text{PRINCIPAL} \)

\( v/s^\land.=\varnothing p \)
Memory Requirement of Membership of Char Matrix $s$ in Char Matrix $P$

$$1000 \ 6 = \rho P$$
Elapsed Time of Membership of Char Matrix $s$ in Char Matrix $P$

$1000 \ 6 = \rho P$
Temporary Memory Requirement

\( \vee / s^\wedge. = \phi P \) requires a matrix transpose that takes up temporary WS and processing time, followed by an inner product that creates a sparse matrix that takes up more temporary WS.
Matrix Membership Function

using inner product
\[ \forall \text{subject} \Rightarrow \phi \text{PRINCIPAL} \]

using nested vector
\[ (\downarrow \text{subject}) \in \downarrow \text{PRINCIPAL} \]
Memory Requirement of Membership of Char Matrix $s$ in Char Matrix $P$

$$1000 \ 6 = \rho P$$
Elapsed Time of Membership of Char Matrix \( s \) in Char Matrix \( P \)

\[
1000 \ 6 = \rho P
\]
Relationship Between Membership $\in$ and Index Of $\iota$

$s \in P$
can be emulated by
$(\rho P) \geq P \iota s$

Similarly
$(\downarrow s) \in \downarrow P$
can be emulated by
$(1 \uparrow \rho P) \geq (\downarrow P) \iota \downarrow s$

or
$(1 \uparrow \rho P) \geq P \{ (\downarrow \alpha) \iota \downarrow \omega \} s$
Memory Requirement of Membership of Char Matrix $s$ in Char Matrix $P$

$1000 \ 6 = \rho P$
Elapsed Time of Membership of Char Matrix $s$ in Char Matrix $P$

$1000 \ 6 = \rho P$

$((↓s) ∈ ↓P) \geq P\{(↓α) \bowtie ↓ω\}s$
If PRINCIPAL argument P is inside a loop and remains constant,

:For I :in ⌽LOOP
    B←(↓s)®P
:Endfor

then P can be taken outside the loop for hash table.

Z←P
HASH←®Z
:For I :in ⌽LOOP
    B←HASH ↓s
:Endfor
Memory Requirement of Membership of Char Matrix $s$ in Char Matrix $P$

$1000 \cdot 6 = pP$
Elapsed Time of Membership of Char Matrix \( s \) in Char Matrix \( P \)

\[ 1000 \ 6 = \rho P \]
If PRINCIPAL argument P is inside a loop and remains constant,

```plaintext
:For I :in ⌽LOOP
    B←(1↑ρP)≥P{(↓⍺)↓⍵}s
:Endfor
```

then P can be taken outside the loop for a hash table.

```plaintext
HASH←P∘{(↓⍺)↓⍵}
:For I :in ⌽LOOP
    B←(1↑ρP)≥HASH s
:Endfor
```
Memory Requirement of Membership of Char Matrix $s$ in Char Matrix $P$ 

$$1000 \cdot 6 = \rho P$$
Elapsed Time of Membership of Char Matrix $s$ in Char Matrix $P$

$1000 \ 6 = \rho P$

$(\downarrow s) \in \downarrow P$

$(1 \uparrow \rho P) \geq P\{ (\downarrow \alpha) \downarrow \omega \}$

Hash $(\downarrow s) \in \downarrow P$

Hash $(1 \uparrow \rho P) \geq P\{ (\downarrow \alpha) \downarrow \omega \}$
Precaution

The P{((↓⍺)↓⍵)}s Idiom is optimized for character matrix and integer matrix arguments.

This idiom is many times less efficient if your matrix arguments have floating point numbers.

In such cases you may wish to explore algorithms more efficient than this Idiom to substitute for the expression \( \lor/s^.=\top P \)
Recommendation

In your APL function libraries, whenever you see

\[ \vee / s \land . = \emptyset P \]

or

\[ (\downarrow s) \epsilon \downarrow P \]

if the character matrices are large, replace them by

\[ (1 \uparrow \rho P) \geq P \{ (\downarrow \alpha) \iota \downarrow \omega \} s \]

and you will see much faster execution speed with less workspace requirement.
Recommendation

Furthermore, if the Matrix Epsilon is inside a loop, and if the PRINCIPAL character matrix P is constant, then replace

:For I :In iLOOP
    B←(1↑pP)≥P{((↓⍺)↓⍵)s
:Endfor

by

HASH←P∘{((↓⍺)↓⍵}
:For I :In iLOOP
    B←(1↑pP)≥HASH s
:Endfor
Another Situation

Up to now, we are concerned with \( \vee/s^\ast=\emptyset P \)

but if you see the expression \( \lor A^\ast=\emptyset B \)

you can do this optimization:

\[
\lor A^\ast = \emptyset B
\]
\[
\downarrow
\]

\[
\lor B^\ast = \emptyset A
\]
\[
\downarrow
\]

\[
(\downarrow B) \in \downarrow A
\]
\[
\downarrow
\]

\[
(1\uparrow \rho A) \geq (\downarrow A) \downarrow B
\]
\[
\downarrow
\]

\[
(1\uparrow \rho A) \geq A \{ (\downarrow \alpha) \downarrow \omega \} B
\]

And see the previous page in case the hash table approach is applicable.
Topic # 2
Arithmetic Optimization
Divide vs Multiply

M ← 5000 10ρ0.1 + 150000

:For I :In 1 to 10000
  {}M ÷ 100
:EndFor

24492 ms

:For I :In 1 to 10000
  {}M × 0.01
:EndFor

901 ms
Readability of Constants

:For I :In 10000
   {}M ÷ 7
:EndFor

:For I :In 10000
   {}M × 0.1428571429
:EndFor

R7 ← ÷ 7
:For I :In 10000
   {}M × R7
:EndFor
In APL, the Reciprocal function is usually faster than the Divide function.

```
:For I :In 1 to 1000000
  {}1÷7
  {}÷7
:EndFor
```

A 1234 ms
A 1051 ms
Since 1 ÷ Array is slower than ÷ Array should we change 0-Array to -Array?

M ← 10000 10ρ0.1+⍳100000
:For I :In ⍳1000000
  {}0-M
  {}-M
:EndFor

⍝ 110814 ms
⍝ 8546 ms
An APL interpreter can sometimes make use of extra help from an APL programmer to make division and subtraction execute faster.

But should the APL programmer make the function arguments conformable to further help the APL interpreter?

e.g. Should $M \times 0.01$ be changed to $M \times (pM)p0.01$
M ← 5000 10ρ0.1+ι50000

:For I :In 10000
   {}M ÷ 100
:EndFor

A 24570 ms

:For I :In 10000
   {}M × 0.01
:EndFor

A 653 ms

:For I :In 10000
   {}M × (ρM)ρ0.01
:EndFor

A 1545 ms
Topic #3. Scalar Extension
M ← 5000 10ρ0.1 + i50000  A Matrix M
S ← 30  A Scalar S

:For I :In 10000
{ } S × M × 0.01
:EndFor  A 1566 ms

:For I :In 10000
{ } M × S × 0.01
:EndFor  A 773 ms
Order of Execution for Scalars

Scalar1 × Array × Scalar2

can be optimized as

Array × Scalar1 × Scalar2

so that two scalar extensions are reduced to only one scalar extension.
Matrix ∈ Scalar
Matrix = Scalar

Should we use '∈'?  
Or should we use '=='?

M←5000 10p0.1+ι50000

:For I :In ⍳10000
  {}M∈1000.1   A 10207 ms
  {}M=1000.1    A  4385 ms
:EndFor

Because of scalar extension,
M=1000.1 is faster than M∈1000.1
Encoding & Decoding Characters

In a ⍝IO=1 environment

26⊥¯1+Ⓐ大幅提升’DYALOG’
46619358

The ¯1+ is performed 6 times because of scalar extension.

⍲A[1+(6⍪26)⊤4619358]
DYALOG

The 1+ is performed 6 times because of scalar extension.
Avoid Scalar Extension

{⎕IO←0 ◊ 26⊥A⌽ω}'DYALOG'

Now there is no need for scalar extension of -1+

{⎕IO←0 ◊ A[(6ρ26)Tω]}46619358

DYALOG

Now there is no need for scalar extension of 1+
Advantage of Localizing System Variable in D-fn

When an error occurs, thanks to D-fn, ⎕IO bounces back to 1

```
SYNTAX ERROR
{⎕IO←0 ◊ 26⊥⎕Aνω}=^
⎕IO
1
```
Recommendation

Scalar extension has been implemented very efficiently in APL. Whenever there is a chance, let APL do the scalar extension for you. Do not reshape the scalar to help the APL interpreter.

If an expression has a combination of scalars and arrays, let APL work with scalars first, then do scalar extension later.
Scalar Array

When you enclose a numeric array to make it a scalar, this scalar array can take advantage of scalar extension to retrieve items from a nested array efficiently.

This is best illustrated by examples.
A 2 by 3 Nested Matrix Where Each Item Is A 3 by 3 Numeric Matrix

DISPLAY A

\[
\begin{array}{ccc}
111 & 112 & 113 \\
112 & 112 & 113 \\
113 & 113 & 113 \\
\hline
211 & 212 & 213 \\
212 & 212 & 213 \\
213 & 213 & 213 \\
\hline
\end{array}
\]

\[\varepsilon\]
A[((1 1)(2 2)) ((2 3)(2 2))]
To select the red items, you can use
A[2 3p((1 1)(2 2)) ((1 2)(2 2)) ((1 3)(2 2)) ((2 1)(2 2))
((2 2)(2 2)) ((2 3)(2 2))] ~ A 8.1 sec 1,000,000 loops

or (⊂2 2)¨A ~ A Squad Indexing 6.7 sec 1,000,000 loops
To select the red items, you can use

2↑[2]¨2↑[1]¨A  \ A 13.3 sec (looping 1,000 times)

or

(<2 2)↑¨A  \ A  4.9 sec (looping 1,000 times)
To select the red items, you can use

\[-1↑[2]¨(⊂-2 2)↑¨A \quad \text{A 10 sec (looping 1,000 times)}\]

or

\[(⊂(2 3)(,2))⌷¨A \quad \text{A 7 sec (looping 1,000 times)}\]
To select the red items, you can use
(1↑[1]··A),[1]··-1↑[1]··A 17.4 sec (looping 1,000 times)
or
(∈1 3)··[1]··A 7.5 sec (looping 1,000 times)
Recommendation

When you need to uniformly index a matrix consisting of nested vectors as items, squad indexing with the left argument enclosed as a scalar can be a good alternative.

The scalar extension could simplify the expression and speed up the execution time.
APL Optimization Techniques
For Commercial Applications

Questions?

Eugene Ying
2013