PROGRAMMING FOR PERFORMANCE IN APL\3000

Dave Elliott
Development Engineer
General Systems Division
Hewlett-Packard Company

INTRODUCTION

The dynamic incremental compiler of APL\3000 uses several advanced techniques for optimizing the evaluation of APL expressions. These include the sharing of data by multiple variables, the evaluation techniques of 'dragalong' and 'beating' and the use of arithmetic progression vectors. A discussion of each of these features will be presented. The APL\3000 user can also enhance performance by proper tuning of the virtual memory paging scheme using the system variable []VM. A number of general and specific internal optimizations have been introduced starting with version 01.03. Programming techniques to take advantage of these improvements will be presented. There have also been several significant enhancements including six new system functions and two system commands. Use of these features will be discussed.

COMPILER OPTIMIZATIONS

The APL\3000 subsystem was designed to minimize the run-time accessing, movement and creation of data items. This was done by having variables share their data areas when possible, decreasing the number of temporary variables created during expression evaluation, and examining each APL operation within the context of the surrounding expression. Also, those APL operators whose execution change not the value of variables, but the order in which variables are accessed, are implemented as operations on the data descriptors rather than manipulating the actual data. This last technique is referred to as 'beating'.

Beating and Arithmetic Progression Vectors (APVs)

There are five operators in APL that are eligible for implementation by beating. They are take, drop, reversal, transposition and some forms of indexing. Before discussing how beating is done, we must examine how data is stored and accessed in APL\3000. Each variable that contains more than one element is allocated a block of the virtual memory array which contains the data. The first word in this data area is a reference count of how many variables share the area. Data sharing will be discussed later. In the symbol table, which also resides in virtual memory, each variable is rep-

resented by a pointer to its data area and a number of accessing parameters. The system command)SHOW (see New Enhancements section) lets us look at the symbol table entry. Suppose MAT is a newly created 10 by 15 element integer matrix. Then)SHOW MAT would cause the following display:

```
****** MAT ******
TYPE:      MATRIX
REP:       INTEGER
RANK:      2
SHAPE:     10  15
DEL:       15  1
OFFSET:    0
VIRTUAL ADDRESS OF DATA BLOCK:  000000 010137
        BLOCK LENGTH (WORDS) :  301
        NON-SHARED DATA BLOCK
```

The meanings of most of these attributes are obvious. The variable is a matrix, as opposed to a scalar, vector, unit or array. It is an integer item, as opposed to a real, character, or boolean item. It has two dimensions and its shape is 10 by 15. The SHAPE, DEL and OFFSET attributes are the keys to the accessing of data. The del values specify how to step through the data area when accessing more than one element of the variable. There is a del value for each dimension of the variable. In the above example, the first value (15) tells us that if we are looking at a particular element in the matrix's data area, then to look at the element in the next row of the same column, we would have to step over 15 elements to find it. The second value (1) indicates that to move from one element to the next in a particular row, we merely have to look at the next element in the data area. The offset value specifies the point within this data area which contains the first element of the variable.

Reversal is one of the operations that can be beaten in APL\3000. Instead of creating a new data area and copying the elements of the argument one by one in reverse order, we can simply change the del and offset values in the symbol table. Assume that RMAT has been specified as the reversal of MAT. Then)SHOW RMAT produces

```
****** RMAT ******
TYPE:      MATRIX
REP:       INTEGER
RANK:      2
SHAPE:     10  15
DEL:       15  -1
OFFSET:    14
VIRTUAL ADDRESS OF DATA BLOCK:  000000 010137
        BLOCK LENGTH (WORDS) :  301
      DATA BLOCK SHARED WITH:   MAT
```

The only values changed from MAT are the second del value and the offset. Now, when looking at a particular element, we know that to find the next element in that row we decrement the index into the data area by one. The first element of RMAT is at the 14th position in the data area.

The other beaten operations are take (changes only the shape values), drop (changes shape and offset), transposition (changes dels and offset), using indexing to access an entire dimension (rows, columns, planes, etc.) of an array, and indexing using APVs.

An arithmetic progression vector, or APV, is a special data item in APL\3000. Generated by using the monadic iota or index generator, an APV assumes the existence of an data area containing all the integers in ascending order. The symbol table entry for an APV looks just like that for a regular vector except that there is no data block address. The values of an APV can be generated directly from the del, offset and shape values. The offset of an APV is the value of the first element in that vector. For example, assume AVEC is created as '3 plus 5 times iota 10'. The symbol table entry for AVEC would be

```
****** AVEC ******
TYPE:      VECTOR
REP:       APV
RANK:      1
SHAPE:     10
DEL:       5
OFFSET:    8
```

Thus, the first element of AVEC is 8 (the offset) and each successive element can be generated by incremented the previous one by 5 (the del value). When using an APV as an index into another vector, the new vector simply shares the data area of the old vector and uses the data descriptor of the APV.

Dragalong

The process of deferring evaluation of operations in an APL statement as long as possible is called dragalong. The advantages include minimizing the number of temporary variables and avoiding unnecessary computations.

The most basic example of dragalong occurs when several variables of identical shape are operated on by dyadic arithmetic operators. Assume that A, B and C are 3 by 5 integer matrices. Then in the expression A + B - C, an interpreter using dragalong would not evaluate the B - C until it had examined what was to the left of B. In this case, the operation of adding A can be included in the loop that subtracts C from B, thus avoiding the use of a temporary variable.

Often dragalong can be used to avoid unnecessary computation. Assume that X and Y are 1000-element vectors. The

expression 3 ^ X "DV Y (3 take X divided by Y) would not divide each of the 1000 elements of X by the corresponding elements of Y. Instead, the interpreter would realize that the user only cares about the first three elements of this result and would therefore only do three divisions. Some controversy about dragalong arises here in the case that one of the 4th through 1000th elements of Y is a zero. Most other APL systems will return a DOMAIN ERROR after attempting to divide by zero. APL\3000 never looks at the 4th through 1000th elements of Y or X, so the expression is evaluated without error unless one of the first three elements of Y is zero.

VIRTUAL MEMORY PARAMETER TUNING

The []VM system function provides the APL\3000 user with the ability to dynamically change the page size and number of pages in the virtual memory algorithms. Because []VM can be a local variable, its value can be tailored to the needs of specific functions. Section X of the APL\3000 Reference Manual contains a thorough explanation of how to set the virtual memory parameters to minimize page faulting and thus maximize performance. The default value for the page size has been changed in version 01.03 to reflect the increase in the average main memory size of our installed systems. The page size in a clear workspace is now 1024 bytes instead of 512 bytes. This will only affect those users who do not explicitly change this parameter.

NEW ENHANCEMENTS

Verify Input and Format Input System Functions

The []VI and []FI system functions work together to convert character data to numeric data. []VI takes as its argument a character vector and returns a boolean vector as its result. The length of the result is equal to the number of non-blank character groups in the argument. For each group of non-blanks, the corresponding element of the result is set to 1 if and only if the non-blank characters form a valid APL numeric constant. []FI takes the same argument as []VI, returns a result of the same length as that of []VI, but the result is numeric. For each group of non-blank characters in the argument, the corresponding element of the result is set to 0 if the group does not form a valid numeric constant. Otherwise, it is set to the constant itself. Therefore, it is similar to the execute operator except that is does not return a syntax error for characters that cannot be converted to numbers. Also, the execute operator has a limit of 8188 numeric constants in its argument. []FI and []VI are limited only by the maximum size of APL\3000 data

items. []VI and []FI are most useful to verify that data entered by a user to a quote-quad input is numeric. If C is a character vector entered by the user, then ([]VI C)/[]FI C will return a numeric vector containing all the valid numeric constants in C.

[]PRINT System Function

The []PRINT system function provides a quick and easy way to send APL data to the line printer or to a disc file from within an APL session or function. []PRINT takes as its right argument any APL variable or expression that has a value. The optional left argument is a character vector representing the termtype of the receiving file or device. The default value is 'ASCII'. []PRINT takes the right argument, formats it in the same manner as monadic format, and sends the result to a file whose formal designator is APLLP. This creates a spool file which is printed immediately. Suppose that you wanted each successive call to []PRINT to append its results to a disc file. The following MPE commands, entered before invoking APL, or during the APL session using the)MPE system command or shared variables, would produce the desired results:

```
:BUILD filename;REC=recsize,blockfactor,F,ASCII
:FILE APLLP=filename,OLD;DEV=DISC;ACC=APPEND
```

It is important to note that []PRINT will not return an error message even if you are attempting to append records past the end of the disc file. Before invoking []PRINT, it is advisable to set the value of []PW (print width) to the record size of the APLLP file.

Character String System Functions

In order to simplify and optimize the manipulation of character strings in APL\3000 and thereby increase its suitability in a commercial environment, three new system functions have been implemented.

[]CSLOC - Character String Locator

The left argument of []CSLOC is a character scalar or vector. The right argument is a character scalar, vector, or matrix. If the right argument is a scalar or vector, then the result of []CSLOC is an integer vector containing the starting indices of each occurrence of the left argument in the right. If the right argument is a matrix, then the result is an integer vector containing the row numbers for which the left argument appears, left-justified, in the right. In either case, an empty vector is returned if there

are no occurrences of the left argument in the right. []CSLOC is particularly useful in table searching and text processing.

[]CSMOD - Character String Modifier

This system function takes as its right argument a character vector to be modified. The left argument is a character vector of the form

<delimiter> <string1> <delimiter> <string2> <delimiter>.

Execution of []CSMOD causes all occurrences of string1 in the right argument to be replaced by string2. The delimiter may be any character. The modified character vector is returned as the result.

Examples:

```
      '/COW/BUFFALO/' []CSMOD 'HOW NOW, BROWN COW?'
HOW NOW, BROWN BUFFALO?

      '*BE**' []CSMOD 'TO BE OR NOT TO BE'
TO  OR NOT TO
```

The second example demonstrates that []CSMOD can be used to delete occurrences of one string within another.

[]CSD - Character String Delimiter

The []CSD system function takes a character vector as its right argument and a character scalar (the delimiter) as its left. The explicit result is a character matrix formed by scanning the right argument (left-to-right) and starting a new row of the result at each occurrence of the delimiter character. Consecutive, leading, or trailing delimiters cause empty rows to be formed. The delimiter characters do not appear in the result. If there are no occurrences of the delimiter in the right argument, then a one-row matrix is returned.

)SHOW System Command

This system command allows the user to see how variables are stored in the workspace. The command is optionally followed by a list of variable names. For each name, a display of the symbol table information for that name is produced. For example:

```
      )SHOW XXX
```

```
***** XXX *****
TYPE:      MATRIX
REP:       BYTE
RANK:      2
SHAPE:     21  6
DEL:       6  1
OFFSET:    0
VIRTUAL ADDRESS OF DATA BLOCK:  000000 005716
        BLOCK LENGTH (WORDS) :  64
NON-SHARED DATA BLOCK
```

If the data block for the requested variable is shared, then the names of the variables sharing the block are displayed. If no list of names follows the)SHOW command, then all the variables in the workspace are displayed.

)* System Command

This command speeds up the process of correcting functions. Its invocation causes the APL editor to be entered with the function currently on the top of the state indicator stack. The line containing the last error is displayed and the editor is placed in modify mode.

NEW OPTIMIZATIONS

It has been recognized that several of the internal algorithms for operators in APL\3000 could be optimized considerably. The most serious offenders have been dyadic iota (indexing) and dyadic epsilon (membership). It was also felt that the grading algorithm needed some improvement. These optimizations have been implemented in APL\3000 starting with version 01.03. Some knowledge of how the improvements were made will probably help the user to take advantage of them.

Dyadic Epsilon (Membership)

In the expression M "EP N ("EP is the ASCII representation of epsilon), assume that m is the length of M and n is the length of N. The old algorithm took the brute-force approach of taking each individual element of M and comparing it to each element of N until either a match was found or there were no more elements in N. This was particularly inefficient if there were several identical elements in M or if M had considerably more unique elements than N. This brute-force method resulted in m passes through N and an average of (m x n)/2 comparisons.

The information gathered during the repeated passes through N can, in fact, be gathered in only one pass if a

table is set up to indicate what values are in N. The size of this table is determined by the range of values in N. If N is character data, then the table need only be as long as []AV (256 elements). Other data types present more of a problem. The number of possible values in real data is not only infinite but, since real numbers can be irrational, we could not possibly make a one-to-one correspondence between real numbers and discrete table elements. The range for integers in APL\3000 is over four billion, therefore we can use a table only if the values in N are relatively close together. The new algorithm for dyadic epsilon uses a table of 4096 bits packed into a logical array of length 256. This array is on the stack and not in virtual memory. An initial pass is made through N to determine the minimum and maximum elements and therefore the range. If the range is not greater than 4096, then a second pass is made through N to set the appropriate bits in the table. Finally, the elements of M are examined. If an element of M is outside the predetermined subrange of values for N, then, of course, it cannot be a member of N and the corresponding element of the result is set to 0. Otherwise, the element of the result is set to the value of the table bit corresponding to that element of M. This algorithm requires only 2 passes though N and, after the table is set up, exactly m comparisons.

The performance improvements gained by the implementation of this method have been substantial. For example, a dyadic epsilon on two 500-element integer vectors which formerly took 34.38 CPU seconds now takes only 0.23 CPU seconds. Using two 250-element character vectors, the execution time has been cut from 8.11 CPU seconds to only 0.04 CPU seconds.

In summation, CPU times for dyadic epsilon will be reduced up to 99% if the data is character, or if both arguments are integer and the range of the right argument is not greater than 4096. Even if these restrictions are not met, improvements in the way the brute-force method accesses virtual memory will result in up to a 60% improvement over previous versions of APL\3000.

Dyadic Iota (Indexing)

This algorithm also took the brute-force approach. Each element of the right argument was individually checked against each element of the left. If the element was found, then its position in the left arguments was returned as the result.

The table method outlined above could not be used for integer data. The table would not only have to indicate the presence or absence of elements of the right argument, but would also have to indicate the position of each element. Therefore, each table entry would have to be a double word and the table would have to be impractically small to fit on the stack. The only exception is in character data for

which we can have a table of 256 double word entries.

Using the table approach, CPU times for dyadic iota on character arguments have been reduced up to 99%. Again, the brute-force method has also been optimized providing an improvement of about 60-65% on all other types of arguments.

Grading

When investigating grading algorithms for APL, there are three important considerations. First, by definition a grade in APL must be a stable grade. This means that the relative order of equal elements of a vector must be maintained. For example, if A is the vector 6 9 4 9 5 2 , the the correct result for "GU A ("GU is the ASCII representation for 'grade up') would be 6 3 5 1 2 4 , even though 6 3 5 1 4 2 would give the same value for A["GU A]. This requirement for stability eliminates some of the faster sorting and grading algorithms. The second consideration is that during grading, the data is never moved. All swapping is done on elements of an index vector through which the actual data is accessed. This index vector becomes the result of the grade operation. This makes each data reference indirect and therefore very costly. The third factor in APL grading is that in many cases the argument to the grade operator is already sorted or nearly sorted. For example, a program may catenate a few elements to a long sorted vector and then use the grade operator to properly position those new elements.

The algorithm implemented in version 01.02 is the natural list-merge sort. This method meets the stability requirement and is also very fast for sorted, nearly sorted, or reversed data. Additional optimization was done for version 01.03 involving the accessing of virtual memory.

Meaningful CPU time comparisons are difficult to produce but it is estimated that the performance of the grade operator has been improved by about 40% for random data and about 75% for nearly sorted data between versions 01.01 and 01.03.

Miscellaneous Optimizations

Reshape on Character Data -

Initialization of a 100 by 100 element character matrix to blanks has been improved from 9.32 CPU seconds to 0.15 CPU seconds.

Monadic and Default Format on Integers -

Improved about 30%.

Absolute Value -

Improved about 20%.

CONCLUSION

The fact that APL is such a full and powerful language can often cause the user to forget about the actual internal algorithms used in its implementation. Rather than presenting APL\3000 as a 'black box' that operates in exotic and mysterious ways, we choose to let the user know as much as necessary about its methods and techniques. This will enable the user to program knowledgably and efficiently.

The efforts to enhance and optimize the APL\3000 system are continuing. Our most valuable resources for these efforts are the suggestions and needs of our users.