An Algorithm for Generating an *m*-ary Summation Tree

M. Sievers
Communications Systems Research Section

An algorithm is presented for generating an m-ary summation tree. The algorithm is completely general and may be applied to any length input string. For an N length sequence summed in groups of m_ℓ at each level ℓ, a maximum of 3L - 2 storage is required where

$$\prod_{\ell=1}^{L} m_\ell = N$$

A special case of the general m-ary tree where all m_ℓ are equal will be used to smooth data in a radio-frequency interference experiment. The maximum storage required when $m_\ell = m$ for all ℓ reduces to the closed form $3 \log_m N - 2$.

I. Introduction

A study is in development that will determine the magnitude and nature of radio-frequency interference (RFI) in the vicinity of the Goldstone tracking complex. Spectra collected in the RFI experiments will be smoothed by a binary summation tree. This tree is a special case of the more general m-ary summation tree.

An m-ary summation tree is the tree formed by initially dividing an input string of length N into groups of length m_1 each. There will be $g_1 = \lfloor N/m_1 \rfloor$ groups. The notation $\lfloor X \rfloor$ denotes "the integer part of X." The m_1 elements in each group are summed to form a group sum. The resulting group sums form the first level in the tree.

The g_1 first level group sums are added in m_2-tuples to form the group sums of the second level. Clearly, the number of groups at the second level will be $\lfloor g_1/m_2 \rfloor$. In general, the number of groups at level L will be $\lfloor g_{L-1}/m_L \rfloor$. The process is shown pictorially in Fig. 1 for $N = 16$, $m_1 = 2$, $m_2 = 4$, and $m_3 = 2$.

M_ℓ-tuple additions are performed until either a single result is formed representing the sum of all inputs, or the tree is incomplete. An incomplete tree is one in which at some level ℓ, g_ℓ groups are present where g_ℓ modulo m_ℓ is not zero. If the tree is incomplete, the final sum (sum of all inputs) is not formed since there will be one or more levels in which it was not possible to form an m_ℓ group.

In order for the final sum to be calculated, it must be possible to form m_ℓ groups at each level in the tree. This implies that there must be an L such that

$$\prod_{\ell=1}^{L} m_\ell = N \tag{1}$$

The value of L is the depth of the tree.

In the remainder of this article, only complete trees will be considered. Incomplete trees are of no practical value in the RFI experiments.

It is worth noting that in generating the m-ary summation tree, at most $N - 1$ additions are required. This is equal to the minimum number of additions needed to add a string of N numbers. The tree structure, therefore, does not add overhead to the number of additions required to sum the N length input string.

For small N it might be reasonable to store each input word and summation result in memory. However, for large N this is not practical even if the summations are done "in place." The algorithm presented in the next section requires only $3L - 2$ storage, where L is the depth of the m-ary tree.

II. *m*-ary Summation Tree Algorithm

The m-ary summation tree algorithm presented below makes the following assumptions:

(1) Input data are available serially.

(2) An accumulator is present for holding partial sums.

(3) When a group sum is completed it is immediately output.

(4) Once a level ℓ sum is formed, the level $\ell - 1$ addends from which it was formed are no longer needed.

It is easy to show that only L data storage is necessary to generate the summation tree. To start, it is clear that only one storage word is needed to generate the first level in the tree. This is because once the first group sum is completed in the accumulator, it is moved to the storage location. The accumulator is then free to accumulate the next group sum. After this second group sum is available, it is added to memory and the result placed back in memory.

A second level group sum is produced when first level storage holds the sum of $m_2 - 1$ first level group sums and the accumulator contains an additional first level group sum. The level two group sum is obtained by adding the accumulator to level one storage. The first level storage is then free to collect the next $m_2 - 1$ first level groups sums.

The level two group sum is left in the accumulator after its formation. This situation is identical to the situation in the first level when a first level group sum is held in the accumulator. Therefore, it follows that only one storage element will be needed at the second level.

In general, any time m_ℓ group sums are available at the $\ell - 1^{st}$ level ($m_\ell - 1$ held in $\ell - 1$ memory and one in the accumulator), these are added to form a level ℓ group sum. This sum is left in the accumulator, and again the situation reduces to the level one situation. Therefore, only one storage element is needed at each level in the tree, except the last.

No storage is needed at level L, because when the level L sum is formed in the accumulator, it is immediately output. Since there are L levels, there will be need of $L - 1$ storage locations in the tree. Including the accumulator, a total of L data storage is required.

An algorithm has been designed to implement the recursion described above. Figure 2 shows the flow chart of the algorithm.

The algorithm makes use of group counters and level counters. Group counters keep a count of the number of group sums stored in a given level storage. Level counters count the number of times a word of a given level has been formed. Since there will be $L - 1$ storage cells, and since there is no need to count the number of level L sums formed (this count will be zero until the final sum is available at which time it becomes one), $L - 1$ group and level counters are necessary.

The counters are used to keep track of which original input elements have been added to form a given level sum. Labeling the original inputs, 1, 2, 3, . . . N, then the c^{th} sum formed at the j^{th} level will be the sum:

$$S = \sum_{i=1+(c-1)r_j}^{cr_j} \tag{2}$$

where

$$r_j = \prod_{\ell=1}^{j} m_\ell$$

The vector of group counters (GC) can be used to indicate the total number of inputs processed. GC is a multiple radix number where the radix vector R is composed of the elements r_j. If each group counter gc_ℓ is interpreted as a radix r_ℓ digit, then the number of inputs processed at any time is:

$$\sum_{\ell=1}^{L} r_\ell \, gc_\ell \tag{3}$$

The total storage required for this algorithm is the sum of the storage needed to generate the tree plus the number of counters. From the discussion in this section, L storage is needed to generate the tree, and $2(L - 1)$ counters are required from Eq. (1). This total equals $3L - 2$ where

$$\prod_{\ell=1}^{L} m_\ell = N \tag{4}$$

III. Special Case: $m_\ell = m$ for all ℓ

The case when all level group sizes are the same is a special case of the general m-ary tree that greatly simplifies Eqs. (1) to (4). This case is also much simpler to implement than the general case. This makes it more attractive to build hardware for as will be done in the RFI experiments for $m = 2$.

Formation of m size groups at each level leads to the closed form

$$L = \log_m N \tag{5}$$

for Eq. 1. Since in this case $r_i = m^i$, Eq. (2) simplies to

$$\sum_{i=1+(c-1)m^j}^{cm^j} \tag{6}$$

for the c^{th} sum at the j^{th} level. Eq. (3) for the total number of inputs processed reduces to

$$\sum_{\ell=1}^{\log_m N-1} m^\ell gc_\ell \tag{7}$$

Finally, Eq. (4), the total storage required, becomes

$$3 \log_m N - 2 \tag{8}$$

IV. Sample Program

A portion of a FORTRAN program that implements the algorithm of Fig. 2 along with sample output data is shown in Fig. 3. The program was written for the special case where all $m_\ell = 2$. The input length N is 256 with input (1) = 1, input (2) = 2, etc. Vector IDATA is the input data vector, IGC is the group count vector, ICNT is the level count vector, and M is the group size. IGC, ICNT and M have been initialized to 0, 0, and 2, respectively, outside the portion of the program segment shown.

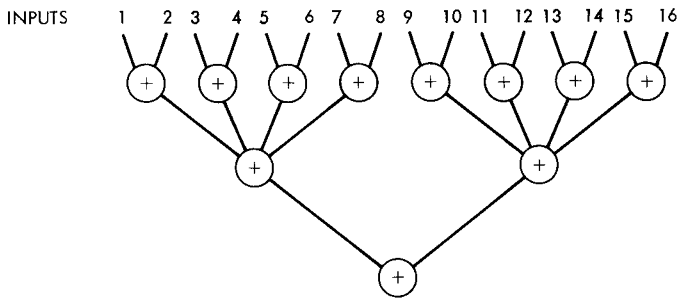

Fig. 1. *M*-ary summary tree for $N = 16$, $M_1 = 2$, $M_2 = 4$, and $M_3 = 2$

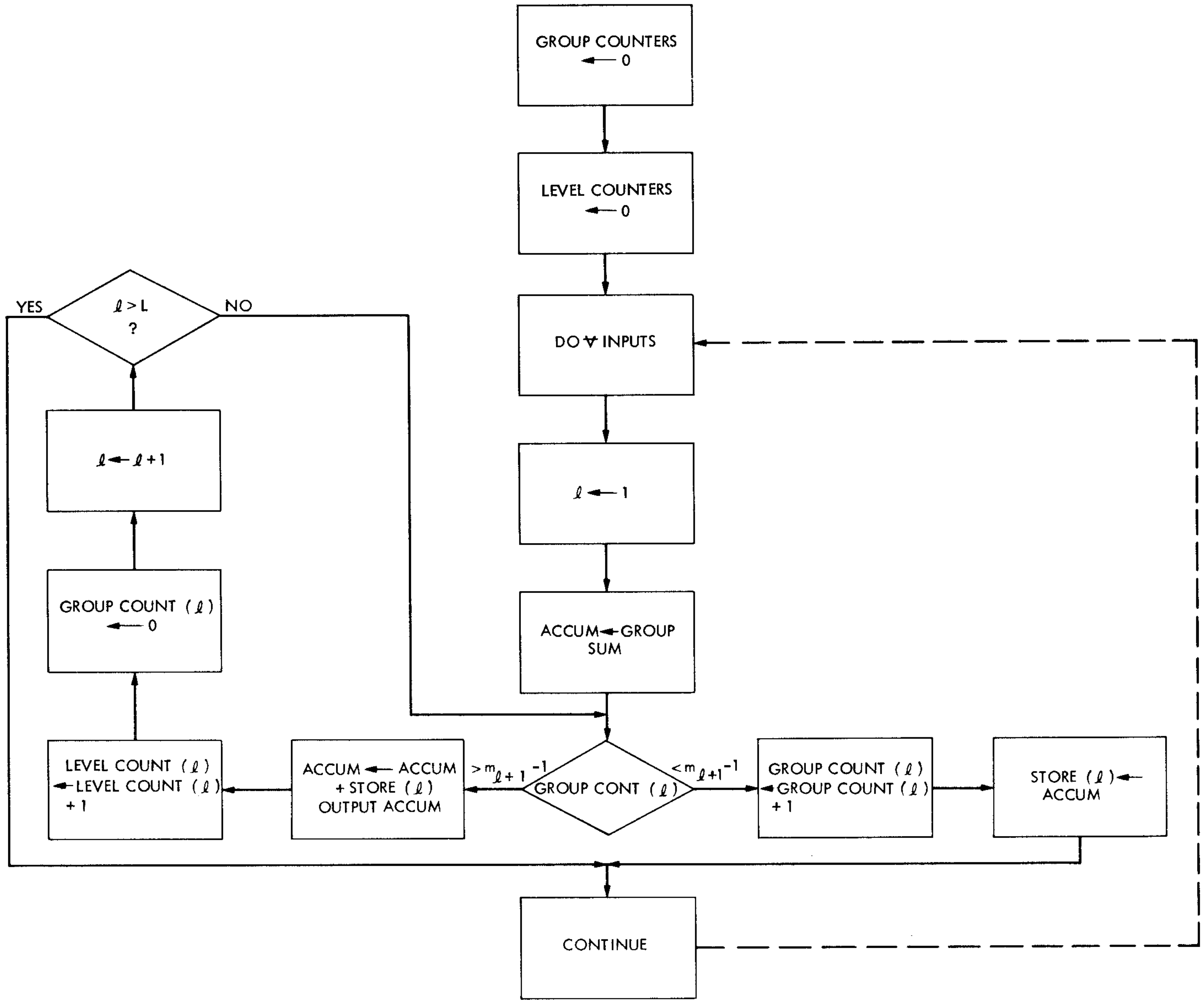

Fig. 2. Algorithm

```
            ISTOP = 256-M + 1
            DO 30 J = 1, ISTOP, M
            L = 1
            IACCUM = 0
            INDEX = J + M -1
            DO 25 JJ = J, INDEX
            WRITE (3,60)JJ, IDATA (JJ)
            IACCUM = IACCUM + IDATA (JJ)
   25       CONTINUE
            WRITE (3,65) IACCUM
   50       IF (IGC(L). LT. M-1) GO TO 40
            IACCUM = IACCUM + ISTORE (L)
            ICNT (L) = ICNT (L) +1
            IBEG = M** (L + 1)* (ICNT (L) ,1) + 1
            IEND = M** (L + 1)* ICNT (L)
            WRITE (3,80) IBEG, IEND, IACCUM, ICNT (L), L
            IGC (L) = 0
            ISTORE (L) = 0
            L = L + 1
            IF (L .EQ. 9) GO TO 30
            GO TO 50
   40       IGC (L) = IGC (L) + 1
            ISTORE (L) = IACCUM + ISTORE (L)
   30       CONTINUE
```

```
INPUT ( 1 )         1
INPUT ( 2 )         2
SUM OF INPUTS       3

INPUT ( 3 )         3
INPUT ( 4 )         4
SUM OF INPUTS       7

             SUM FROM 1 to 4 = 10

INP UT ( 5 )        5
INPUT ( 6 )         6
SUM OF INPUTS      11

INPUT ( 7 )         7
INPUT ( 8 )         8
SUM OF INPUTS      15

             SUM FROM 5 TO 8 = 26

             SUM FROM 1 TO 8 = 36
```

Fig. 3. FORTRAN Program sample output, *M* = 2, *N* = 256