Proceedings of the 2006 WSEAS International Conference on Mathematical Biology and Ecology, Miami, Florida, USA, January 18-20, 2006 (pp146-153)

Comparing DNA Sequences By Dynamic Programming In Sequential And Parallel Computer Environments

ERIC N.D. NGUYEN^', DON N. NGUYEN^", DUC T. NGUYEN'''^^, AND SIROJ TUNGKAHOTARA'''

^Biology Department
Virginia Commonwealth University
Richmond, VA 23284
(USA)
^^Correspondent Author
'''Civil & Environmental Engineering Department
Old Dominion University
1319 ECSB
Norfolk, VA 23529
(USA)

Abstract: - Comparing two sequences by using dynamic programming algorithms is studied. Both serial and (multiple processor) parallel computer algorithms are discussed. Numerical performance of the developed software is validated through small to large-scale applications. Results (based upon comparing 2 large sequences with 40,000 and 36,000 character length, respectively, and using 2-24 parallel processors) indicate that the developed software is reliable and highly efficient.

Key-words: Sequences, alignments, DNA, dynamic programming, parallel algorithms, Fortran-90, molecular biology, MPI

1 Introduction

Due to large-scale data manipulation required in the general areas of computational biology [1-6], and especially with the availability of modern, inexpensive high-performance computers (which have multiple processors) [7], larger problems' sizes in molecular biology can now be more efficiently analyzed, and to speed-up the solution process.

In this paper, the problem of comparing two DNA sequences using the basic version of dynamic programming algorithm is specifically considered. To facilitate the discussions, consider the following 2 DNA sequences:

GACGGATTAG and GATCGGAATAG.

The similarity between these 2 sequences can be even more obvious when they are aligned on top of each other, as following:

```
G A - C G G A T T A G
G A T C G G A A T A G        (1)
```

It should be noted here that the lengths of the above 2 sequences are NOT the same. For this reason, a space (indicated by a dash) is inserted in (1), to assure these 2 sequences to have the same length. Thus, one defines an alignment as the insertion of space(s) in arbitrary location(s) along the sequences so that they will end up with the same size. The augmented sequences can then be placed on top of each other, creating a one-to-one correspondence between characters and/or spaces among these sequences. However, one

requires that no space in one sequence be aligned with a space in the other.

Our main objective here is to describe efficient serial and parallel algorithms that will take 2 sequences and determine the best alignment, as it has been done in (1). To achieve this goal, one needs to assign a "scoring" system, as following:

Each column of the alignments (between the 2 sequences) will receive a "certain value" depending on its contents, and the "total score" will be the sum of the values assigned to its columns. If one adopts the policies that "+1" is assigned to a column which has 2 "identical" characters, "-1" is assigned to a "mismatch" case, and "-2" is assigned to a column which has a "blank space" (indicated by a symbol "dash"), then the best alignment will be the one with a maximum total score. This maximum score will be called the "similarity" between the 2 sequences, and will be denoted as similar(s,t), for sequences s and t. In practical cases, there may be several alignments with the same maximum score.

For the alignment shown in (1), there is 1 column with a "blank space", 1 column with a mismatch character, and 9 columns with identical characters. Thus, a total score can be computed as:

1 * (-2) + 1 * (-1) + 9 * (+1) = 6 (refer to the Highest Total Score, shown in the authors' computer output (Table 1), which is also matched with the results in Ref.[1]).

The particular choice of scores "+1, -1, -2" has often been used in practice. It is based upon rewarding for "matching characters" case, and penalizing for "mismatching characters", or "a column with a blank space" cases.

With the above paragraphs as backgrounds, the objective of this study is to re-visit an efficient Dynamic Programming algorithm for computing the similarity between 2 given sequences, and to propose a parallel computation procedure to improve its speed for solving even larger-scale problems. Basic reviews of the "serial" version of the dynamic programming algorithm is summarized in Section 2. Parallel computational procedures for the Dynamic Programming algorithm are explained in Section 3. Validation for the "serial" computer software is conducted in Section 4. Finally, conclusions are drawn in Section 5. For readers' convenience, the entire serial (FORTRAN-90) source code, including Input/Output data files are listed in the Appendix.

2 Brief Reviews of Dynamic Programming Algorithms [1-6]

One possible (but highly inefficient) approach for computing the similarity between 2 sequences would be to generate all possible alignments, the total score for each case is computed, and the best score is selected. However, the number of possible alignments between 2 sequences can be exponential (especially for the cases where the lengths of the 2 sequences are not only long, but also significantly different), which makes this "brute force" approach to be impractical!

In the following section, a more efficient way for computing the similarity between 2 sequences is briefly reviewed. This algorithm is called "dynamic programming", which basically solves an instance of a problem using the already computed solutions for smaller instances. Given 2 sequences s and t, the solution can be built up by determining all similarities between arbitrary prefixes of the 2 given sequences. One starts with shorter prefixes and used previously computed results to solve the problem with larger prefixes.

Let m, and n represents the sizes of 2 sequences s, and t, respectively. There are (m+1), and (n+1) possible prefixes of s, and t, respectively, including the empty string. Therefore, one may arrange the calculation in a 2-dimensional matrix (m+1) x (n+1) array, where the entry (i,j) represents the similarity between s(1 ... i), and t(1 ... j)

Eq.(4) shows a 2-dimensional array [M] corresponding to the 2 given sequences:

s = AAAC, and t = AGC (2)

In this specific example, since the 2 sequences "s" and "t" have 4, and 3-character length, respectively, hence, there are only 4 possible ways for aligning these 2 sequences:

A A A C A A A C A A A C A A A C
- A G C A - G C A G - C A G C -

Using the same scoring convention as used in Section 1, the final scores corresponding to the above possible alignments are -1, -1, -1, and -3, respectively. Thus, the highest total score in this particular example is -1 (please see the value of M(4,3), shown in Eq. 4).

One places the sequences "s", and "t" along the rows, and columns of matrix [M], respectively. This arrangement will indicate the prefixes more clearly. It is noted that the 0-th row (and column) of [M] are initialized with multiples of the "blank space" penalty (-2 is used here). This is because there is only one alignment possible if one of the sequences is empty. In this case, one just adds as many spaces as there are characters in the other sequence. The score of this alignment is -2k, where k is the length of the nonempty sequence. Thus, initializing the values for the 0-th row (and column) of [M] is a trivial task.

To compute the value of a general entry M(i,j), one just needs to look at its 3 neighboring entries: M(i-1,j), M(i-1,j-1), and M(i,j-1). Thus, one can visualize that the entry M(i,j) is located at the bottom right corner of an imaginary square, with its 3 neighboring entries occupy at the other 3 corners this imaginary square! The reason for this observation is there are just three ways for obtaining an alignment between s(1...i) and t(1...j), and each one uses 1 of these previous values. The following 3 possible choices are listed here[1]:

* Align s(1...i) with t(1...j-1) and match a space with t(j), or
* Align s(1...i-1) with t(1...j-1) and match s(i) with t(j), or
* Align s(1...i-1) with t(1...j) and match s(i) with a space.

These possibilities are exhaustive because we are not allowed to have 2 spaces paired in the last column of the alignment. Scores of the best alignments between smaller prefixes have already been stored in the array if one chooses an appropriate order for which the entries are computed. As a consequence, the similarity sought can be computed by the formula:

$$\text{sim}[s(1\ldots i), t(1\ldots j)] = \max \begin{cases} \text{sim}[s(1...i), t(1...j-1)]-2 \\ \text{sim}[s(1...i-1), t(1 \ldots j-1)]+p(i,j) \\ \text{sim}[s(1...i-1), t(1...j)]-2 \end{cases} \quad (3)$$

In Eq.(3), p(i,j) = +1, if s(i) . EQUAL. T(j), and
p(i,j)=-1, if s(i) .NOT EQUAL. t(j)

The values of p(i,j) are written inside the parenthesis, shown in Eq.(4)

$$[M] = \begin{array}{c|cccc} & & A & G & C \\ & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & -2 & -4 & -6 \\ \hline A\ 1 & -2 & 1 & -1 & -3 \\ & & (+1) & (-1) & (-1) \\ \hline A\ 2 & -4 & -1 & 0 & -2 \\ & & (+1) & (-1) & (-1) \\ \hline A\ 3 & -6 & -3 & -2 & -1 \\ & & (+1) & (-1) & (-1) \\ \hline C\ 4 & -8 & -5 & -4 & -1 \\ & & (-1) & (-1) & (+1) \end{array} \quad (4)$$

Eq. (4) can be expressed as:

$$M(i,j)=\max \begin{cases} M(i,j-1) - 2 \\ M(i-1,j-1) + p(i,j) \\ M(i-1,j) - 2 \end{cases} \quad (5)$$

The orders of computation for entries of M(i,j) can be proceeded row-wise (or column-wise), or by any other orders. The only requirement is that M(i-1,j), M(i-1,j-1), and M(i,j-1) should be already computed before attempting to compute M(i,j).

3 Parallel Dynamic Programming Algorithms [1,7]

Based upon the example shown in Eq.(4), and the formula given in Eq.(5), and assuming there are 4 processors available for parallel computation purposes, the step-by-step parallel computation procedures can be described for the following 8x8 matrix [M]:

Step 1: The first row/column, or row #0/column #0 of matrix [M] are initialized

Step 2: The entry M(1,1) is computed by processor P1
Step 3: In parallel, M(1,2), M(2,1) are computed by processors P2, P3
Step 4: In parallel, M(1,3), M(2,2), M(3,1) are computed by P4, P1,P2
Step 5: In parallel, M(1,4), M(2,3), M(3,2), M(4,1) are computed by P3, P4,P1,P2
Step 6: In parallel, M(1,5), M(2,4), M(3,3), M(4,2), M(5,1) are computed by P3, P4,P1,P2,P3
Step 7: In parallel, M(1,6), M(2,5), M(3,4), M(4,3), M(5,2), M(6,1) are computed by P4,P1,P2,P3,P4,P1
Step 8: In parallel, M(1,7), M(2,6), M(3,5), M(4,4), M(5,3), M(6,2), M(7,1) are computed by P2,P3,P4,P1,P2,P3,P4
Step 9: In parallel, M(2,7), M(3,6), M(4,5), M(5,4), M(6,3), M(7,2) are computed by P1,P2,P3,P4,P1,P2
Step 10: In parallel, M(3,7), M(4,6), M(5,5), M(6,4), M(7,3) are computed by P3,P4,P1,P2,P3,.. etc ..

The above parallel computational steps can also be conveniently summarized according to the following table:

	0	1	2	3	4	5	6	7
0	0	-2	-4	-6	-8	-10	-12	-14
1	-2	1.1	2.2	3.4	4.3	5.3	6.4	7.2
2	-4	2.3	3.1	4.4	5.4	6.1	7.3	8.1
3	-6	3.2	4.1	5.1	6.2	7.4	8.2	9.3
4	-8	4.2	5.2	6.3	7.1	8.3	9.4	10.4
5	-10	5.3	6.4	7.2	8.4	9.1	10.1	11.4
6	-12	6.1	7.3	8.1	9.2	10.2	11.1	12.3
7	-14	7.4	8.2	9.3	10.3	11.2	12.4	13.1

In the above matrix table, row #0 and column #0 are first initialized. Each entry M(i,j) of the above table (where i=1-7; and j=1-7) consists of 2 numbers, which are separated by the "." symbol. The first number represents the order of tasks, and the second number represents the processor number. Thus, typical entries, such as 5.3, 5.4, 5.1, 5.2 and 5.3 indicate that task order # 5 can be done in parallel by processors # 3, 4, 1, 2 and 3, respectively. This task order #5 can NOT be executed unless task order #4 (such as 4.3, 4.4, 4.1, and 4.2) have already been completed by processors #3, 4, 1 and 2, respectively.

Carefully observing the above table has also revealed that for the computation of the total 49 entries of matrix [M] (excluding the initialized row #0, and column #0), processors P1, P2, P3 and P4 calculates 13, 13, 12 and 11 entries, respectively. Thus, good workload balancing amongst processors can be expected from the suggested parallel strategies!

For practical, large-scale sequence comparisons, the above parallel dynamic programming algorithm can be further improved by using "block" parallel dynamic programming algorithm. The key idea in "block" parallel algorithm is to "increase" the amount of workloads done by each processor. Thus, each entry M(i,j) should be thought as a "block", or as a sub_matrix rather than containing a single value!

Different patterns of allocating the number of processors to different "blocks" are possible, such as (assuming there are 3 processors available: P0, P1, and P2) indicated in Figure 1:

	0	-2	-4	-6	-8	-10	-12
-2	P0	P1	P0	P0	P1	P0	
-4	P2	P1	P1	P2	P1	P0	
-6	P2	P2	P0	P2	P1	P2	
-8	P0	P1	P0	P2	P0	P0	
-10	P2	P1	P0	P1	P1	P0	
-12	P2	P1	P2	P2	P1	P2	

(a)

	0	-2	-4	-6	-8	-10	-12
-2	P0	P0	P0	P0	P0	P0	P0
-4	P1	P1	P1	P1	P1	P1	P1
-6	P2	P2	P2	P2	P2	P2	P2
-8	P0	P0	P0	P0	P0	P0	P0
-10	P1	P1	P1	P1	P1	P1	P1
-12	P2	P2	P2	P2	P2	P2	P2

(b)

Fig. 1: Different Patterns for Allocating Processors to Matrices [p], [M]

If one carefully observes the above 2 possible choices of patterns, then the following conclusions can be made:

(a) Both the above choices (shown in Figures 1a, and 1b) do offer good "work balancing" amongst processors. For example, each processor will execute the same amount of (12) block sub-matrices.
(b) The 2nd choice (shown in Figure 1b) has a "more simple" pattern; hence parallel code implementation should be easier.

(c) Furthermore, the 1-st choice (see Figure 1a) requires BOTH the "last row" and "last column" of a completed block to be sent to adjacent processors. However, the 2nd choice (see Figure 1b) only requires the "last row" to be sent (downward) to its neighboring processor. The "last column" needs NOT be sent (rightward) to its next block, since this column is also owned by the same processor!

The parallel "block computation" procedures discussed in this section will, not only save computational time, but also solve much larger problems since large (integer) matrices [p], and [M] will be distributed and stored in different processors.

4 Numerical Applications

Example 1: A Small-Scale Problem

In this example, the lengths of the 2 sequences {s}, and {t} are 10, and 11 (characters), respectively.

Based upon the discussions presented in the previous sections, the serial FORTRAN-90 computer program has been developed by the authors, and is listed in the Appendix. User's input data for this FORTRAN-90 code is quite simple, and only contains the following information (using Eq. 1, as an example of a small-scale problem):

===> Number of lines (each line can have a maximum of 80 characters) required to input the first sequence {s}. In this particular example (see Eq. 1), one has: 1
===> Input sequence {s}. In this particular example (see Eq. 1), one has: GACGGATTAG
===> Number of lines (each line can have a maximum of 80 characters) required to input the second sequence {t}. In this particular example (see Eq. 1), one has: 1
===> Input sequence {t}. In this particular example (see Eq. 1), one has: GATCGGAATAG

The computer output obtained from the authors' developed FORTRAN-90 code is given in Table 1:

Table 1: Computer Output for a Small-Scale Example

```
==================================
  Authors = Eric+Don+Duc Nguyen, Version
              Date: 10-22-05
==================================
# lines for sequences {s} =  1
sequence {s} =
GACGGATTAG
# lines for sequences {t} =  1
sequence {t} =
GATCGGAATAG
lengths of sequences {s} and {t} =  10 11
Highest Total Score =  6
==================================
```

The above results (see Highest Total Score) does match with the ones given in Ref. [1]

Example 2: A Medium-Scale Problem

In this example, the lengths of the 2 sequences {s}, and {t} are 798, and 720 (characters), respectively.

The computer output obtained from the authors' developed FORTRAN-90 code is given in Table 2:

Table 2: Computer Output for a Larger-Scale Example

```
==================================
  Authors = Eric+Don+Duc Nguyen, Version
              Date: 10-22-05
==================================
# lines for sequences {s} =  10
sequence {s} =
AGTACCTTGGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTTTTT
AGTACCTTGGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTACGT
```

```
ACGTAACCGTTATTTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTTTTT
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTACGT
AGTACCTTGGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTTTCC
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTTTAC
AACCTTGGTTTAAAACGTAAGGGGCTT
TACCGTTCAGTCATGGCATTCAGGTAC
GTTAACTGGGGCCATATATACGCG
# lines for sequences {t} =  9
sequence {t} =
ACGTAACCGTTACGAACCGGTTCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTACGT
AACGTTGGCCGTTTTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTTTTT
ACGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTACGT
AACGTACGTGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
CCGTAACCGTTACGTTAGGTACCTTAC
TTTACGGGGACTTACCTTATGCAAAAC
CGTTAAAACCTCACGTTTTTCCGGTT
AGTACCTTGGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
AGTACCTTGGTTTAAAACGTAAGGGGC
TTTACCGTTCAGTCATGGCATTCAGGT
ACGTTAACTGGGGCCATATATACGCG
AAACCGCCGTTACGTTAGGTACCTTAC
TTTACGGGGTTAACCCTTGCTTAAAAC
CGTTAAAACCTCGTACGTACGTACGT
CGGTTTCGTAACCGTTATTTTAGGTAC
CTTACTTTGACTTACCTTGCTTAAAACC
GTTAAAACCTCGTACGTACGTTTTT
 lengths of sequences {s} and {t} =  798 720
 Highest Total Score =  138
==============================
```

Example 3: Large-Scale Parallel Computation

In this example, the length of the 2 sequences {s}, and {t} are 40000, and 36000 (characters), respectively. For this large-scale problem, if serial computation was done in a single processor environment, one needs to generate 2 large integer matrices P(I,j), and M(I,j), where i=0,1,2…,40000 and j=0,1,2…,36000. Since each integer number (or word) requires 4 bytes, one needs to have approximately 11.5 GBytes (= 11.5*10**9 Bytes) RAM to store the above 2 integer matrices entirely in the core memory. For many distributed computer clusters (such as the BERNOULLI cluster, available at Old Dominion University), one only has approximately 1.7 GBytes RAM per processor (excluding the memory reserved for system processes). Thus, one needs to use at least 7 processors in order to execute the developed parallel MPI-FORTRAN90 code in an "incore" fashion. However, in a parallel, distributed computer cluster environment, each of the large integer matrices P(I,j), and M(I,j) will be distributed over different processors. Thus, each processor will need to store only "portions" of the above 2 large matrices. Using the "parallel block" algorithms developed in Section 3 of this paper, the wall-clock time for solving this large-scale problem is reported in Table 3. It should be noted here that the wall-clock time is rather large when 6 or less processors are employed, due to the fact that the entire matrices P(I,j), and M(I,j) will NOTfit in the core memory, cache missed will occur, and out-of-core I/O is required. However, when 7 (or more) processors are employed, the wall-clock time is dramatically reduced. In fact, when 24 parallel processors are used, it took only 10.88 seconds to solve this large-scale problem.

Table 3: Wall-Clock Time for Parallel Computing

# Processors:	2	4	6	7	10	24
Time (sec)	6622	2931	343	43.93	25.45	10.88

5 Conclusions

In this paper, Dynamic Programming Algorithms for efficient comparison and computation of the highest total score for aligning 2 sequences (with either equal, or different lengths) is reviewed. A simple, and efficient serial FORTRAN-90 code is developed, and listed in the Appendix. Due to

page limitation, parallel MPI/FORTRAN_90 source code is NOT listed here. Interested readers should contact the 3-rd author for obtaining the parallel version of the source code.

Both small, and medium-scale examples are used to validate the developed code (using the Old Dominion University SUN computer platform). For truly large-scale problems, such as the one discussed in Example 3, implementing the developed "parallel block" procedures (see Section 3 of this paper) have resulted in substantial time saving, as illustrated in Table 3.

Appendix. FORTRAN Source Code, I/O Data Files

```
c
c     program biology (comparing 2 DNA
      sequences)
c
c...authors:   Eric Nguyen, Don Nguyen and
               Duc Nguyen
c...version:   October 22, 2005
c...stored at: cd ~/cee/mail/*hanh*/
c...status:    Same answers as Setubal's book
               Examples 1 & 2 (pp. 50-51)
c...           Correct answers for Duc's
               medium-scale Example 3
c...           Correct answers for Duc's large-
               scale Example 4
c
      implicit real*4(a-h,o-z)
      integer g
parameter (maxlines=100, maxsize=80)
integer m(8000, 8000)
integer p(8000, 8000)
character*80 ipline   ! note: 80 = maxsize
character*8000 s  ! 8000 = maxlines*maxsize
character*8000 t  ! 8000= maxlines*maxsize
c     input 2 DNA sequences {s} = {AAAC},
      and {t} = {AGC}
open(unit=5, file='biology.dat', status='old',
      form='formatted')
c.....
write(6,*)'======================='
write(6,*)'Authors =Eric+Don+Duc Nguyen,
           Version Date: 10-22-05'
write(6,*)'======================='
c...
read (5,*) mlines
write (6,*) '# lines for sequences {s}=',mlines
c
      write(6,*) 'sequence {s} = '
      nloc=0
      do 31 ii=1,mlines
      read(5,1) ipline
      write(6,1) ipline
1     format(a)
      do 32 jj=1,maxsize
      if ( ipline(jj:jj) .ne. ' ' ) then
      nloc=nloc+1
      s(nloc:nloc)=ipline(jj:jj)
      endif
32    continue
31    continue
      isizes=nloc
c
      read (5,*) mlinet
write (6,*) '# lines for sequences {t} = ',mlinet
c
      write(6,*) 'sequence {t} = '
      nloc=0
      do 41 ii=1,mlinet
      read(5,1) ipline
      write(6,1) ipline
      do 42 jj=1,maxsize
      if ( ipline(jj:jj) .ne. ' ' ) then
      nloc=nloc+1
      t(nloc:nloc)=ipline(jj:jj)
      endif
42    continue
41    continue
      isizet=nloc
c
c....  output the size ( = length ) of sequences
       {s} and {t}
write (6,*) 'lengths of sequences {s} and
{t} = ',isizes,isizet
c....construct the integer matrix p(isizes,isizet)
      do 11 i=1,isizes
      do 12 j=1,isizet
      if ( s(i:i) .eq. t(j:j) ) then
      p(i,j)=+1
      else
      p(i,j)=-1
      endif
12    continue
11    continue
c....output p(isizes,isizet)
c     write(6,*) 'p(isizes,isizet) = ',((p(i,j),
      j=1,isizet),i=1,isizes)
```

```
c...construct matrix m(0...isizes, 0...isizet)
c...initialize row #0 of m(-,-)
    g=-2         ! g is an integer variable
    do 14 j=0,isizet
    m(0,j)=j*g
14  continue
c....initialize column #0 of m(-,-)
    do 15 i=0,isizes
    m(i,0)=i*g
15  continue
c..    now, generate other entries of m(-,-),
       based on its known 3 neighbors
       do 21 i=1,isizes
       do 22 j=1,isizet
       ii=m(i,j-1)+g
       jj=m(i-1,j-1)+p(i,j)
       kk=m(i-1,j)+g
       largest=max(ii,jj,kk)
       m(i,j)=largest
22     continue
21     continue
c......output m(isizes+1,isizet+1)
c-------------------------------------------
c      write(6,*) 'for row #0 of array m(-,-)'
c      write(6,*) (m(0,j),j=0,isizet)
c-------------------------------------------
c      write(6,*) 'for column #0 of array m(-,-)'
c      write(6,*) (m(i,0),i=0,isizes)
c-------------------------------------------
write(6,*)  'Highest Total Score = ',
            m(isizes,isizet)
c-------------------------------------------
      stop
      end
```

References:

[1] Joao Setubal, and Joao Meidanis, Introduction to Computational Molecular Biology, PWS Publishing Company, ISBN # 0-534-95262-3 (1997), pages ??-??

[2] David W. Mount, Bioinformatics: Sequence and Genome Analysis, 2-nd Edition, Cold Spring Harbor Laboratory Press, ISBN # 0-87969-712-1 (2004), pages 83-93

[3} Jeff Augen, Bioinformatics in the Post-Genomic Era: Genome, Transcriptome, Proteome, and Information-Based Medicine, Addison-Wesley, ISBN # 0-321-17386-4 (2005)

[4] Yi-Ping P. Chen, and Limsoon Wong (Editors), Proceedings of the 3-rd Asia-Pacific Bioinfomatics Conference, Imperial College Press, ISBN # 1-86094-477-9 (2005)

[5] Jason T.L. Wang, Cathy H. Wu, and Paul P. Wang (Editors), Computational Biology and Genome Informatics, World Scientific Publishing, ISBN # 981-238-257-7 (2003)

[6] Proceedings of the 2003 IEEE Bioinformatics Conference, August 11-14'03, Stanford, California, USA, IEEE Computer Society Press, ISBN # 0-7695-2000-6 (2003)

[7] Duc T. Nguyen, Parallel-Vector Equation Solvers for Finite Element Engineering Applications, Kluwer Academic/Plenum Publishers, ISBN # 0-306-46640-6 (2002)