Dynamic Programming Algorithm and Software in Parallel Computer Environment With Application in Computational Biology

DUC T. NGUYEN (Contact Author)
Civil & Environmental Engineering Department, Old Dominion University, 135 KAUF
Norfolk, VA 23529, USA
dnguyen@odu.edu

SIROJ TUNGKAHOTARA
INCA Engineers, Inc.
701 Poydras St., 14-th Floor
New Orleans, LA 70139, USA
s.tungkahotara@incainc.com

ERIC N.D. NGUYEN
Virginia Commonwealth University
School of Medicine (SoM)
Richmond, VA 23284, USA
nguyenen@vcu.edu

DON N. NGUYEN
Virginia Commonwealth University
School of Medicine (SoM)
Richmond, VA 23284, USA
nguyendn3@vcu.edu

ANDY L. NINH
The University of Florida
Department of Chemistry
Gainesville, FL 32611
aninh87@ufl.edu

Abstract: - The maximum score and the "best alignments" between the user's input DNA sequences and the large database for existing DNA sequences is conducted in this study. Both serial and (multiple processors) parallel computing algorithms are discussed. Numerical performance of the developed "parallel dynamic programming" algorithms and software are validated through small to large-scale applications. Results indicate that the developed parallel software is reliable and quite efficient, in terms of both computational speed and in-core (RAM) memory requirements.

Keywords: Sequence analysis/alignments; parallel dynamic programming algorithms; MPI/Fortran-90.

1. Introduction

Due to large-scale data manipulation required in the general areas of computational biology [1-7], and especially with the availability of modern, inexpensive high-

performance computers (which have multiple processors) [8, 9], larger problems' sizes in molecular biology can now be more efficiently analyzed, and the solution process can be dramatically shorten.

In this paper, the problem of finding the “maximum score” and the corresponding “alignments” between the user’s input sequence {t} with 1 (or more) sequences $\{s_i\}$ residing in an existing database, using “single level” parallel dynamic programming is 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:

$$\begin{matrix} G & A & - & C & G & G & A & T & T & A & G \\ G & A & T & C & G & G & A & A & T & A & G \end{matrix} \qquad (1)$$

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:

$1\cdot(-2)+1\cdot(-1)+9\cdot(+1)=6$ (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 are summarized in Section 2. Even though materials

presented in Section 2 has already been well-documented in existing literatures [1-7], it is still useful to summarize in here in order to facilitate the discussion of "parallel computational procedures" for Dynamic Programming algorithms, to be explained in Section 3. Validation for the "serial and parallel" computer software is conducted in Section 4. Finally, conclusions are drawn in Section 5. For readers who wish to obtain the serial and/or the parallel version(s) of the computer software, the 1st author of this joint paper (Prof. Nguyen, dnguyen@odu.edu; http://eng.odu.edu/cee/directory/dnguyen.shtml) should be contacted.

2. Brief Reviews of Dynamic Programming Algorithms [1-7]

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 exponentially large (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.(3) shows a 2-dimensional array [M] corresponding to the 2 given sequences:

$$s = AAAC, and\ t = AGC \tag{2}$$

				A	G	C
			0	**1**	**2**	**3**
		0	0	-2	-4	-6
	A	**1**	-2	1	-1	-3
				(+1)	*(-1)*	*(-1)*
	A	**2**	-4	-1	0	-2
[M] =				*(+1)*	*(-1)*	*(-1)*
	A	**3**	-6	-3	-2	-1
				(+1)	*(-1)*	*(-1)*
	C	**4**	-8	-5	-4	-1
				(-1)	*(-1)*	*(+1)*

(3)

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:

$$\begin{array}{cccc} A & A & A & C \\ - & A & G & C \end{array} \quad \begin{array}{cccc} A & A & A & C \\ A & - & G & C \end{array} \quad \begin{array}{cccc} A & A & A & C \\ A & G & - & C \end{array} \quad \begin{array}{cccc} A & A & A & C \\ A & G & C & - \end{array}$$

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. 3).

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 of this imaginary square! The reason for this observation is that 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 same 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:

$$sim[s(1...i), t(1...j)] = \max \left\{ \begin{array}{c} sim[s(1...i), t(1...j-1)] - 2 \\ sim[s(1...i-1), t(1...j-1)] + p(i, j) \\ sim[s(1...i-1), t(1...j)] - 2 \end{array} \right\} \quad (4)$$

In Eq.(4), $p(i, j) = +1$, if $s(i) = t(j)$, and $p(i, j) = -1$, if $s(i) \neq t(j)$.

The values of p(i,j) are written inside the parenthesis, shown in Eq.(3)

Eq. (3) can be expressed as:

$$M(i, j) = \max(M(i, j-1) - 2, M(i-1, j-1) + p(i, j), M(i-1, j) - 2) \quad (5)$$

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 Algorithms [8,9] For Dynamic Programming

3.1 *Finding the maximum score between two given DNA sequences*

Based upon the example shown in Eq.(3), 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 just 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	P_0	P_1	P_0	P_0	P_1	P_0
-4	P_2	P_1	P_1	P_2	P_1	P_0
-6	P_2	P_2	P_0	P_2	P_1	P_2
-8	P_0	P_1	P_0	P_2	P_0	P_0
-10	P_2	P_1	P_0	P_1	P_1	P_0
-12	P_2	P_1	P_2	P_2	P_1	P_2

(a)

0	-2	-4	-6	-8	-10	-12
-2	P_0	P_0	P_0	P_0	P_0	P_0
-4	P_1	P_1	P_1	P_1	P_1	P_1
-6	P_2	P_2	P_2	P_2	P_2	P_2
-8	P_0	P_0	P_0	P_0	P_0	P_0
-10	P_1	P_1	P_1	P_1	P_1	P_1
-12	P_2	P_2	P_2	P_2	P_2	P_2

(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.

3.2 *Finding the best alignments between two given DNA sequences*

Having computed the maximum score between the 2 sequences {s} and {t}, the 2-D integer array matrix [M] can be shown in Figure 2. In order to find the one-to-one "best alignments" between these 2 sequences, one starts with the final best score's location of the integer matrix [M]. For this particular example [see eq. (4)], the final best score's location (and its best score value) is M (i=4, j=3) = -1. The "backward path(s)", with solid "arrow" signs can then be efficiently found (as indicated in Figure 2). Information from these backward path(s) can be used to construct the one-to-one "best alignments" between the 2 sequences {s} and {t}, as illustrated in Figure 3. Details of finding the backward path(s) can be explained in the following paragraphs.

Starting with the final best score's location M (i=4, j=3) = -1, the backward path(s) from the location M(i,j) can only be one of the following 3 possibilities:

Case 1: If M (i,j) was originated from the previous location M(i-1,j), then the following conclusions and actions can be made:
(a) M(i,j) = M(i-1,j) + (g=-2)
(b) We align s(i) with a "blank space" (or "-")
And we set i = i – 1
j = j

Case 2: If M(i,j) was originated from the previous location M(i-1,j-1), then the following conclusions and actions can be made:
(a) M(i,j)=M(i-1,j-1) + p(i,j), where:
p(i,j) = +1, if s(i) .eq. t(j)
p(i,j) = -1, if s(i) .ne. t(j)
(b) We align s(i) with t(j)
and we set i=i-1
j=j-1

Case 3: If M(i,j) was originated from the previous location M(i,j-1), then the following conclusions and actions can be made:
(a) M(i,j)=M(i,j-1) + (g=-2)
(b) we align a "blank space" (or "-") with t(j)
and we set i=i
j=j-1
For this particular data, [M(i,j)] is a 4x3 matrix, the backward path(s) and the corresponding one-to-one alignments are shown in Figure 2 (see the "solid arrow" symbols) and Figure 3, respectively.

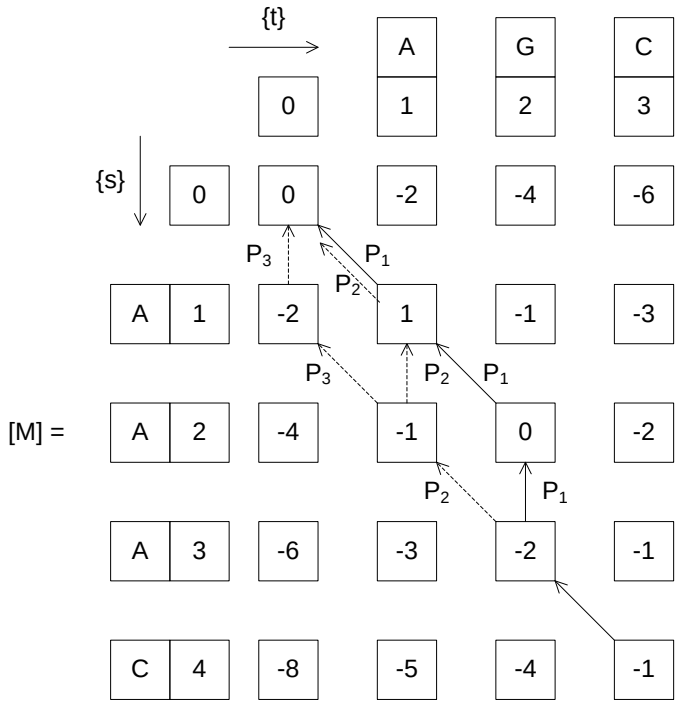

Figure 2: Three Different Paths (P_1, P_2, P_3) with the Same Maximum Scores

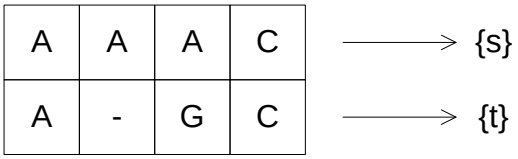

Figure 3: Best Alignments Between Sequences {t} and {s} (Using Path P_1 of Figure 2)

Remarks:

(1) If the orders of Case 1 and Case 3 are switched, then Case 3 is considered first and Case 1 is considered last, then different backward path(s) can be resulted (see dash arrow" symbols in Figure 2).

(2) All different backward paths and different best alignments can also be found and recorded (by minor changes made to the FORTRAN code) if one is willing to spend more computational efforts, and more computer memory (RAM) requirements!

(3) Since the computational efforts required to find the best one-to-one alignments between 2 DNA sequences is much less than the constructions of the (large) integer matrix [M] for finding the maximum score, and furthermore, the steps involved in this phase is more "serial" in natures, therefore, not much efforts have been spent to parallel this phase in this study.

4. Numerical Applications

Based upon the discussions presented in the previous sections, the serial and parallel MPI/FORTRAN-90 computer programs have been developed by the authors.

Example 1: A Small-Scale "serial" Example Problem

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

User's input data for this "serial" 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 "serial" Example

Authors = Eric+Don+Duc Nguyen, Version Date: 09-04-2006
Highest total score = 6 Output for optimum alignment GA-CGGATTAG GATCGGAATAG

The above results (see Highest Total Score) and the corresponding alignments between the above 2 sequences does match with the ones given in Ref. [1]

Example 2: A Large-Scale Parallel Computing Example Problem

In this example, the maximum score and the best alignments between two sequences {t} (with 36,000 character length) and {s} (with 40,000 character length) are computed by 5, 10, 15, 20 and 25 processors, respectively.

The computer output obtained from the authors' developed one-level parallel MPI FORTRAN-90 code is given in Table 2.

Example 3: A Medium-Scale Parallel Computing Example Problem

In this example, the user's input sequence {t} (with 16,000 character length) will be compared with the existing 40 sequences {s1, s2, …, s40} resided inside the database. For convenience, each of the 40 sequences {s_i}, where i = 1,2, …, 40, is assumed to have the same character length (= 16,000). Parallel processing is conducted in this study,

with upto 30 processors are utilized to assess the numerical performance of the proposed parallel algorithms.

The main components of the developed parallel algorithms and software (including the required input data) are summarized in the following paragraphs:

Step 1: User's specified input data for sequence {t}, such as:

(a) Number of lines (each line can have a maximum of 80 characters) required to input the sequence {t}. For example, number of lines = 1
(b) Input sequence {t}. For example, {t} = GATCGGAATAG
(c) User's input parameter "nrepeat1". For example nrepeat1 = 4 means that the input sequence {t}, specified in steps 1(a) and 1(b) will be REPEATED by "nrepeat1" times. This option is useful for generating a very long sequence {t}.

Step 2: Exactly as Step 1 input data, for a sequence {s1}, resided inside the database, such as:

(a) Number of lines. For example, number of lines = 1
(b) Input sequence {s1}. For example, {s1} = GACGGATTAG
(c) User's input parameter "nrepeat2"
(d) In addition to items 2(a), 2(b) and 2(c), the user will be required to provide the value of the input parameter "num_sequences_s" (say = 40), which specifies the number of sequences {s} = {si} inside the database. For convenience, the data specified in items 2(a,b,c) will be again repeated by "num_sequences_s" times.

Step 3: User's specified input parameters NP, and NGROUPS. For example NP = 60 processors, and NGROUPS = 1 (or 2, or 4 etc…). In all three examples considered in this work, we use NGROUPS = 1.

Note: If NGROUPS = 1 is specified by the user, then it implies that the user wants to use "1-level" parallel algorithms. Otherwise, if NGROUPS = 2 (or greater than 2), then "2-level" parallel algorithms will be activated. This "multi-level" parallel computation option is currently under development, and therefore, is unavailable for users at this moment.

Step 4: Using the "single level" parallel algorithms, the maximum score between the user's input sequence {t} and the sequences {s_i} resided inside the database will be computed in a parallel environment (as described in Section 3). The "global" maximum score amongst these "local" maximum scores will be identified. For example, the "global" maximum score occurs between sequence {t} and sequence {s_i*} can be identified from this example 3, say sequence {s_i*} = {s_1}.

Remark: To save the computer RAM memory requirements, the large integer array [M] (described in Section 3) will be distributed amongst the number of processors used. Thus, each processorwill store only "portions" (= blocks of rows) of the matrix [M]. Furthermore, the "alignment" process will "NOT" be done in this step, in order to avoid storing multiple copies of matrices [M], such as [M1] between sequence {t} and {s1} and [M2] between sequence {t} and {s2}, etc…

Step 5: Computing the maximum score (again) and the "alignment" process between the sequence {t} and with only 1 particular sequence {s_i*} inside the database.

The (wall-clock time) results for this example 3 are presented in Table 3.

Table 2: Computer Output for a Large-Scale "single level" parallel computation example (sizes of sequence {s} and {t} are 40000 and 36000 respectively)

No. of Processors	Time to find maximum score (sec)	Time to find the best alignment (sec)
5	21.70	2.50
10	10.65	2.17
15	7.33	2.14
20	5.73	2.28
25	4.83	2.43

Table 3: Computer Output for a Large-Scale "single level" parallel computation example (sizes of sequence {s_i}, where i=1,2, …, 40 and {t} are 16000 and 16000, respectively)

Authors = Andy+Eric+Don+Duc Nguyen +Tungkahotara,

Number of CPU	Level of parallelization	Time to find the maximum score (sec)
10	Single-level	126.93
15	Single-level	79.32
20	Single-level	53.86
25	Single-level	41.13
30	Single-level	39.83

Remarks About Table 3's Timing Results:

In this case, we find the maximum score and do the alignments between the user's specified sequence {t} with the existing 40 sequences {s1, s2, …, s40} resided in the database. One (single)-level parallel processing with 10, 15, 20, 25 and 30 processors are conducted. When 30 processors (1-level) are used, it means that we use 30 processors to compare (find the maximum score) of sequence {t} and sequence {s1}. When this task is completed, then the same 30 processors are used to compare sequence {t} and {s2}, etc…

As can be seen from Table 3, the 1-level parallel algorithm scales very well between 10 and 25 processors (speed-up factor = 126.93/41.13 = 3.09, when the number of processors are increased by a factor of 25 processors / 10 processors = 2.5, with the calculated efficiency = 3.09/2.50 = 123.44%. Thus, super-linear speed-up has been achieved). When 30 processors are used, one obtains the speed-up factor = 126.93/39.83 = 3.19, when the number of processors are increased by a factor of

30 processors / 10 processors = 3, with the calculated efficiency = 3.19/3.00 = 106.23%.

This super-linear efficiency factor is less than the one obtained earlier when 25 processors have been used.

The above observation can be explained that for the "fixed problem's size" of sequence {t} and {s_i}, the usefulness of parallel processing will be diminished when the number of processors exceed a certain value (say around 30 processors, in this particular example), since the computational efforts per processor will become unimportant as compared to the increasing communication time amongst the processors.

5. Conclusions

In this paper, both "serial" and single-level "parallel" Dynamic Programming Algorithms for efficient comparison and computation of the maximum scores for best aligning 2 (or more) pairs of sequences (with either equal, or different lengths) is reviewed. A simple, and efficient (single-level) parallel MPI FORTRAN-90 software have been developed. Both small, and large-scale examples are used to validate the developed code (using the Old Dominion University SUN computer platform). In the small-scale example, the obtained results have been compared with Ref. [1]. For the large-scale example, the numerical performance has been highly scalable. In facts, super-linear parallel speed-ups have been observed in several cases. The developed single-level parallel algorithms, therefore, will allow computational biology research communities to solve very large-scale problems without worrying about the computer memory (RAM) restriction, and the wall-clock computational time. Extension of this research work for "multi-level" parallel processing (for even further reduction in computational time !) will be reported in a very near future.

References

1. Setubal J, Meidanis J, *Introduction to Computational Molecular Biology*, PWS Publishing Company, ISBN # 0-534-95262-3, 1997.
2. Borodovsky M, Ekisheva S, *Problems and Solutions in Biological Sequence Analysis*, Cambridge University Press, problem 2.9, pp. 39, 2006.
3. Wang JTL, Wu CH, Wang PP (Editors), *Computational Biology and Genome Informatics*, World Scientific Publishing, ISBN # 981-238-257-7, 2003.
4. Richard Durbin, Anders Krogh, Sean R. Eddy, and Graeme Mitchison, Biological Sequence Analysis, Cambridge University Press (1998)
5. Augen J, *Bioinformatics in the Post-Genomic Era: Genome, Transcriptome, Proteome, and Information-Based Medicine*, Addison-Wesley, ISBN # 0-321-17386-4, 2005.
6. Chen YP, Wong L (Editors), Proceedings of the 3-rd Asia-Pacific Bioinfomatics Conference, Imperial College Press, ISBN # 1-86094-477-9, 2005.
7. Mount DW, *Bioinformatics: Sequence and Genome Analysis*, 2-nd Edition, Cold Spring Harbor Laboratory Press, ISBN # 0-87969-712-1, pp. 83-93, 2004.

8. Nguyen DT, *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*, Springer Publisher, ISBN# 0-387-29330-2, April 2006.

9. Nguyen DT, *Parallel-Vector Equation Solvers for Finite Element Engineering Applications*, Kluwer Academic/Plenum Publishers, ISBN # 0-306-46640-6, 2002.