Paralellization of Needleman-Wunsch String Alignment Method

Jaime Seguel
Department of Electrical and Computer Engineering
University of Puerto Rico at Mayaguez
Mayaguez, Puerto Rico
Jaime.seguel@upr.edu

Carlos Torres
Department of Electrical and Computer Engineering
University of Puerto Rico at Mayaguez
Mayaguez, Puerto Rico
Carlos.torres9@upr.edu

***Abstract*— The identification of homologies between protein sequences is a central problem in molecular biology and several algorithms have been proposed for accomplishing this task. The Needleman-Wunsch Algorithm and its close variant, the Smith-Waterman dynamic programming methods, solve the problem exactly in quadratic time and space. However, due to the massive amount of data involved in sequence-to-database of sequences comparisons, heuristic methods such as BLAST, are used instead. This article explores the design of a parallel version of Needleman-Wunsch based on a divide-and-conquer idea derived from an algorithm first proposed by Hirschberg, for the identification of the longest common substring of two strings.**

Keywords: string alignment, dynamic programming, divide-and-conquer, recursion, parallel computation.

I. INTRODUCTION

The identification of homologies between protein sequences is a central problem in molecular biology. In computational terms, the problem is stated as the search of an optimal alignment between a pair of strings over the 20-character amino acid alphabet. An alignment of a pair of strings $S_1=x_1\ldots x_m$ and $S_2=y_1\ldots y_n$, is a $2 \times q$ matrix A[i, j], where i = 1, 2 and $q \geq \max\{m, n\}$; whose entries are the characters of the amino acid alphabet or a gap symbol " – ". In addition, A satisfies:

i. For each i = 1, 2 and j, A[i, j] is a character of the protein alphabet or either A[1, j] = – or A[2, j] = –, but not both;
ii. If all gap symbols are removed and each empty cell is filled by left shifting by one all the cells to its right, then A[1, j] = x_j and A[2, j] = y_j

The similarity is quantified with respect to a substitution matrix that establishes the rate at which a character in the amino acid alphabet changes to each of the other characters in the same alphabet, over time. We denote the substitution rate of pair of characters x_k and y_j as $r(x_k, y_j)$. A gap penalty, in turn, is a numerical cost imposed for the insertion of the gap symbol in either A[1, j] or A[2, j]. Gap penalties may be assigned as a constant per gap symbol inserted, or as an affine gap penalty containing separate costs for initiating and for extending the gap with consecutive gap symbol insertions. For the sake of simplicity, we consider solely constant gap penalties. The score of an alignment is the sum of the rate of substitution of each pair of aligned characters minus a gap penalty per each character that is aligned with the gap symbol. The higher the score, the better the alignment. For example, let's consider the alignment of the pair (S_1, S_2); with S_1 = PAWHEAE and S_2 = HEAGAWGHEE [1]. The next alignment is optimal with respect to the BLOSUM50 substitution matrix and a penalty gap of $\gamma = -8$.

TABLE I. AN OPTIMAL ALIGNMENT OF S_1 AND S_2

H	E	A	G	A	W	G	H	E	–	E
–	–	P	–	A	W	–	H	E	A	E

This alignment, which reaches the optimal score of 1, is not necessarily unique. Indeed, in this particular case as in many others, there are other alignments that reach the same score.

A. Dynamic Programming

An exhaustive search for an optimal alignment takes exponential time. Needleman and Wunsch [2] introduced a quadratic time and space dynamic programming method known today as the Needleman-Wunsch algorithm (NWA). NWA computes an optimal alignment in two main steps. The first step uses a recursive formula to fill in, usually row-by-row or column-by-column; a dynamic programming matrix D . The next pseudo-code describes this process.

Step 1: Computation of the Dynamic Programming Matrix
For each k, $0 \leq k \leq m$;
  D[k, 0] $\leftarrow k\times\gamma$
For each j, $0 \leq j \leq n$;
  D[0, j] $\leftarrow j\times\gamma$
For each k, $1 \leq k \leq m$;
  For each j, $1 \leq j \leq n$;
    D[k, j] $\leftarrow \max\{D[k-1, j-1]+r(x_k, y_j), D[k-1,j]+\gamma, D[k,j-1]+\gamma\}$

Matrix D contains the optimal scores for the alignment of all subsequences of S_1 and S_2 that start at x_1 and y_1;

respectively. The last entry in this matrix, this is D[m, n], is the optimal score for the alignment of S_1 and S_2. The second step of NWA backtracks the path of solutions of the subsequence alignments that led to the optimal score D[m, n]. We refer to a path of indices of D that results from this process as backtrack-path or b-path. Backtracking is summarized in the next pseudo code.

Step 2: Computation of a b-path
b-path ←*{[m, n], [0, 0]}*
While [k, j] ≠ [0, 0] is in b-path
If D[k, j]=D[k-1,j-1]+r(x_k,y_j)
b-path ← *b-path* ∪ *{[k-1,j-1]};*
Else if D[k, j]=D[k-1,j]+γ
b-path ← *b-path* ∪ *{[k-1, j]};*
Else if D[k, j]=D[k,j-1]+γ
b-path ←*b-path* ∪ *{[k, j-1]};*

The actual alignment follows from the b-path by applying the next rule:

If [k-1, j-1] and [k, j] are in the b-path, x_k is aligned with y_j
If [k-1, j] and [k, j] are in the b-path, x_k is aligned with –
If [k, j -1] and [k, j] are in the b-path, y_j is aligned with –

NWA solves the global alignment problem, which is, the optimal alignment of the whole S_1 and S_2 input sequences. However, in many biological instances, sequences share only segments of meaningful similarity. These may vary from short regions to large domains of recognizable similarity. The so-called local alignment problem is the problem of identifying the segments in S_1 and S_2 with the highest alignment score. An exact algorithmic solution of the local alignment problem is the Smith-Waterman algorithm (SWA) due to Smith and Waterman [3]. The SWA is a variant of the NWA that replaces negative scores with zeroes masking thus, segments of low similarity score. SWA dynamic programming and backtracking steps are similar to the ones in NWA except that backtracking starts from the indices of the entry with the highest score in the whole matrix D, and ends right before the first 0 encountered while performing the backtracking process. Thus, unlike b-paths, local backtracking paths or lb-path are not necessarily anchored in [m, n] and [0, 0]. This difference has deep algorithmic consequences.

B. Some Previous Parallelization Attempts

Filing D with either NWA or SWA takes O(mn) time and space. Backtracking, in turn, is accomplished in O(m + n) time. Although most implementations do not store D, the information needed to backtrack the optimal alignment still takes O(mn) space. Due to the large size of protein sequences and protein databases, SWA is often replaced with the heuristic BLAST [4] (Basic Local Alignment Sequence Tool) algorithm. Unlike BLAST, whose heuristics is well suited for parallelization; the parallelization of NWA and SWA is limited by the recursive nature of their core processes. Some NWA and SWA parallelization attempts exploit the fact that each entry in D depends solely on its northern, western and northwestern neighbors. Thus, entries lying in the same anti-diagonal of D can be computed in parallel. This method is referred as the wave-front computation of D and is sometimes attributed to Gotoh [5] in the literature. Hsien-Yu Liao [6] et. al. use the wave-front computation to pipeline the search for an optimal alignment of a query sequence and the sequences in a database. The idea is to slide the front-wave across an enhanced scoring matrix whose rows correspond with the query sequence while the columns, to the concatenation of all the sequences in the database. Such concatenation is expected to diminish the latency incurred in starting each new comparison. Another method, introduced by Fa Zheng [7] et. al. splits the query sequence S_1 into a fixed number of sequences of approximately the same length. Each sub-sequence of S_1 is aligned with S_2 independently. A drawback in this method is that the scores matrices, which are computed in parallel, do not always correspond to sub-problems of the original alignment problem. Thus, in order to retrieve the alignment from the computed matrices, the authors propose what they called *combine and extend* method; which compromises the sensitivity of the result.

Most attempts to speedup NWA or SWA concentrate in the computation of the score matrix. This article takes a more integral approach. The starting point in our search for a parallel method is a non-recursive alternative to backtracking. The proposed alternative is based on symmetry properties that arise when matrix D is compared D*, the dynamic programming matrix of the alignment of S_1* and S_2*, which are the original sequences S_1 and S_2 but in reversed order.

II. PROPERTIES OF THE DYNAMIC PROGRAMMING MATRIX

Hirschberg [8] introduced an O(m + n) space algorithm for finding the longest common sub-string of a pair of strings. His method relies on the rather obvious fact that the longest common sub-string of S_1 and S_2 is the same as the longest common sub-string of S_1* and S_2*. Hirschberg's idea has been extended to the calculation of the edit distances between pairs of sequences and to the global alignment of a pair of sequences. The authors did not find in the literature any extension of Hirschberg's method to the parallelization of NWA. This section develops the mathematical foundations of Hirschberg-NWA space saving algorithm, and describes it in a way that makes it more suitable for the parallel NWA discussed in section III.

A. The D-D symmetry*

The mathematical facts that allow the use of Hirschberg's ideas in the solution of the problem of the global alignment of a pair of sequences are stated below.

Lemma 1. *The optimal alignment of a pair (S_1, S_2) in reversed order is an optimal alignment for the pair (S_1*, S_2*); and vice versa.*

The next theorem is crucial in the parallelization and space saving strategy of NWA to be discussed in the next section.

Supported by NIH-MARC "Assisting Bioinformatics Efforts at Minority Institutions" and NIH-RISE "RISE-4-BEST"

Theorem 1. *Let D and D* be* the *score matrices produced with NWA for the pairs (S_1, S_2) and (S_1*, S_2*), respectively. Then, for each $0 \le k \le m$ and each $0 \le j \le n$,*

i. $D[k, j] + D^*[m - k, n - j] \le D[m , n]$

ii. $D[k, j] + D^*[m - k, n - j] = D[m, n]$ *if and only if* $[k, j]$ *is in a b – path.*

Proof. The proof of assertion i. is by induction on k and j. For the base case we set k = 0 and j = 0. Thus, the statement to be proved is $D[0, 0] + D^*[m, n] \le D[m , n]$. This statement is true because, after Lemma 1, $D[m, n] = D^*[m , n]$ and $D[0, 0] = 0$. We assume now that there is a pair of indices k, j for which $D[k, j] + D^*[m - k, n - j] \le S[m, n]$ and prove that under this assumption the statement:

(a) $D[k + 1, j] + D^*[m - k - 1, n - j] \le D[m, n]$, and

(b) $D[k, j + 1] + D^*[m - k, n - j - 1] \le D[m , n]$, and

(c) $D[k + 1, j + 1] + D^*[m - k - 1, n - j - 1] \le D[m, n]$,

is also true. The proof of claim (a) is as follows. Since by definition of the NW recursion $D[k + 1, j] \le D[k , j] + \gamma$, we have that

$D[k + 1, j] + D^*[m - k - 1, n - j] \le$

$D[k , j] + \gamma + D^*[m - k - 1, n - j] \le$

$D[k, j] + D^*[m - k, n - j] \le D[m, n]$.

Sub-statement (b) is proved similarly. In order to demonstrate claim (c) let B[k, j] be the entry for the pair (x_k, y_j) in the substitution matrix. Then,

$D[k + 1, j + 1] + D^*[m - k - 1, n - j - 1] \le$

$D[k, j] + B[k + 1, j + 1] + D^*[m - k - 1, n - j - 1] \le$

$D[k, j] + D[m - k, n - j] \le D[m ,n]$.

Assertion *ii* is a direct consequence of Lemma 1. □

The index relation ([k, j], [m – k, n – j}) is referred as D-D* symmetry and the entries D[k, j] and D*[m – k, n – j] as D-D* symmetric entries.

B. The O(m + n) Space Hirschberg-NWA

Theorem 1 provides the mathematical basis for a space saving Hirschberg-NWA (HNWA). This method, which follows the divide-and-conquer paradigm, uses Step 1 of NWA repeatedly, each time over sequences of approximately half the size of the previous ones, to divide the problem in sub-problems, until a predetermined sub-problem size is reached. Only the last column of each intermediate sub-problem dynamic programming matrices D and D* are temporarily stored to determine the indices [k, j] that satisfy statement *ii* of Theorem 1. Once [k, j] is known, the problem is split in two sub-problems. Indeed, because of the general form of a b-path, the indices [r, s] of the b-path segment from [0, 0] to [k, j] must satisfy $0 \le r \le k$. Similarly, the indices [r, s] of the b-path segment from [k, j] to [m, n] must satisfy $k \le r \le m$. Thus, the search for the next indices that satisfy statement *ii* of Theorem 1 is reduced to the upper leftmost k × j block D[r, s], $0 \le r \le k$, $0 \le s \le j$; and the lower rightmost (m – k) × (n – j) block, this is, D[r, s], $k \le r \le m$, $j \le s \le n$. This splitting identifies a pair of subsequences of S_1 and S_2, which are aligned in the global alignment of S_1 and S_2, except perhaps for the introduction of gaps. By selecting j as close as possible to the middle of the subsequence of S_2, the corresponding blocks in the dynamic programming matrix are of similar size in the average case. This process of splitting sequences in subsequences, which we refer as the divide phase, is repeated on each of the newly identified subsequence up until a predetermined subsequence length is reached. The divide phase ends with the application of NWA to each of the subsequences of predetermined size. It is easy to demonstrate that the divide phase can still be performed in O(mn) time, although with a higher constant. The cost in memory space, in turn, is reduced in the best case (i.e. one point subsequences) to O(m + n). The conquer phase of the method is also linear in time and space as it consists basically in pasting together the b-path segments computed at the end of the divide phase.

C. A Case Study

We illustrate the base divide and conquer technique of HNWA with the problem of finding an optimal alignment for S_1 = HEAGAWGHEE and S_2 = PAWHEAE. Before the illustration of the divide phase it is worth remarking that Step 1 of NWA can be modified to compute the rightmost column of D in-place, this is using only the storage space of one column. This is an essential element in HNWA memory space reduction strategy. The next pseudo code, which illustrates such in-place computation, uses a one-dimensional array C[k], $0 \le k \le m$; to store intermediate and final result.

Step 1.a. In-place computation of the rightmost column of D

```
For k = 1 to m
    Aux2 ← C[k]
    If Aux2 + γ  > Aux1 + r(x_k, character)
        C[k] ← Aux2 + γ
    Else
        C[k] ← Aux1 + r(x_k, character)
    If C[k] < C[k – 1] + γ
        C[k] ← C[k – 1] +γ
    If C[k] < 0
        C[k] ← 0
    Aux1 ← Aux2
Return C
```

We return now to the example. In the first step of the division phase, we select j = 5 and split sequence S_1 in two halves each of length 5. The second half is written in reversed order. Thus, we split the original problem of aligning the pair (HEAGAWGHEE, PAWHEAE) into two independent problems, namely; that of aligning the pair (HEAGA, PAWHEAE) and that of aligning the pair (EEHGW, EAEHWAP). Now, using the Step 1.a described above, we compute the rightmost columns of the dynamic programming matrices of each of these pairs of sequences. Although in practice these matrices are not store, for the sake of clarity we present in Table II the full dynamic programming matrices for these two pairs of subsequence alignments.

TABLE II. DYNAMIC PROGRAMMING MATRCES FOR (HEAGA, PAWHEAE) AND (EEHGW, EAEHWAP)

Ind		0	1	2	3	4	5
	Char		H	E	A	G	A
0		0	-8	-16	-24	-32	-40
1	P	-8	-2	-9	-17	-25	-33
2	A	-16	-10	-3	-4	-12	**-20**
3	W	-24	-18	-11	-6	-7	-15
4	H	-32	-14	-18	-13	-8	-9
5	E	-40	-22	-8	-16	-16	-9
6	A	-48	-30	-16	-3	-11	-11
7	E	-56	-38	-24	-11	-6	-12

Ind		0	1	2	3	4	5
	Char		E	E	H	G	W
0		0	-8	-16	-24	-32	-40
1	E	-8	6	-2	-10	-18	-40
2	A	-16	-2	5	-3	-10	-18
3	E	-24	-10	4	5	-3	-11
4	H	-32	-18	-4	14	6	-2
5	W	-40	-26	-12	6	11	**21**
6	A	-48	-34	-20	-2	6	13
7	P	-56	-42	-28	-10	-2	3

Two border columns and rows have been added to keep track of the matrix indices and their corresponding characters in the sequences. By adding the D-D* symmetric entries of the rightmost columns of the dynamic programming matrices (i.e. the fifth column of each matrix in this case) we find that:

$$2 = \arg\max \{D[k, 5] + D^*[7 - k, 5]: 0 \le k \le 7\}. \quad (1)$$

Therefore, [2, 5] is in the b-path and the search for the segment of the b-path to the left of [2, 5] is reduced to the set of indices $\{[r, s]: 0 \le r \le 2, 0 \le s \le 5\}$; while the search for the segment to right of [2, 5] is reduced to $\{[r, s]: 2 \le r \le 7, 5 \le s \le 10\}$ or, in terms of D*, to $\{[r, s]: 0 \le r \le 5, 0 \le s \le 5\}$. The next step is to reduce the sequences accordingly. This gives the reduced pairs (HEAGA, PA) and (EEHGW, EAEHW). At this point, the algorithm checks whether the lengths of all the latter sequence segments are less than or equal to the predetermined maximal length. If this is not the case, subsequences HEAGA and EEHGW are split into two new sequences and the above process is repeated to get two new reduced pairs out of each of (HEAGA, PA) and (EEHGW, EAEHW). This decomposition generates a binary tree that at each leaf has a pair of segments of the original sequences whose length is less than or equal to the predetermined length. At this point, a b-path for each pair of segments is computed and the conquer phase started. For the sake of simplicity, let's assume that the predetermined length is 5 in the example. Then, the following dynamic programming matrices need to be computed and stored and process with Step 2 of NWA.

TABLE III. DYNAMIC PROGRAMMING MATRICES FOR (HEAGA, PA) AND (EEHGW, EAEHW)

Ind		0	1	2	3	4	5
	Char		H	E	A	G	A
0		0	-8	-16	-24	-32	-40
1	P	-8	-2	-9	-17	-25	-33
2	A	-16	-10	-3	-4	-12	**-20**

Ind		0	1	2	3	4	5
	Char		E	E	H	G	W
0		0	-8	-16	-24	-32	-40
1	E	-8	6	-2	-10	-18	-40
2	A	-16	-2	5	-3	-10	-18
3	E	-24	-10	4	5	-3	-11
4	H	-32	-18	-4	14	6	-2
5	W	-40	-26	-12	6	11	**21**

By applying Step 2 of NWA to each of these matrices we get the b-paths are {[2, 5], [1, 4], [1, 3], [0, 2], [0, 1], [0, 0]} and {[5, 5], [4, 4], [4, 3], [3, 2], [2, 1], [1, 1], [0, 0]}, respectively. And by applying the previously discussed rules for constructing alignments to each b-path we get the alignments,

H	E	A	G	A
–	–	P	–	H

and

E	–	E	H	G	W
E	A	E	H	–	W

Finally, by reversing the second alignment and concatenating it to the first one we retrieve the optimal alignment of Table 1.

It can be easily proved that all other optimal alignments are obtained from combinations of the alternative optimal alignments of each of the leaf pairs of sequence segments.

III. PARALLELIZING HIRSCHBERG-NWA

The parallelization of the previously discussed method exploits all independent computations in HNWA divide and conquers phases. These are, in summary, the computation of

the rightmost column of the dynamic programming matrix for the optimal alignment of each pair of subsequences, the computation of the b-path of each pair of subsequences of length less than or equal to the predetermined maximum length, and the production of the corresponding alignments. We use the master-workers paradigm with 2^P workers for describing the parallel method. Each worker is identified by a worker's identification number q, $1 \le q \le 2^P$. The master's identification number is 0.

A. Parallel HNWA

The following pseudo code is a high level description of a parallel HNWA.

Master:
p ← 1 (global variable)
On input (S_1, S_2)
Aux ← S_1
If (length (Aux) > L or length (S_2) > L) and p < P
S_1 ← first half of Aux
Send (S_1, S_2) to Worker 1
S_1 ← reversed second half of Aux
*S_2 ← S_2**
Send (S_1, S_2) to Worker 2
Receive (Al(2^p – 1), Al(2^p))
*Concatenate Al(2^p – 1) and Al(2^p)**
Return
Else perform NWA(S_1, S_2)
Worker q:
If $1 \le q \le 2^{p-1}$
Receive (S_1, S_2)
Step a: Compute column C with Step 1.a on (S_1, S_2)
If $2^{p-1} < q \le 2^p$
Send C to Worker $q - 2^{p-1}$
Else, Receive C from Worker $q + 2^{p-1}$
Compute index k in formula (1)
Send $k \leftarrow m - k$ to Worker $q + 2^{p-1}$
S_2 ← First k characters of local S_2
Aux ← local S_1
p ← p + 1 (global update)
If (length (Aux) > L or length (S_2) > L) and p < P
Local S_1 ← first half of Aux
S_1 ← reversed second half of Aux
*S_2 ← S_2**
Send (S_1, S_2) to Worker $q + 2^{p-1}$
Go to Step a
Else Al(q) ← NWA alignment of (S_1, S_2)
If $1 \le q \le 2^{p-1}$
Send Al(q) to Worker $q + 2^{p-1}$
Else Receive A($q - 2^{p-1}$)
*Al(q) ← Concatenation of Al($q - 2^{p-1}$) and Al(q)**
p ← p – 1
Send Al(q) to Worker $q + 2^{p-1}$

The pseudo code imposes an additional condition for the halting of the HNWA divide phase. The divide phase stops when the length of all subsequences is less than or equal to a predetermined length L > 0 or when all workers are busy. If the conditions for splitting a local subsequence are met at Worker q, then Worker q keeps the first half of its local S_1 segment and the first k characters of its local segment of S_2 to repeat the processes on them, and sends the second half and m – k (local m) remaining characters of S_2, both in reversed order, to Worker $q + 2^{p-1}$. Therefore, if for instance, P = 2 and the conditions for splitting the sequences are always met, the divide phase will involve 2 parallel steps. First, the master sends tasks to Worker 1 and Worker 2. When these parallel tasks are completed, Worker 1 sends a sub-task to Worker 3 and Worker 2 a sub-task to Worker 4. All four workers process their sub-tasks in parallel. So, ideally, the parallel tasks in the divide phase spawn a binary tree of height 2, rooted at the master's task. There is P parallel communications, as well. The conquer phase, in turn, traverses this tree from the leaves up in P additional parallel steps. First, workers 1, 2, 3 and 4 produce their local alignments in parallel. Then, Worker 1 sends its alignment to Worker 2 and Worker 3 sends its alignment to Worker 4. At this point, Worker 2 and Worker 4 concatenate their alignments in parallel and send the result to the master.

B. Performance Estimations

The next analysis, which is based on a highly simplified performance model, shows that the proposed parallelization has the potential to speed up the execution time of NWA. Let t(N) be the execution time of the NWA on a problem of size N. Then, t(N) = d(N) + b(N), where d(N) is the time for the computation of the dynamic programming matrix and b(N), the time for computing the b-path and forming the alignment. The P steps in the divide phase of the parallel HNWA will take approximately

$$(\tfrac{1}{2} + \tfrac{1}{4} + \ldots + \tfrac{1}{2^P})d(N) = (1 - \tfrac{1}{2^P})d(N) \text{ time.} \qquad (2)$$

Since local b-paths and alignments are computed in parallel, the time for producing them can be estimated as approximately $\frac{1}{2^P} \times b(N)$. We also estimate the parallel communication overheads as a linear function of N, which we represent as $c \times N$. Thus, the speed up formula is:

$$[d(N) + b(N)] / [(1 - \tfrac{1}{2^P})\, d(N) + \tfrac{1}{2^P}\, b(N) + cNP]. \qquad (3)$$

Now, taking into account that d(N) is in general much larger than b(N), the quotient f(N) = d(N) / b(N) is always a fraction $0 < f(N) < 1$, which tends to 0 as N growths to infinity. By dividing equation (3) by d(N) we get,

$$[1 + f(N)] / [(1 - \tfrac{1}{2^P}) + \tfrac{1}{2^P}\, f(N) + cPN/d(N)]. \qquad (4)$$

Therefore, as N growths, the theoretical speed up approaches $S = 1 / [1 - \frac{1}{2^P}]$.

C. Pipelining

After returning their local alignments Worker 1 and Worker 2 are idle. Therefore, a new pair of sequences (S_1, S_2) can be received from the master. Subsequent returns from workers liberate the necessary processors for the new sequences to spawn the binary tree of tasks, if required. This is especially suitable for the parallel processing a query sequence; let's say S_1, against a database of sequences.

IV. CONCLUSIONS

The exploitation of D-D* symmetries, which are derived form the original ideas of Hirschberg, renders a parallel version

of the NWA. The parallel method has a theoretical speed up over the serial NWA and allows for the pipelined processing of a query sequence against a database of sequences. The speed up formula obtained from a simplified performance model seems to indicate that for large problems, the parallel method is more advantageous for small number of processors, this is, a coarse grain parallelization.

REFERENCES

[1] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, Biological Sequence Analysis Cambridge University Press, 2007.

[2] Needleman S, Wunsch C, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," Journal of Molecular Biology, Vol. 48, Issue 3, pp. 443-453, 1970.

[3] T.F Smith and M.S. Waterman, "Identification of common molecular subsequences", Journal of Molecular Biology, 147(1), 195-197, 1981.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool", J. Mol. Biol., 215, pp.403-410, 1990.

[5] O. Gotoh, "An improved algorithm for matching biological sequences", J. Mol. Biol., 162, 705-708, 1982.

[6] Hsien-Yu Liao, Meng-Lai Yin, Yi Cheng, "A parallel implementation of the Smith-Waterman algorithm for massive sequences searching", Proceedings of the 26th Annual International Conference of the IEEE EMBS, San Francisco, CA, USA; September 1-5, 2004.

[7] F. Zhang, X. Z. Qiao, Z. Y. Liu, "A parallel Smith-Waterman algorithm based on divide and conquer", Proceedings Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PPí02) 2002.

[8] D. Hirschberg, "A linear space algorithm for computing maximal common subsequences", in Communications of ACM, Vol. 18, No. 6, pp. 341-343, 1975.