SEQUENCE ALIGNMENTS

ON A MULTI-TRANSPUTER SYSTEM

SEQUENCE ALIGNMENTS

ON A MULTI-TRANSPUTER SYSTEM

By

ZHIGUANG QIAN

B.ENG., M.SCI

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Engineering

McMaster University

MASTER OF ENGINEERING (1992) **McMASTER UNIVERSITY**
(Computer Engineering) Hamilton, Ontario

TITLE: Sequence Alignments on a Multi-Transputer System

AUTHOR: Zhiguang Qian,
B.Eng. (D.S.T.U., China)
M.Sc. (The 2nd Institute of A.S.I.M., China)

SUPERVISOR: Dr. Tao Jiang

NUMBER OF PAGES: viii, 115

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Dr. Tao Jiang, my supervisor, for his guidance throughout my research. His encouragement and support has been invaluable to me in order to fulfil this thesis.

I would like to thank Dr. Sanzheng Qiao and Mr. Dan Trottier for their help on some technical matters related to transputer. Thanks are also given to Dr. Qiao and Dr. Dan McCrackin for agreeing to read this thesis.

Finally, I would like to thank my wife and son for their love, support and understanding.

ABSTRACT

This thesis is concentrated on parallelizing a sequential algorithm for finding k best non-intersecting local sequence alignments.

In this thesis, the DNA local sequence alignment and the related problems are formally defined and efficient algorithms for solving these problems are presented. The problem have important applications in molecular biology. Based on the analysis of the characteristics of the local sequence alignment problem and a multi-transputer system, the problem was partitioned into subproblems and nicely mapped onto the transputer nodes. Then, an efficient parallel program is designed and implemented.

By comparing the outputs of the sequential program and the parallel program, the performance of the parallel program is estimated. An average speed-up of 6.3 is achieved on a 8-node configuration and an average speed-up of 11 is achieved on a 16-node configuration.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued) page

LIST OF ILLUSTRATIONS

page

LIST OF TABLES

page

CHAPTER 1

INTRODUCTION

The central dogma of modern biology is that DeoxyriboNucleic Acid (DNA) is the primary genetic material. It encodes the information necessary to understand life. From a biologist's point of view, DNA is a molecule composed of four nucleotides: adenine, cytosine, guanine, and thymine, which, conceptually, are linked linearly to form long chains called polynucleotides. Often these chains are called DNA sequences (or sequences, for simplicity).

The coming of new DNA sequencing technologies has led to an explosive growth in the quantity of biological sequence information available to researchers, a trend that is likely to accelerate in the near future [1, 20]. The benefits of this sequence information have already been clearly established, with gains in knowledge of the biological structure and function of many genes and the proteins they encode, resulting in important insights into human biochemistry, physiology, and disease processes.

The biologists' ultimate goal is to discover the semantics of DNA sequences, i.e., the meaning of the DNA. To understand the semantics, one needs to know the relationship between DNA and proteins. Proteins are sequences made from 20 animo acids. A piece of DNA can encode a protein. This means that each triple of nucleotides corresponds to an amino acid. Each of these triples is called a codon. The code (mapping)

from the 64 possible triples to the amino acids is redundant. That is, some amino acids correspond to more than one triple. Since proteins are responsible for important biochemical functions within a living cell, it is important to know which parts of the DNA encode proteins as well as what the proteins do.

When two or more sequence are displayed with one sequence written over another, the resulting configuration is known as an alignment of the sequences[27]. These displays are very common in molecular biology as they communicate information about proposed common evolution or function of the nucleotide positions found in any given column of an alignment.

The topic of sequence alignments has received much attention as the advent of molecular biology. One application of sequence alignment is the study of evolutionary relationship, where we need to assess the degree of similarity between sections of DNA often belonging to different species or genes. Now there are many biological sequence database systems to store genetic information, e.g., Genbank and EMBL, and they are expanding very fast. The Genbank and EMBL contain 186×10^6 nucleotide organized in 143×10^3 entries. The average length of the sequence is about 1,000 and the longest one is 172,282 [4]. The study of these biological sequences using sequence alignment has provided insights into topics such as disease and heredity. An example is the discovery of similarities between human growth factors and cancer-causing genes that may show how the genes cause uncontrolled cell growth [19].

The multiple sequence alignment is usually based on alignments of pairs of sequences. In this thesis, we are only interested in alignments of a pair of sequences. The

basic algorithms for this problem all use the dynamic programming technique. Thus, an alignment of two (long) DNA sequences may require extensive computation since the time complexity of these algorithms is $\Theta(m \cdot n)$, where m and n denote the lengths of two input sequences, respectively. For example, we have to spend 17 hours to find 20 local alignments between a 73,360-nucleotide sequence, containing human beta-like globin cluster, and a 44,594-nucleotide sequence, containing rabbit beta-like globin cluster, on a SUN-4/280 computer. To achieve a better running time, one may look for new algorithms, or use more powerful computers. Our study is to investigate the efficiency of the transputer system in solving the sequence alignment problem.

A multi-transputer system (or transputer system, for simplicity) is a popular coarse-grain MIMD multiprocessor system. Its building block, transputer, has build-in communication links and large memory. The goal of our study has two folds. One is to attempt to solve the sequence alignment problem in shorter time; The other is to study the transputer system architecture to see if it is suitable for implementing dynamic programming kind of algorithms.

There are seven chapters in this thesis. In the second chapter we formally define the sequence alignment problem and give a literature survey in this area. A summary of our results is also given. Chapter 3 introduces the platform of our study. Chapter 4 introduces the local sequence alignment algorithm used in our study. Chapter 5 gives an efficient implementation of the algorithm on the transputer system. Chapter 6 provides some empirical results and discussion on them. Chapter 7 is the conclusion of this thesis.

CHAPTER 2

THE SEQUENCE ALIGNMENT PROBLEM AND LITERATURE SURVEY

In this chapter, we formally define the sequence alignment problem and survey the previous works on efficient algorithms for the problem. From now on, let Σ be a fixed alphabet. (For DNA sequences, Σ = { A(adenine), C(cytosine), G(guanine), T(thymine) }.) By a sequence or string, we mean a sequence of characters from Σ.

We first give an overall idea of sequence alignments. An alignment between two sequences (or strings), as illustrated in figure 2.1(a), consists of a matrix of two rows. The upper row consists of the source sequence X, possibly interspersed with null symbol, which is represented by a blank. The lower row consists of the target sequence Y, possibly interspersed with null symbol too. The column $\begin{bmatrix} \\ b \end{bmatrix}$ is a deletion and the column $\begin{bmatrix} a \\ \\ \end{bmatrix}$ is insertion. The column $\begin{bmatrix} a \\ b \end{bmatrix}$ is the match, if a = b, or mismatch, if a $\neq$ b. The column $\begin{bmatrix} \\ \end{bmatrix}$ is not permitted.

Actually, two versions of the sequence alignment problem are of importance: the global sequence alignment problem, aiming at finding an alignment between two full input sequences, and the local sequence alignment problem, aiming at find segments (i.e., subsequences) of the two input sequences that can be well aligned. The global sequence

alignment problem is of interest when protein evolution is being studied. Since the genome is a mosaic of variously sized blocks of DNA, to detect evolutionary relationships and important homologies, it is essential to search for well-matched the segments in the given sequences, i.e., to find a best local alignment. The global sequence alignment problem is actually well-known in computer science, but under a different name — string edit. In the following, we define the string edit problem and local sequence alignment problem separately.

2.1 String Edit

Assume all symbols in the strings are from the fixed alphabet Σ. Given two strings X, of length m, and Y, of length n, and edit operations

a) insert symbol *a* into the string X with cost I_a,

b) delete symbol *a* from the string X with cost D_a,

c) replace symbol *a* of the string X by symbol *b* in Y with cost $R_{a,b}$,

we want to transform string X into string Y in minimum editing cost. For example, let strings X = *abaa* and Y = *abba*, and operations $I_a = 2$, $I_b = 3$, $D_a = 1$, $D_b = 1$, $R_{a,b} = 2$ and $R_{b,a} = 1$. We can transform X into Y in two ways:

```
| a b a a |
| a b b a |
```

(1)

```
| a b   a a |
| a b b a   |
```

(2)

The first transformation cost is $R_{a,b} = 1$ and the second transformation cost is $I_b + D_a = 4$. The first transformation is better.

The string edit problem has applications to file comparisons and revisions maintenance. For example, there may be several versions of a same computer program, and, if the versions are similar and all of them need be stored, it is more efficient to store only the differences than to store all versions. In this case, we need find differences between the versions first. This is essentially a string edit problem with uniform edit costs.

2.2 Local Sequence Alignment

A local sequence alignment of two biological sequences is an alignment found in some conserved regions of sequences. That is, a local sequence alignment is an alignment between two subsequences, one from each given sequence. In the local alignment problem, the subsequences are selected to maximize the similarity score, which is a kind of measure of the quality of the alignment. The alignments in figure 2.1 (c) are two typical local sequence alignments. The numbers outside the matrices indicate the starting position of each subsequence in its parent sequence. For convenience, we first give some necessary terminology here. Then we define the local sequence alignment problem.

Formally, let nil be a unique symbol for null sequence, i.e., the sequence with zero character. Denote an aligned pair as <a,b>, where a and b can be any character from the alphabet Σ or nil. An alignment is a finite sequence of aligned pairs. In each aligned pair, the first symbol is from the first sequence and the second symbol is from the second

sequence. So

<a, a> <nil, c> <b, b> <g, nil>

denotes an alignment of sequence *abg* and sequence *acb*. An aligned pair with nil as its first element is called an insertion pair and an aligned pair with nil as its second element is called a deletion pair. A consecutive sequence of insertion pairs is called an insertion gap and a consecutive sequence of deletion pairs is called a deletion gap.

Since deleting character *a* from sequence X can be considered as inserting character *a* into sequence Y, we treat an insertion and a deletion as an *indel*, or an extend-gap. So, the score for an *indel* is called an extend-gap-penalty. Another meaningful penalty is the open-gap-penalty. The open-gap means that, in the alignment, a non-deletion pair is followed by a deletion pair, or a non-insertion pair is followed by an insertion pair. The former is called an open-deletion-gap and the later is called an open-insertion-gap. The open-gap-penalty may be assessed as a barrier to allowing the gap. That is no gap would be allowed in the local alignment unless the benefit of allowing that gap would exceed the barrier. Figure 2.1 (b) and (c) are two examples of local alignments of sequences SEQ3 = *aaagctaacgtac* and SEQ4 = *aagtacg* using different scores. The open-gap-penalty in (c) is smaller than that in (b). In fact, the benefit gained by allowing that gap exceeds the open-gap-penalty in (c). So the best local sequence alignment in (c) is longer than the local sequence alignment in (b).

A score is assigned to an alignment based on a user-specified scoring function WEIGHT(a, b) and the open-gap-penalty. The range of WEIGHT is divided in to 3

classes of values:

$$\text{WEIGHT}(a,b) = \begin{cases} \text{match-values} & a=b \ \&\ a,b \in \Sigma; \\ \text{mismatch-values} & a\neq b \ \&\ a,b \in \Sigma; \\ \text{extend-gap-penalties} & a=\text{nil} \ \&\ b\neq\text{nil or } b=\text{nil} \ \&\ a\neq\text{nil} \end{cases}$$

In the WEIGHT function, match-value is greater than zero and the others are less or equal to zero. An insertion gap or deletion gap with length of k is scored as the sum of one open-gap-penalty and k extend-gap-penalties.

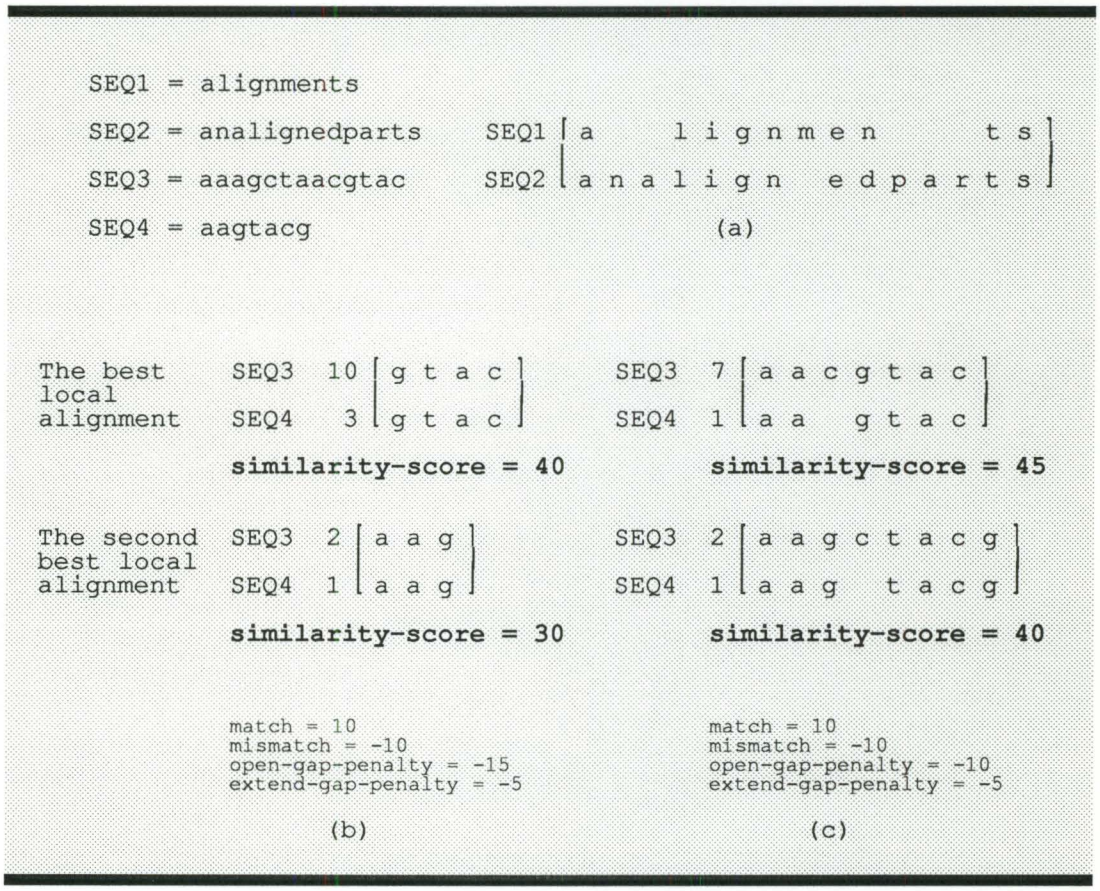

Figure 2.1 Examples of sequence alignment

(a) is a global sequence alignment; (b) is an example of 2 best non-intersecting local sequence alignments. (c) is another 2 best non-intersecting local sequence alignments. Note open-gap-penalty are different in (b) and (c).

Formally, we can state the local sequence alignment problem as: given two sequences X and Y, the operation score function WEIGHT(a,b) and the open-gap-penalty, find a local sequence alignment with the highest similarity score.

A natural extension of this problem is the k best non-intersecting local alignment problem. Here, non-intersecting means that if an aligned pair appears in an alignment it will not appears in other alignments. After finding the best local alignment, people often also want the next largest scoring local alignment that does not intersect with the best one. Since intersecting alignments are too many and are very similar to each other, they tell us nothing new. So, people are interested in best non-intersecting alignments [14]. Two examples of the second best non-intersecting alignment are given in Figure 2.1 (b) and (c). The k best non-intersecting local alignment problem is defined here:

Given two sequences X and Y, the score function WEIGHT(a,b), and the open-gap-penalty, find k local alignments which have the k highest similarity scores and do not share a same aligned pair.

2.3 Previous Works

For the string edit problem, Wagner and Fischer obtained an O(m•n) time and space sequential dynamic programming algorithm[25], where m and n are the lengths of the two input strings respectively. Edinston and Wagner proposed an O(m) processor pipeline architecture for string edit which takes O(n+m) time. Ranka and Sahni found an algorithm on SIMD hypercube machine, which has $O(((n \bullet \log n)/P)^{0.5}+\log_2 n)$ time complexity when $n^2 \bullet P$ processors are available, and $O((n^{1.5}/P) \bullet (\log n)^{0.5})$ time when P^2

processors available[23]. Ibarra, Jiang and Wang showed if edit cost are discrete the string edit problem can be solved in O(m+n) time on a one-way linear iterative array using m+n nodes. In a more practical setting, Lipton and Lopresti proposed an algorithm solving the string edit problem on a (2•P-1)-node systolic array in O(m•n/P) time[18].

In the area of sequence alignments, Needleman and Wunsch applied the dynamic programming method to the sequence comparison problem in 1970 [22]. Their algorithm takes O(m•n) time to find an optimal global alignment using O(m•n) space. Later, Smith and Waterman gave an O(m•n) time algorithm for finding a pair of segments from two sequences with largest similarity score[24]. This is the first local alignment algorithm. Gotoh[10] introduced multiple-sized gaps into Smith & Waterman's algorithm. Huang and Miller proposed an O(m+n+K) space and $O(m \bullet n + \sum_{k} L^2)$ time algorithm[8] for finding k best non-intersecting local alignments, where K is the total length of the k local alignments. This algorithm still takes a lot of time to run when k is large (e.g., 50).

Edmiston, Core, Saltz and Smith[4] studied two algorithms, Needleman & Wunsch's and Smith & Waterman's, and implemented the algorithms on two multiprocessor systems, Intel's iPSC/1 Hypercube and Thinking Machine's Connection Machine (CM-I). The algorithm for Hypercube needs O(m•n/P) space, which may cause problems when very long sequences are used. The algorithm for the Connection Machine needs O(max(m,n)) space and O(m+n) time. But a problem occurs when min(m, n) is larger than the number of processor in the Connection Machine.

Lander and Mesirov[16] implemented a dynamic programming sequence alignment algorithm on Connection Machine CM-II for exhaustively comparing one protein with all

proteins in a database. The algorithm allocates one sequence to each processor and then broadcasts a selected sequence, Each processor compares its sequence with the broadcasted sequence. So, the space requirement is O(m), where m is the length of the resident sequence at each node. The problem of this algorithm is the loss of efficiency since the lengths of sequences are very different.

Arendt[2] studied a concurrent file system for parallel genome sequence comparison. He used the dynamic programming method and the k-tuple heuristic, which is developed by Wilbur and Lipman[28], to implement a sequence comparison program on iPSC/2. Intel CFS (concurrent file system) was used in his study as sequences input/output device. The program execution time on a 16-node iPSC/2 is sometimes faster than on a Cary X-MP in scalar mode.

The most recent work was done by Huang[6]. He introduced a parallel algorithm to find an optimal alignment. He put much effort on task partition among processors to balance the computation load and reduce space requirement. The algorithm takes $O((M+N)^2/P)$ time and $O((M+N)/P)$ space when $P \leq \max(M, N)$. Another paper of Huang, Miller, and Hardison presents an algorithm for finding k best non-intersecting local alignment algorithm on Intel's hypercube[9]. They used linear array topology to implement the algorithm and got an average speed-up around 10.

2.4 A Summary of Our Results

We have implemented the algorithm for finding k best non-intersecting local sequence alignments on a 16-node transputer system. To get the speed-up, also we moves

Huang's sequential program for the same problem to the single node of the transputer system. We have performed tests on 3 real DNA sequence pairs and 4 random sequence pairs using 8-node and 16-node configuration. Totally, 44 tests were done and the sequences' length range from $4 \bullet 10^3$ to $45 \bullet 10^3$. The results are compared with the results of Huang's sequential algorithm running on a single-node transputer system. The comparison shows that the average speed-up is around 11 on the 16-node transputer system and around 6.3 on the 8-node transputer system. The memory requirement is O(m+n) on each node.

Our parallel sequence alignment program greatly reduces the time for finding k best non-intersecting local sequence alignments. For instance, it takes 11 hours (CPU time) to find 5 optimal local sequence alignments for rabbit and human beta-like globin clusters on a SUN4/280, while our parallel program can solve this problem in less than two hours. According to our experiences, we believe that the transputer system is a good candidate for the sequence alignment problems using dynamic programming method. It provides a smoothly scalable performance by allowing for easy addition or deletion of nodes.

CHAPTER 3

THE MULTI-TRANSPUTER SYSTEM

Transputer is a microprocessor chip product of INMOS. The first 32-bit transputer was introduced in September 1985. Since the goal of the transputer design is to support the concurrent processing, it has got much attention and there are many systems using transputer as its processor. More often, many transputers are connected to form a multiprocessor system.

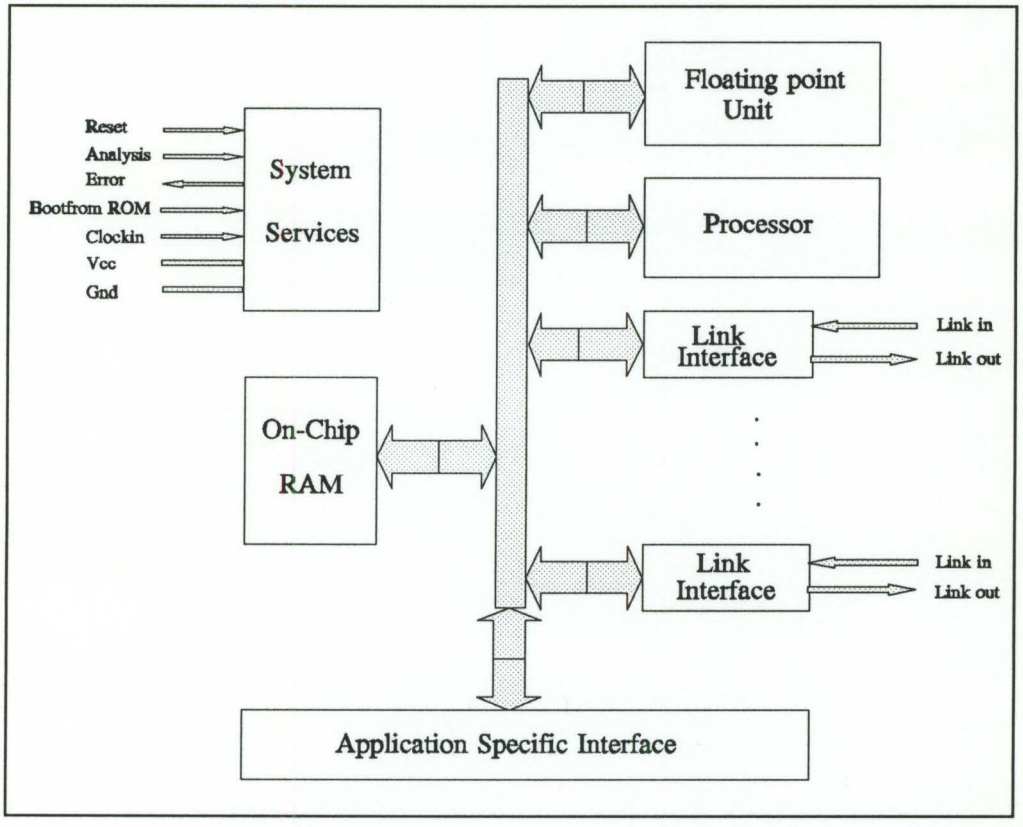

Figure 3.1 Transputer architecture.

3.1 The Transputer Architecture

Transputer is a microprocessor with links for connecting one transputer to another transputer and has a 4kB on-chip memory. Figure 3.1 is the transputer architecture [13]. Except the block of application specific interface, the other blocks in Figure 3.1 are on the same VLSI chip.

The CPU of a transputer is a 32-bit stack oriented processor. It has only three registers as the top three positions of the stack. The transputer instruction set supports high level programming languages such as OCCAM, C and Fortran. The latest transputer has an embedded the floating-point coprocessor on the chip, which improves the floating-point processing performance in graphic applications.

The on-chip memory is a special feature of the transputer. Since communication within device is much faster than between devices and memory is the most frequently accessed device, putting processor and memory on one chip improves the system performance. Transputer T800 has a 4 Kbyte on-chip RAM and its off-chip memory addressing space is 2^{32}- 1.

The communication links on the transputer are another important feature. To provide maximum speed with minimal wiring, the transputer uses point-to-point serial communication links for direct connection to other transputers. Each transputer link has two lines, one for input and one for output. A transputer has four links to connect with others. So, one transputer can directly connect to four transputers at most. It is very easily to organize transputers into two dimensional network, as shown in Figure 3.2. So, in this thesis, the transputer system denotes the multi-transputer system.

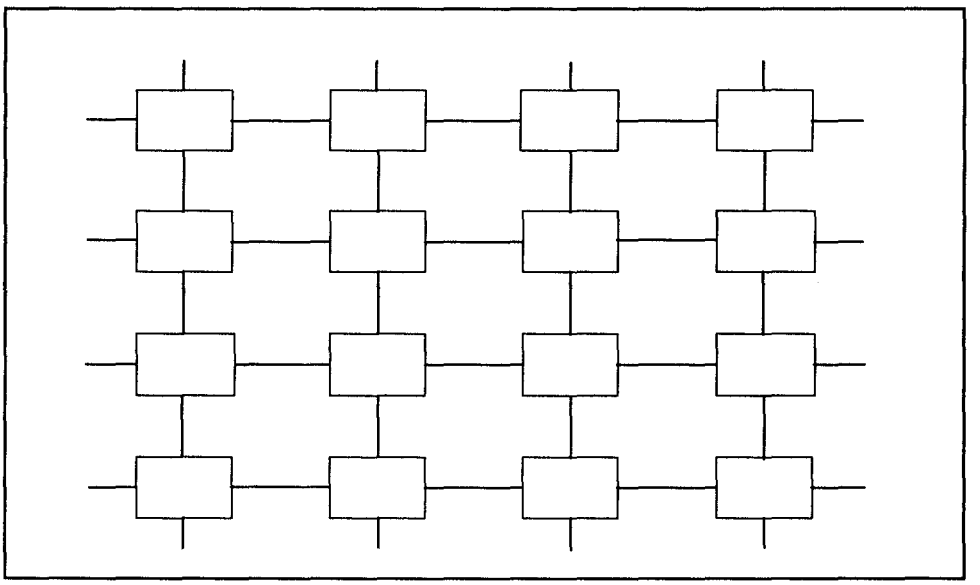

Figure 3.2 The Transputer Network

The ideal language for transputer system is OCCAM, a concurrent language based on the Communicating Sequential Processes. In OCCAM the assignment is indicated by ":=". The example

v := e

sets the value of variable v to value of expression e. The input is indicated by symbol "?". The example

c ? x

inputs a value from the channel c, assigns it to the variable x. The output is indicated by "!". The example

c ! e

outputs the value of the expression e to the channel c. If we use the terminology of communication, symbol "?" is to receive and symbol "!" is to send. OCCAM allows an application to be described as a collection of processes operating concurrently and communicating through channels which are implemented as links on the transputer. People can construct different systems depending on the processes of an application. For example, if the application is control system that has processes of input, output and computation, we can assign one process to a transputer. Then we use three transputers in this system. The input transputer receives input and sends data to the computation transputer through transputer link. The computation transputer sends the output data to the output transputer. The communications among processes are the communications among transputers.

3.2 Why Do We Choose Transputer System as the Platform

Transputer based multiprocessor system is a coarse-grain MIMD system. That is, each node of this system has a powerful processor and a large local memory. Several programs may run on the system at the same time.

According to the product Databook, T800 transputer can run at 30 MHz and provide 30 MIPS (peak) and 3.3 Mflops (peak) processing power. Its floating point unit is 64 bit wide, which can support graphics applications.

3.2.1 Flexible Connection

Transputer's hardware supported links offer flexible connection ability. Usually, the transputer system can be configured when booting the system. The configuration can

be defined by user or by system program. After booting, in the application program, user can define virtual circuits or data links to connect nodes according to special topology. In a medium or large application, the algorithm may consist of many parts and each part may need a special optimal topological network. For this kind of application, the transputer system is a good choice.

The sequence alignment algorithm that we are interested in is actually composed of several algorithms. Some of them behave well in the linear structure and some require the mesh structure. So, we prefer a transputer system because it can satisfy all these algorithms well. What we did is that we configured the transputer network to a mesh structure with special naming of nodes. Then to form a linear array, we used data link layer communication.

3.2.2 Message Passing on Transputer System

Message passing is one of the synchronization mechanisms on the concurrent process management. The basic idea of message passing is that when two processes need exchange information in the middle of the processing, the message passing commands are inserted where the exchange is planned to happen. At that point one process sends data and the other receives data. So, if one of the processes arrives at the point first, it must wait until the other process reaches the point. Then message passing occurs.

In a uni-processor computer the message passing is among the processes, while in the transputer system, the message passing may occur between processes on the same transputer or between processes on different transputers. The message passing technique is a well-understood mechanism and is widely used in multiprocessor systems, such as

Intel iPSC Hypercube and Thinking Machine CM/2.

The transputer system uses the message passing since its hardware is designed for message passing directly. The OCCAM language, designed for transputer, uses message passing as its basic synchronization mechanism.

Many languages exist on the commercial transputer system and the concept of the message passing mechanism is very intuitive. So it is not too hard to do programming on a transputer system. Since the communication network protocols have embedded in the languages on the transputer system, programmers can call these communication related functions to do message passing. The system will take care of message packaging, routing and buffering. There are many utilities on a transputer system, such as the utility for configuring topological layout and the debugger.

Most previous works on parallel sequence alignment were done on message passing system, e.g., CM/1 and iPSC hypercube. This means message passing is a reliable synchronization method.

As far as we know, there is no previous work related to sequence alignment on a transputer system. So, we choose transputer as the platform to investigate the efficiency of the transputer system in solving the sequence alignment problem.

3.3 The DCSS Transputer System

In Department of Computer Science and Systems, McMaster University, there is a 16-node transputer system running GENESYS on MACCS (SUN-4/280). The DCSS transputer system, outlined in Figure 3.3, is the product of TransTech Parallel Systems

Ltd., named as MCP1000. It is composed of four self-contained blocks which are called sites physically and NAS logically. Each site houses four T8 transputers, two of them are boundary nodes with 4 MByte memory and the other two are internal nodes with 2 Mbyte memory. Users can set the software controlled switches, implemented in 32x32 crossbar, on the MCP1000 board to construct the specific network topology required by their applications. Since each site is self-contained, it can be assigned to a user. Thus four users can work on the MCP1000 simultaneously.

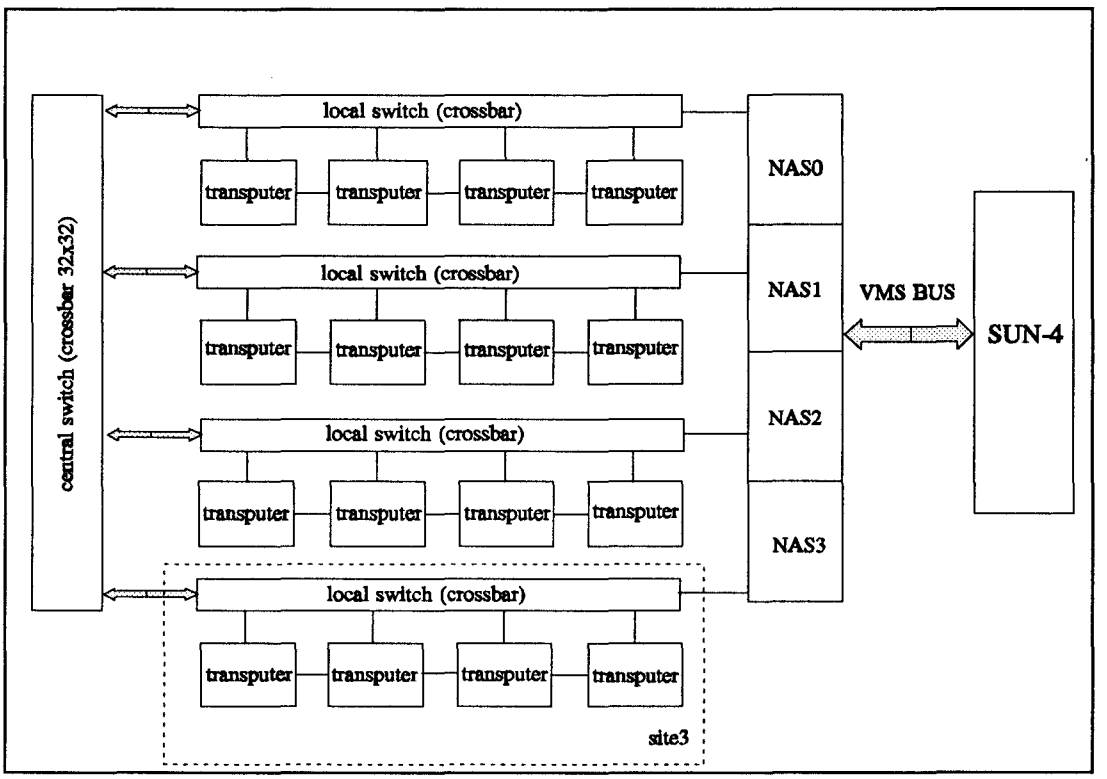

Figure 3.3 DCSS Transputer System

The supporting software for transputer system is GENESYS which is a concurrent operating system. It runs on the host and the transputer. Programming languages on transputer are Transputer C (tcc) and Transputer Fortran (tf77). Along with these languages, there are some procedure libraries providing transputer-oriented functions, e.g., communication related functions.

CHAPTER 4

THE ALGORITHMS FOR SEQUENCE ALIGNMENTS

In this chapter, we present the algorithms for solving the problems of string edit, finding the best local sequence alignment and finding k best non-intersecting local sequence alignments.

4.1 String Edit

The string edit problem has been defined before. Now let us consider how to solve this problem. Recall that the three allowed edit operations are:

(1) insert —— insert symbol *a* into the X with cost I_a;

(2) delete —— delete symbol *a* from the X with cost D_a;

(3) replace —— replace symbol *a* of the X by symbol *b* of the Y with cost $R_{a,b}$.

There are usually many ways to edit a string into another. For example, to change string *abbc* into the string *babb*, we can delete the first *a*, forming the string *bbc*, then insert an *a* between the two *b*'s yielding *babc*, and then replace the last *c* with a *b* for a total cost of $D_a+I_a+R_{c,b}$. However, we can also insert a new *b* at the beginning forming *babbc*, and then delete the last *c*, for a total cost of I_b+D_c. Our goal is to find a transformation from X to Y with the minimum cost. The basic technique to solve this problem is dynamic programming. Dynamic programming procedure builds on previous trial solutions to

generate a solution that satisfy the specified conditions.

We denote the string X as a one-dimensional array $X = a_1\ a_2\ ....\ a_m$ and string Y as a one-dimensional array $Y = b_1\ b_2\ .....\ b_n$. The matrix CC is a cost matrix and CC[i,j] denote the minimum cost of changing $a_1\ a_2\ ....\ a_i$ to $b_1\ b_2\ ....\ b_j$. Let us consider how to compute CC[i,j]. At point [i,j], the last operation which leads to the minimum cost can be delete, insert, replace or no operation (i.e., match).

delete: if delete a_i is the best change, then the minimum-cost transformation from $a_1\ a_2\ ...\ a_i$ to $b_1\ b_2\ ...\ b_j$ is the minimum-cost transformation from $a_1\ a_2\ ...\ a_{i-1}$ to $b_1\ b_2\ ...\ b_j$ plus one more deletion. In other words, $CC[i,j] = CC[i-1,j] + D_{ai}$.

insert: if insert b_j is the best change, then the minimum-cost transformation from $a_1\ a_2\ ...\ a_i$ to $b_1\ b_2\ ...\ b_j$ is the minimum-cost transformation from $a_1\ a_2\ ...\ a_i$ to $b_1\ b_2\ ...\ b_{j-1}$ plus one more insertion. In other words, $CC[i,j] = CC[i,j-1] + I_{bj}$.

replace: if replace a_i with b_j is the best change, then the minimum-cost transformation from $a_1\ a_2\ ...\ a_i$ to $b_1\ b_2\ ...\ b_j$ is the minimum-cost transformation from $a_1\ a_2\ ...\ a_{i-1}$ to $b_1\ b_2\ ...\ b_{j-1}$ and a replace. In other words, $CC[i,j] = CC[i-1,j-1] + R_{ai,bj}$.

match: if a_i matches b_j, $CC[i,j] = CC[i-1,j-1]$ since the cost of a match is zero.

Based on the above analysis, we have the formula:

$$CC[i,j] = \min \begin{cases} CC[i-1,j] + D_{ai} \\ CC[i,j-1] + I_{bj} \\ CC[i-1,j-1] + R_{ai,bj} \\ CC[i-1,j-1] \qquad \text{if } a_i = b_j \end{cases} \tag{4.1}$$

Figure 4.1 shows the dependency of CC[i,j].

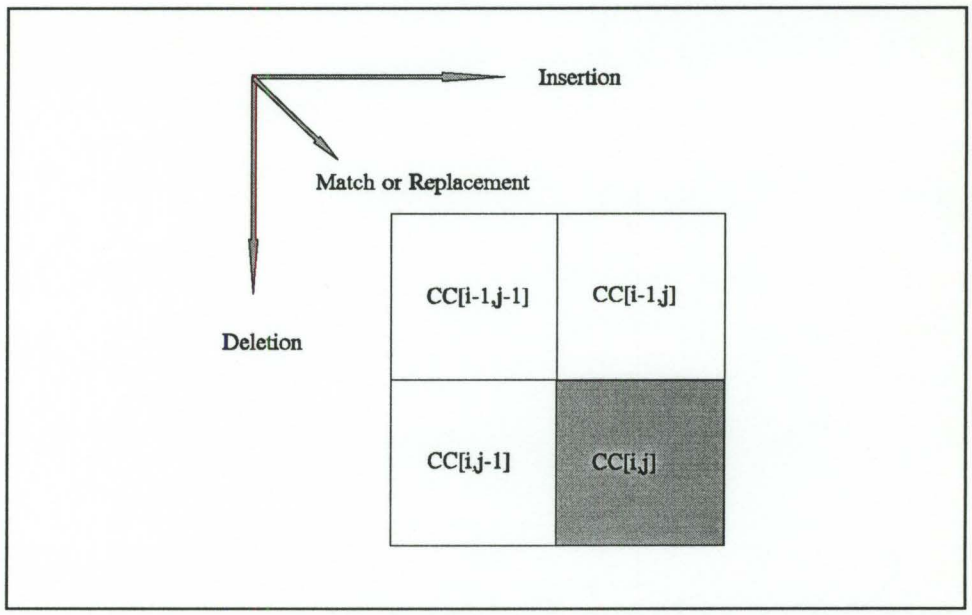

Figure 4.1 Dependency in matrix CC.

The algorithm for string edit is given in Figure 4.2. In the algorithm, we have to remember the best transformation from $a_1 \ldots a_i$ to $b_1 \ldots b_j$ for every i,j. M is a record matrix. M[i,j] stores the last transformation step which leads to position [i,j] with cost CC[i,j]. The complexity of this algorithm is O(m•n) in terms of space and time.

4.2 Local Sequence Alignment Algorithms

Now we consider the local sequence alignment problem, i.e., the problem of finding the best alignments between a segment of one sequence and a segment of another sequence. We will discuss two algorithms. They are Smith-Waterman's algorithm for

finding the best local sequence alignment and an algorithm for finding k best non-intersecting local sequence alignments.

```
Algorithm Optimal_Edit (X, m, Y, n);
Input : X (a string of size m) and Y (a string of size n);
Output: CC (the cost matrix) and M (the record matrix);
Begin
    for i:=0 to m do CC[i,0] :=i;
    for j:=1 to n do CC[0,j] :=j;
    for i:=1 to m do
        for j:=1 to n do
            x:=CC[i-1,j] + D_ai;
            y:=CC[i,j-1] + I_bj;
            if a_i = b_j then z:=CC[i-1,j-1]
            else    z:=CC[i-1,j-1] + R_ai,bj;
            CC[i,j] = min (x, y, z);
            if CC[i,j] = x then M[i,j] = -1;
            if CC[i,j] = y then M[i,j] = 1;
            if CC[i,j] = z then M[i,j] = 0;
    Print out transform using M;
end.
```

Figure 4.2 The String Edit Algorithm.

4.2.1 Smith-Waterman's Algorithm

Recall the definition of the local sequence alignment problem, the WEIGHT function, and the open-gap-penalty. The goal is to find a local alignment with the

maximum similarity score.

Smith-Waterman's algorithm also uses the dynamic programming technique to find the best local sequence alignment. A difficulty arised here since we have the open-gap-penalty, which is added to the score only at the opening of a (deletion or insertion) gap. To solve this problem, we need the history information of each move to determine if we should apply open-gap-penalty to the score at position [i,j]. So, several matrices are needed.

In Smith-Waterman's approach, three matrices are used. Assume again that the two input sequences have lengths m and n. Then the size of each matrix is (m+1)•(n+1). Matrix C is the alignment score matrix as before; Matrix D is a score matrix keeping track of the best alignments which end with a deletion pair, and matrix I is a score matrix keeping track of the best alignments which end with an insertion pair.

Let us see how to compute matrices C, I and D. C[i,j] is the maximum score of an alignment ending at point [i,j]. It may equal the value of I[i,j], if an alignment ending with a insertion pair has the maximum score, or D[i,j], if the alignment ended with a deletion pair has the largest score, or C[i-1,j-1] + WEIGHT(a_i,b_j), if a mismatch or a match leads to the largest score. Thus we have the formula:

$$C[i,j] = \max \{ I[i,j], D[i,j], C[i-1,j-1] + \text{WEIGHT}(a_i,b_j) \} \qquad (4.2)$$

Again, here, I[i,j] is the maximum score of an alignment ending with an insertion pair at position [i,j] and D[i,j] is the maximum score of an alignment ending with a deletion pair

at position [i,j].

At the position [i,j], I[i,j] may equal the value of I[i,j-1] plus an extend-gap-penalty (i.e., WEIGHT(nil,b_j)), if the next-to-last aligned pair is an insertion pair, or C[i,j-1] plus the sum of an open-gap-penalty and an extend-gap-penalty (i.e., WEIGHT(nil,b_j)), if the next-to-last aligned pair is a match, mismatch, or deletion pair. So, we have the formula:

$$I[i,j] = \max \{ I[i,j-1], C[i,j-1]+\text{open-gap-penalty} \} + \text{WEIGHT}(\text{nil},b_j) \quad (4.3)$$

Similiarly, D[i,j] may equal the value of D[i-1,j] plus an extend-gap-penalty (i.e., WEIGHT(a_i,nil)), if the next-to-last aligned pair is a deletion pair, or C[i-1,j] plus the sum of an open-gap-penalty and an extend-gap-penalty (i.e., WEIGHT(a_i,nil)), if the next-to-last aligned pair is a match, mismatch, or insertion pair. So, we have the formula:

$$D[i,j] = \max \{ D[i-1,j], C[i-1,j]+\text{open-gap-penalty} \} + \text{WEIGHT}(a_i,\text{nil}) \quad (4.4)$$

The relationship or dependency among C, I and D can be described by an alignment graph. Figure 4.3 shows the graphical explanation of the relationship among matrices C, I and D, when the input sequences are X = *ab* and Y = *ab*.

In Figure 4.3, the alignment graph has $3\bullet(m+1)\bullet(n+1)$ vertices denoted C[i,j], D[i,j] and I[i,j], where $0 \le i \le m$, $0 \le j \le n$. The edges represent the dependencies between vertices. The edges are divided into 7 classes:

1) delete edge D[i-1,j] →D[i,j], labelled <a_i,nil>, where 1≤i≤m, 0≤j≤n.

2) open delete edge C[i-1,j] →D[i,j], labelled<a_i,nil>, where 1≤i≤m, 0≤j≤n.

3) insert edge I[i,j-1] →I[i,j], labelled <nil,b_j>, where 0≤i≤m, 1≤j≤n.

4) open insert edge C[i,j-1] →I[i,j], labelled <nil,b_j>, where 0≤i≤m, 1≤j≤n.

5) replace edge C[i-1,j-1] →C[i,j], labelled <a_i,b_j>, where 1≤i≤m, 1≤j≤n.

6) null edge D[i,j] →C[i,j], where 0≤i≤m, 0≤j≤n.

7) null edge I[i,j] →C[i,j], where 0≤i≤m, 0≤j≤n.

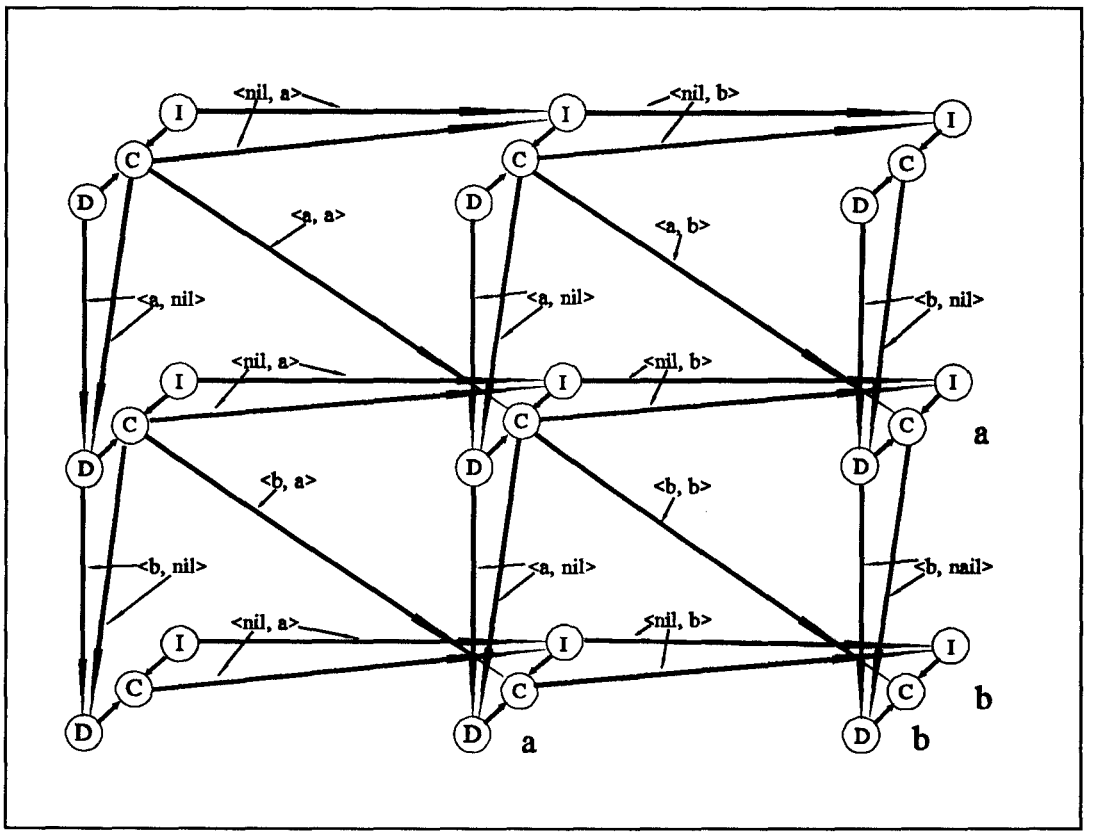

Figure 4.3 Graphical explanation of relations among C,

I and D on sequences *ab* and *ab*.

```
Algorithm Local_Alignment(X, m, Y, n)

    Output: score_max, i_best, j_best and the best local aligmment;
    Begin
        D[0,0] = I[0,0] = -∞
        C[0,0] = 0;
        score_max = i_best = j_best = 0;
        for j:=1 to n do
            D[0,j] := -∞;
            I[0,j] := WEIGHT(nil,b_j) - g;  /* g is the open-gap-penalty */
            C[0,j] := 0;
        for i:=1 to m do
            I[i,0] := -∞;
            D[i,0] := WEIGHT(a_i,nil) - g;
            C[i,0] := 0;
            for j:=1 to n do
              D[i,j] := max {D[i-1,j],C[i-1,j] - g} + WEIGHT(a_i,nil); set MD[i,j];
              I[i,j] := max {I[i,j-1],C[i,j-1] - g} + WEIGHT(nil,b_j); set MI[i,j];
              C[i,j] := max {0,D[i,j],I[i,j],C[i-1,j-1]+WEIGHT(a_i,b_j); set MC[i,j];
              if C[i,j] > score_max then
                  i_best := i;
                  j_best := j;
                  score_max := C[i,j];
        write "A best local alignment with score" score_max "ends at"
        (i_best,j_best);
        trace MI, MD and MC to display the best local alignment;
    end.
```

Figure 4.4 The Smith-Waterman Local Alignment Algorithm.

To summarize the above discussion, the recurrence formulas for computing C, I, D are

$$\begin{cases} C[i,j] = \max \{ I[i,j], D[i,j], C[i-1,j-1] + WEIGHT(a_i,b_j) \} \\ I[i,j] = \max \{ I[i,j-1], C[i,j-1]+\text{open-gap-penalty} \} + WEIGHT(nil,b_j) \\ D[i,j] = \max \{ D[i-1,j], C[i-1,j]+\text{open-gap-penalty} \} + WEIGHT(a_i,nil) \end{cases}$$

Figure 4.4 is the Smith-Waterman's local alignment algorithm. In the algorithm, matrices MC, MD, and MI are used to remember the path. The best local sequence alignment is displayed by tracing these three matrices backward. The space and time complexities of Smith-Waterman's algorithm are O(m•n).

4.2.2 The Linear Space K Best Non-intersecting Local Alignments Algorithm

The biggest drawback of Smith-Waterman's algorithm is the space complexity. When a program running on a computer, the running time is limited by the electronic components life time which can be very long, but the space of computer is limited by the memory size. Certainly, the virtual memory system can provide large memory space, but it involves mechanical operation which not only takes at least 10 times the memory access time but also increases the possibility of system errors.

A linear space algorithm for finding k best non-intersecting local alignments is given by Huang[8]. Let us see how to find just one best local alignment using linear space. Huang's approach is based on Smith-Waterman's algorithm and Myers-Miller's algorithm [21]. Since in the dynamic programming approach, the element C[i,j] of the matrix depends only on C[i-1,j-1], C[i-1,j] and C[i,j-1] and the computation is done row

by row, Huang uses three one dimension arrays C, I and D to save the row scores. In this way, the space requirement is reduced from 3•(m+1)•(n+1)+S to 3•(n+1)+S, where S is the extra space to store sequences and parameters.

Since this algorithm uses only linear space, it can not remember the actual best alignment when doing dynamic programming. Huang defined the starting point arrays E, F, V and W to remember the starting points of the best local alignments. The best local alignment at position [i,j] starts at [E[j], F[j]] and the best local alignment ending with a deletion at position [i,j] starts at [V[j], W[j]]. To remember the starting point of the best local alignment ending with an insertion, instead of arrays, a pair of varaibles are used because the computation order is row by row. By keeping trace of the position yielding the max score and starting point of the corresponding alignment, this algorithm can find the best local alignment's starting and ending points at the end of the dynamic programming. To display the result, Huang uses Myers-Miller's algorithm on the two subsequences to find an optimal global alignment in linear space. So, the space requirement for finding the best local alignment algorithm is O(m+n).

The algorithm of Myers and Miller for actually finding an optimal (global) alignment in linear space is a recursive algorithm using divide-and-conquer. Suppose that the input sequences are $X = a_1 \ldots a_m$ and $Y = b_1 \ldots b_n$, C, I and D will be the cost arrays as before for the upper-half matrix, and C′, I′ and D′ are the corresponding cost arrays for the lower-half matrix.

This algorithm begins with picking a mid-point of sequence X (midi). The arrays C, I and D are then computed in the area [1,midi]x[1,n] and the arrays C′, I′ and D′ are

computed in the area [midi+1,m]x[1,n] reversely, i.e., starting from position [m,n] to the position [midi+1,1]. Here, it uses Smith-Waterman's algorithm to find the maximum alignment scores in the two areas. (Because the algorithm aims at finding the position that the alignment pass, only cost arrays are needed.) At the boundary of two regions, the adjacent elements of D and D′ are added, and the adjacent elements of C and C′ are added, since only deletion, match and mismatch can cross the boundary of upper and lower regions. The mid-point of the sequence Y (midj) is where C + C′ or D + D′ - oper-gap-penalty is the largest. Thus, [midi, midj] is the point where the best alignment passes. At this point, the problem is split into two subproblems. One is finding the best global sequence alignment of the subsequences $a_1 \ldots a_{midi}$ and $b_1 \ldots b_{midj}$. The other is finding the best global sequence alignment of the subsequences $a_{midi+1} \ldots a_m$ and $b_{midj+1} \ldots b_n$. Then, the algorithm recursively finds middle point and splits the subproblem, until an input sequences of the subproblem has just one symbol. Figure 4.5 shows the splitting of a problem into subproblems.

To find the k best non-intersecting local sequence alignment, a list, denoted LIST, is used to save the k best alignments. Each element of LIST is a tuple defined as AL_tuple = <SCORE, START[i,j], i, j, T, B, L, R>, where

SCORE —— the score of the alignment;
START[i,j] - the start point of the alignment ending at [i,j];
[i, j] —— the end point of the alignment;
[T,B]x[L,R] - a region covering the alignments with the same starting point and scores greater than the minimum score in the current LIST, i.e., the region that is affected by the alignment starting from START[i,j].

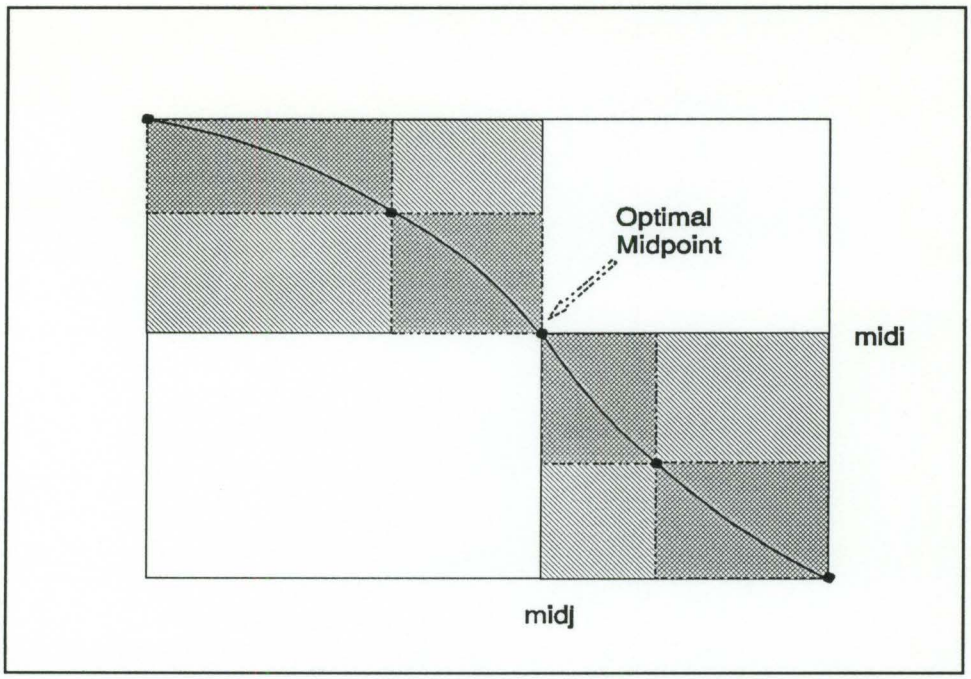

Figure 4.5 Splitting the problem into subproblems.

So, an element of LIST defines an alignment with its specific starting point and ending point, as well as a region which contains all alignments sharing the same starting point and having score greater than the smallest score in LIST.

LIST is maitained by the function adnd(score, st_pt, i, j, list_size) that creates an AL_tuple with score, start point, end point, and the boundary values of T, B, L, R, in case the score of the alignment is greater than the minimum score in the current LIST. If the alignment at current position has score greater than the minimum score in LIST and the entry with same starting point exists in LIST, function adnd modify the all data in the AL_tuple except starting point. So, each entry of LIST has a different starting point. In the meantime, the function adnd() keeps track of the minimum score in LIST.

In Huang's algorithm, the AL_tuples are filled with alignments found using Smith-Waterman's algorithm and the function adnd(). When finishing the last row of C matrix, k promising alignments are saved in LIST. These alignments must be non-intersecting because each entry of LIST has a unique starting point. If two alignments intersect at position [i,j], at most one of them can survive and enter LIST, due to the definition of adnd(). After finding k promising local alignments, the alignment with maximum score is the best local alignment. Then the Myers-Miller's recursive divide-and-conquer algorithm is called to find the aligned pairs in the best local alignment.

The rest k-1 alignments in LIST may not include the second best alignment since the best alignment may mask the second best. We can recompute the C matrix. However, it may cost too much because only a small region is really needed to recompute. Huang's algorithm has a function to find this region, which is called the masked region.

The condition of non-intersecting is secured by a linked list of used aligned pairs that remembers those already output. This list is expended when the aligned pairs are displayed.

Huang's linear space local alignment algorithm is outlined in Figure 4.6. (Note that the real implementation is different than the outline; it uses arrays instead of matrices.) The algorithm has four parts. First, using dynamic programming technique, find the k promising alignments; Second, display the best alignment and fill the used aligned pairs list; Third, find the masked region that needs recomputed; Fourth, if there is any promising alignment in this region, recompute this region.

Function Fisrt(i,j) can find the starting point of the alignment ending at position [i,j]. As we mentioned the above, this funtion is implemented by using several arrays. Function maxtuple() finds the AL_tuple with largest similarity score in LIST.

```
Algorithm k_best_alignment (X, m, Y, n, k)
Output: k best non-intersecting alignments
Begin
        min_score := 0;
        for i:=0 to m do
            for j:=0 to n do
                compute C[i,j] and First(i,j);
                if C[i,j] > min_score then
                        min_score := adnd(C[i,j], First(i,j),i,j,k);
        for r:=1 to k do
            S := maxtuple();
            display_alignment(S);
            if r != k then    /*  T' -- Top, B -- Bottom, L' -- Left, R - Right.  */
                Determine the affected region [T', B] x [L', R] in which there
                are alignments with score greater than min_score;
                for i:= T' to B do
                        for j:= L' to R do
                                Compute C[i,j] and First(i,j);
                                if C[i,j] > min_score then
                                        min_score = adnd(C[i,j],First(i,j),i,j,k-r);
end.
```

Figure 4.6 Outline of Huang's Linear Space Alignment Algorithm.

As we mentioned above, the best local alignment may mask some other alignments. Let us see how it could happen. For some i,j, the alignment found at [i,j] (with score C[i,j]) may intersect the the best local alignment. Thus, other alignments (with scores less than or equal to C[i,j]) are not recorded in LIST. These other alignments are masked by the best local alignment. Recall the meaning of the [T,B]x[L,R] region. We can say that the region [T,B]x[L,R] covers the masked alignments or part of the masked alignments. There are four cases of the coverage (refer to the Figure 4.7).

Case 1: The masked alignment is totally covered.

Case 2: The part of the masked alignment, from the middle to the end, is covered.

Case 3: The part of the masked alignment, from the start to the middle, is covered.

Case 4: The middle part of the masked alignment is covered.

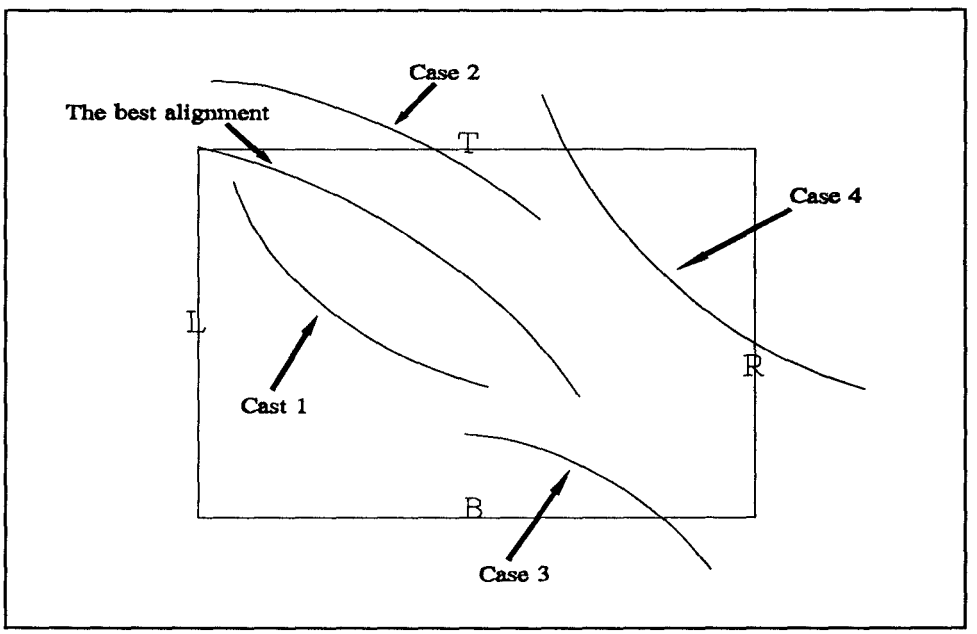

Figure 4.7 The masked region and the masked alignments

Actually, case 3 and 4 will not happen. In case 3, if the alignment could extend longer, those part would be included in [T,B]x[L,R] region, since the similarity score may increase and be greater than the minimum score of the LIST. For the same reason, the case 4 can not happen. Now, we know that the lower and the right boundaries of the masked region are B and R. The only task left is to find the upper and left boundaries of the masked region. To find the upper and left boundaries of the masked region, Huang uses Smith-Waterman's algorithm in the region [T,B]x[L,R] reversely, and extends the T and L boundaries, until there is no alignment starting outside the current region [T,B]x[L,R] and ending inside the region.

Figure 4.8 is a graphical explanation of finding the masked region. The dark shaded area in the Figure is the original [T,B]x[L,R] region. When the above process reaches the upper-left corner of [T,B]x[L,R], the alignment S′ is still extending. So, the T and L are decreased, until the score of S′ is less than the minimun score of the current LIST at position [T′,L′]. Now, the masked region is [T′,B]x[L′,R]. Note that, the alignment S″ starts and ends outside the region [T′,B]x[L′,R]. Thus it does not affect the reverse computation.

Up to now, we know how to find the k best non-intersecting local sequence alignments in linear space. We will discuss the implementation of this algorithm on a multi-transputer system in the next chapter.

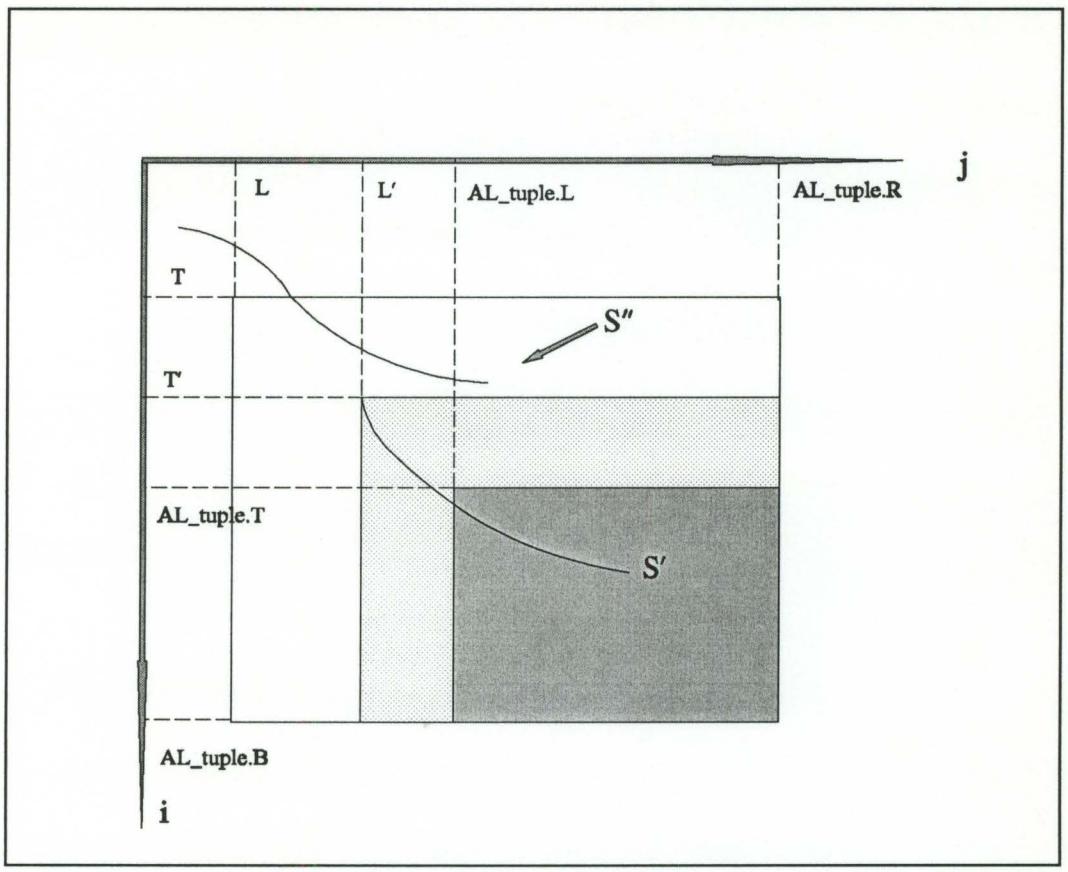

Figure 4.8 A graphical explanation of finding the masked region

CHAPTER 5

PROGRAMMING ON THE MULTI-TRANSPUTER SYSTEM

In this chapter we present our main considerations in mapping Huang's sequential algorithm for finding k best local alignments to a parallel system. We mainly concentrate on the parallelization of the algorithm to create an efficient implementation.

5.1 The Transputer Sequence Alignment Program

A transputer program usually has two subprograms OTB and ITB. OTB (Outside The Box) is a host program running on the host computer and ITB (In The Box) is a real transputer program running on the computing nodes.

OTB is a user interface of the transputer program. It accesses file system for input, supports user interface, and prints out alignments. ITB is the main part of the program. It has six sections. Please refer to Figure 5.1 which outlines our program structure. We use node_0 as the leader of all computing nodes. The head file list is in Appendix I, the ITB transputer program list is in Appendix II, and the OTB program is in Appendix I.

The first section uses the dynamic programming approach to find k promising alignments. All computing nodes are involved in this process and each node uses neighbourhood communication only, i.e., each node receives data from one neighbour and sends its result to another neighbour after computing.

```
OTB Program
begin
        input parameters;
        input two sequences;
        prepare messages;
        send messages to all nodes in transputer network;
        for i=1 to k do
                receive alignment and display it on screen
end;

ITB Program
begin
    receive messages from host;
    receive two sequences from host;
    compute k best promising alignments and save them in LIST;
    for i=1 to k do
        construct a global LIST in node_0 and find max alignment with max score;
        recursively find optimal alignment;
        send alignment to host;
        mark all used aligned pairs and inform the other nodes;
        find masked region;
        if there is promising alignment then
            compute k-i best promising alignments in the region of [T',B]x[L',R]
            and modify LIST;
end;
```

Figure 5.1 Outline of transputer program

The second section merges the local LIST to the global LIST in node_0, finds the maximum score and maintains the LIST of k promising alignments. On the first sight this section is sequential, but we use two-way merge to boost the concurrency.

The third section finds the aligned pairs using Myers and Miller's algorithm. This divide-and-conquer algorithm is implemented in a recursive fashion, which allows the system to run at its top speed and all node work in parallel in the ideal case. At the meantime, the aligned pairs are marked using a linked list.

The fourth section does book-keeping works. In each computing node, there is a used-aligned pair list. When an aligned pair is sent to OTB, it is added to the used-aligned pair list. In this section, nodes exchange information about the used-aligned pair list for the later reference.

The fifth section finds the masked region of previous output alignment. It involves dynamic programming too and this part is somewhat similar to section one. The different point is that after finishing the rectangle, several rows or several columns are added. Those operations are sequential. Please refer to Figure 4.8.

The sixth section recomputes the masked region. In this section, the first division of program applies to a small region.

Data structures used in the parallel program are several arrays to save scores and starting points, a list to save k promising local alignments and a list to save aligned pairs. The list for saving k promising local alignments and arrays were defined in the previous chapter. The aligned pair list is actually a set of lists. It is implemented as a pointer array. Each element corresponds to a position in input sequence X. When an aligned pair is

outputted, an list element, that contains a position in input sequence Y and a pointer, is attached to a list according to the first element of the aligned pair.

5.2 Selection of the Network Topology

The computer network topology affects the efficiency of the system solving a specific problem. For example, image processing is better on mesh connected system rather than on a ring system.

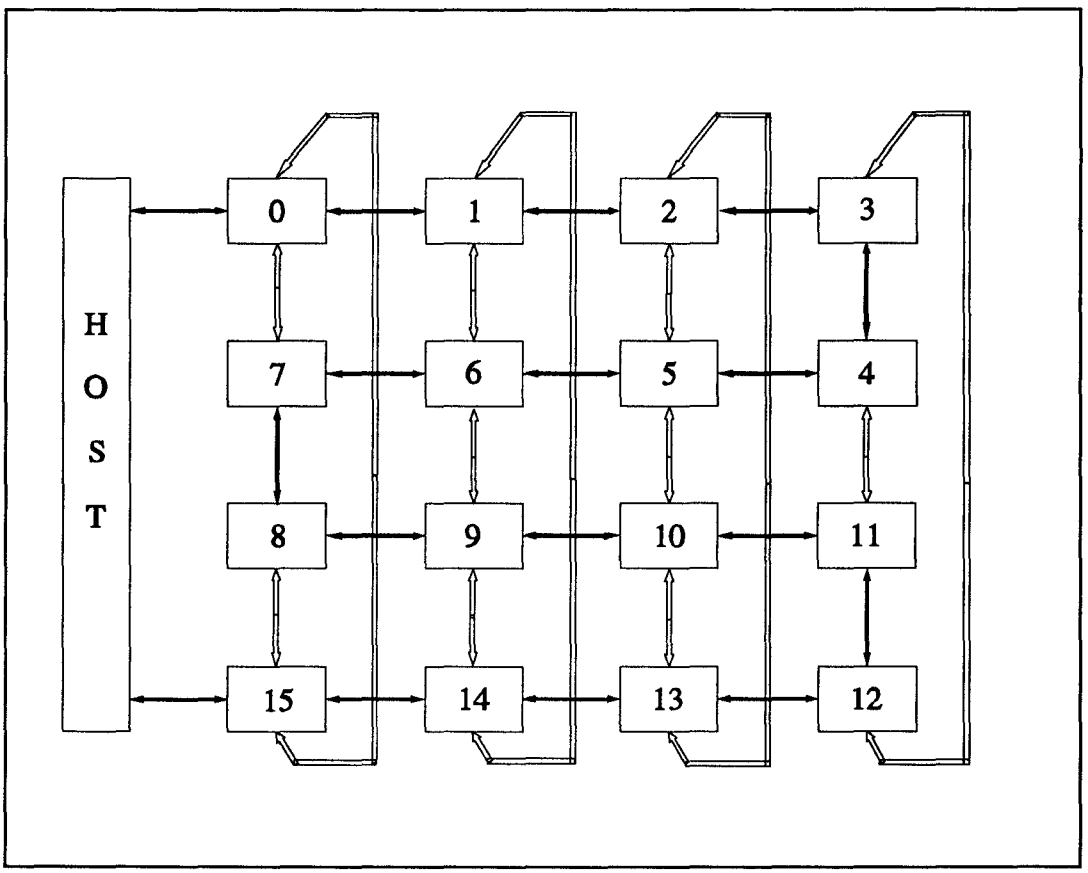

Figure 5.2 DCSS transputer system under my configuration.

Basically, there are four main topological choices: mesh, ring, linear array and star. Our DCSS transputer system can be configured in any one of the four structures due to the connection flexibility of the transputer. Which one is good for the sequence alignment problem?

Ring, linear array and star are simple networks and require a small number of links, which may lead to a efficient communication since the reduction of routing and buffering, while mesh connection offers a powerful and flexible system. In the sequence alignment problem, linear connection seems to work better when doing dynamic programming, and mesh connection is better for doing sorting and recursion. So, we select mesh connection and use data link layer commands to form linear connection when doing dynamic programming.

5.3 Parallelization

So the topology we have chosen is mesh. Figure 5.2 shows the interconnection between nodes and the host under my configuration. We number the node differently from the default since we plan to use linear structure to do dynamic programming. In Figure 5.2, the dark lines connect the 16 nodes as a linear array. Below, we discuss all the problems related to boosting concurrency.

5.3.1 Task and Data Allocation

The task and data allocation is a key issue when programming parallel computer systems. The basic rule suggests partitioning along data dependencies without cutting dependencies, which in turn reduces the amount of communication between partitions[29].

The objective of task allocation is to reduce communication overhead and the idle time for each node.

The main part of the k best non-intersecting local alignment algorithm is a dynamic programming program which uses a matrix as its platform. We can cut the matrix into squares or slices and assign squares or slices to each transputer. If we cut the matrix into squares, we can implement the parallel algorithm in two ways. Suppose that we have 4 processors T0, T1, T2 and T3 (refer to Figure 5.3).

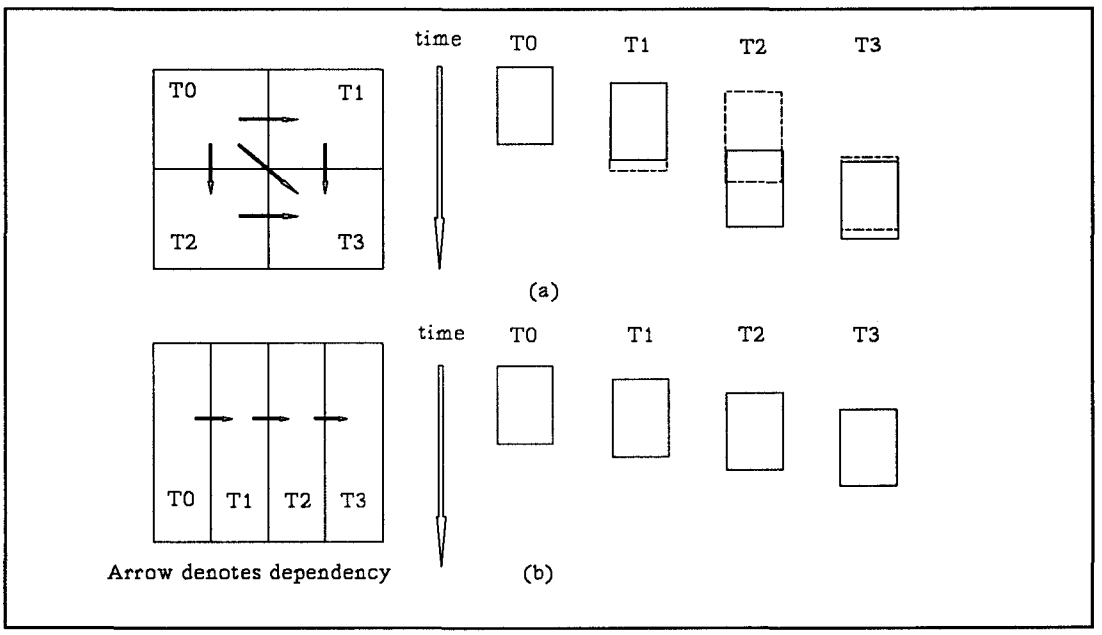

Figure 5.3 Different partitioning strategies for the problem

The first approach is to follow the row-by-row order, i.e., T0 processes its square row by row, and at the end of each row, T0 sends data to T1. Then T1 can start to work. When T0 works on the last row, T2 can start. After T2 finishes one row and T1 works on its last row, T3 can start. Let us estimate the wait time of T3. Suppose that the matrix

is m•m and each element needs one step to process. T3 has to wait m•n/4 steps. So, if we do not consider the impact of communication, the run time is m•n/4 + m•n/4 steps. Figure 5.3 (a) shows the parallelism by rectangles in solid line.

The second approach is to follow the diagonal order, i.e., T0 processes its matrix one row and then one column alternatively. In this order, T3 will start working after T1 and T2 reach the last row and last column. The wait time for T3 is m•n/4 steps since T0 has to finish the last element of its sub-matrix before T1 and T2 can start their last row and last column. So the run time of this approach is also m•n/4 + m•n/4 steps. Figure 5.3 (a) shows parallelism by rectangles in dot line.

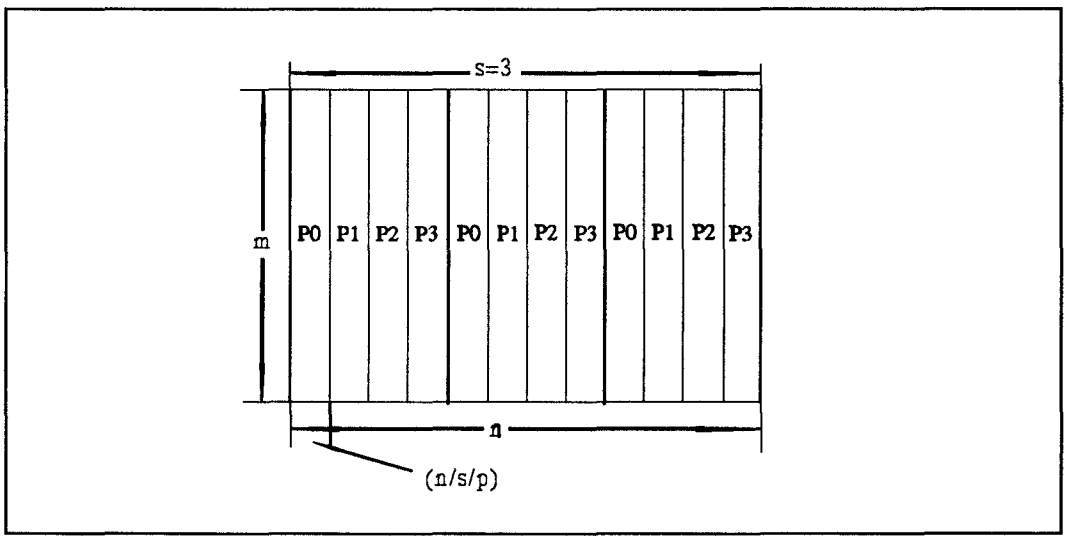

Figure 5.4 Example of task allocation for a 4-processor system.

Another possibility is that we cut the matrix into slices, as shown in Figure 5.3 (b). After T0 finishes a quarter of the first row, T1 starts working. T3 starts working after three quarters of the first row are finished. Then the wait time for T3 is 3•m/4 steps. So,

the run time is 3•m/4 + m•n/4 steps, which is better than cutting the matrix into squares. Thus we choose to cut the matrix into slices.

Now an issue is how many slices we should have. Suppose we have p transputers, we can cut matrix to p or s•p slices, where s is an integer that represents the number of sections. Each section contains p slices. It seems that the more the sections the partition has, the less the setup time is involved in the computation. Assume that

T_c —— the computation time for one step;

T_{com} —— the communication time for sending one message;

n, m —— lengths of the two sequences;

s —— number of sections that the score matrix is divided into;

p —— number of transputer nodes in the system.

We cut this matrix into s sections, i.e., totally s•p slices, and assign a slice to a node in a round-ribbon fashion, as shown in Figure 5.4. In the Figure, again we assume that there are 4 processors.

According to this job partition among the transputer nodes, the system has a set-up stage, i.e., the period between the time when the first node starts working to the time when the last node starts working, and a clean-up stage, i.e., the period from the termination of the first node to the termination of the last node (refer to Figure 5.5). The total program run time, including communication and computation, is

$$\frac{(n/s)}{p} \bullet (p-1) \bullet T_c + \frac{n}{p} \bullet m \bullet T_c + s \bullet m \bullet T_{com} + (p-1) \bullet T_{com}$$

The first two terms are computation related and the other two are communication related. The first term is the computation set-up time; The second term is the parallel computation time; The third term is the communication time in parallel computation; The last term is the communication time in the computation set-up stages. Figure 5.5 is the interpretation of the run time estimation.

We want to find how many sections, s, the matrix should be divided into to keep the program run time the smallest. Clearly, $1 \leq s \leq n/p$. There are two terms involving s. When s=1, i.e., each node has one slice, the third term becomes the smallest and first term goes to its maximum. When s = n/p, i.e., each node has s slices and each slice has only one column, the third term reaches to its largest value and the first term goes to its minimum. By looking at the sequential program and the transputer Databook, we can estimate T_c and T_{com}. In the sequential program, there are about 350 machine instructions. We assume that the average instruction time is 2.25 machine cycles. So

$$T_c \approx 800 \text{ machine cycles.}$$

The preparation of data for sending a message needs about 40 instructions (i.e., 90 machine cycles) and the transputer instructions IN and OUT need 50 machine cycles each. So

$$T_{com} \approx 200 \text{ machine cycles.}$$

So, $T_c = 4 \bullet T_{com}$. We pick out the two terms involving s below.

$$\frac{n \bullet (p-1) \bullet T_c}{p \bullet s} + s \bullet m \bullet T_{com}$$

$$\approx \left(4 \bullet \frac{n}{s} + m \bullet s \right) \bullet T_{com}$$

To keep formula (5.2) the smallest, we should set s to $(4 \bullet n/m)^{0.5}$. Since m and n usually have the same magnitude, for simplicity, we will choose s to be 1.

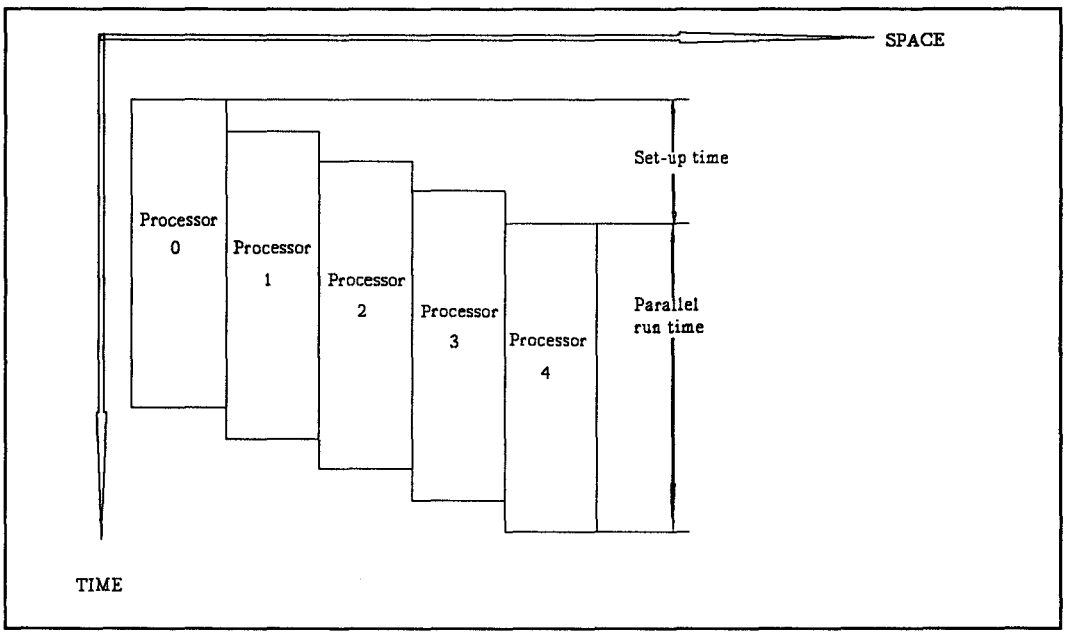

Figure 5.5 Interpretation of run time estimation.

5.3.2 Load Balancing

The second part of the program solves a merge problem on a multi-processor. That is, we want to merge p local k promising alignments LISTs to construct the global k-promising-alignments LIST in node_0. The intuitive way is to insert each tuple of one

local LIST to another local LIST, until there is one LIST left. This final LIST is the global LIST. For example, suppose that we have three transputers N0, N1 and N2. N2 sends its list to N1. N1 inserts the received elements into its own list, then sends resulting list to N0. After inserting the list from N1, N0 holds the global k promising alignment list. This method is simple, but is sequential in nature. When one node is working, the others have to wait. Another approach is the reverse-binary-tree-merge, i.e., we suppose that the local LISTs are the leaves of a tree. The merge starts by merging each pair of leaves and then the resulting lists, until the root is reached. Using this method, it takes only $\log_2(p)$ steps, while the sequential method takes p steps. In our system there are 16 transputers, i.e., p = 16. The parallel merge takes 4 steps and keeps most nodes working at most of the time. In general, keeping load as even as possible is one important aspect in the design of parallel programs.

5.3.3 Reduction of the Communication Overhead

As we choose the partition with more dependencies and more parallelism, we also need to reduce the communication overhead. Although a transputer has the communication hardware embedded on the chip, the communication impact can not be ignored. The communication instructions OUT and IN take 19 + 2•W machine cycles to execute, where W is the number of the words in the message. This does not include the sending time. Since transputer links are serial ports, sending a message needs machine cycles equal to the number of bit in the message. Certainly, the sending of a message may be overlapped with other instructions. Also, here we do not take into account of the three communication protocol layers: logical, data link and network, which route and buffer

messages. Sometimes, the data type conversion may be involved in the network layer.

Generally speaking, there are two factors that have direct effect on communication time. One is the size of the message. The other is the number of the messages (due to protocol overhead), i.e., how many times an application program calls the communication function which may in turn call routing and buffering. In our program we try to reduce the effect of these two factors. To reduce the first factor, we decided that the messages passing through the computing nodes will have a message header only. The method to reduce the effect of the second factor will be discussed later.

In transputer C, the facility for communication is data structure of a message and functions Send and Receive. The message data structure is a structure in C language. It has the form shown below.

```
struct nmsg {
int     nh_dl_event,        /* datalink event      */
        nh_node,            /* destination node    */
        nh_event,           /* message event       */
        nh_type,            /* message type        */
        nh_length,          /* length of message   */
        nh_flags,           /* option flags        */
        nh_data[8];         /* data pouch          */
char    *nh_msg;            /* pointer to message buffer      */
}
```

Here, nh_event and nh_type form the identification of a message; nh_dl_event is used to specify the port (link) for an outgoing message when data-link communication is used;

nh_flag specifies the data types of the message and data pouch; nh_msg is a pointer to the buffer where the message is stored. A message consists of two parts, the header, which is the structure described above, and the message buffer. The header is the essential part of the message. The message buffer contains the actual message known as the body of the message. To send a message, a node has to fill in the above header and the message buffer first, then call function Send using this header as the parameter.

In our program, we try to use message header only. However, eight integers are not enough to pass the information required by computation. To solve this problem we compressed the required data to eight integers before it is sent and uncompress them after receiving them. Since the operations compression and uncompression are bit-oriented operations, they take less time than using the message body instead.

The second factor on communication time is fixed after the partition of the problem. However, we can use broadcast to reduce the number of messages needed. In the broadcast mode, a message is propagated to all nodes in the system. On the other hand, if broadcasting is not used, some nodes may have to pass the same message many times! For example, suppose that we have a linear array of p nodes, and we want to send a message to all nodes. In the broadcast mode, the message passes through the nodes like a wave front. The total number of messages involved is the number of the nodes, p. If broadcast is not used, $\sum_{1}^{p} i$ messages are required to achieve the same goal.

Another approach is using data link communication protocol instead of using network communication protocol to reduce the communication overhead. In the network

layer, there are routing and data type conversion functions. Routing function is called very often. If the network has many connections, the route table will be big. Looking up a path in a large table needs time. Using data link communication reduces the overhead of communication.

5.4 Recursion on a Multiprocessor

Recursion is a widely used technique in computer science. How to write a recursive function on a multiprocessor system is a question we meet when writing the sequence alignment program.

Generally speaking, we can use two kinds of recursion when solving a problem using divide-and-conquer approach, the space recursion and the time recursion. On a uni-processor computer, only time recursion is possible. In other words, the processor solves each smallest sub-problem in one time slice. On a multiprocessor computer, the recursion can be the time or the space recursion. The space recursion means allocating each smallest sub-problem to a processor to achieve parallelism. This is feasible only when the system has a large number of processors, which is hard to achieve in practice. Maybe we can do it on the next generation of Connection Machine. The time recursion on a multi-processor system is similar to that on a single processor computer. However, it lacks parallelism. Thus, the best strategy is to combine the two kinds of recursion in to the so-called space-time recursion.

How does the space-time recursion work? The key point here is to allocate a part of the problem to a processor and then let each processor solve its sub-problem by the

time recursion. So, there are two phases in the space-time recursion, the recursion on processors phase and the recursion within a processor phase.

In our sequence alignment program we use the space-time recursion to implement Myers-Miller's algorithm for finding an optimal alignment in two phases. The first phase begins with 16 nodes working in one group. The 16 nodes divide the problem into two subproblems and reorganize themselves into two 8-node groups, with each group holding one subproblem. Then each 8-node group divide its subproblem into two sub-subproblems and split themselves into two 4-node groups, with each group holding one sub-subproblem. This process keep going, until each group contains one node. In the second phase, each node does the time recursion till the solution of the its assigned problem is found.

5.5 Deadlock

Deadlock is not a new issue in the computer operating system design. We have met this problem when we worked with GENESYS, the transputer system control kernel. The phenomena are that the nodes wait for messages from each other and thus block each other, or the node GENESYS process blocks itself. It seems that GENESYS has bugs.

5.6.1 Deadlock Caused by Communication

Most deadlocks on GENESYS are caused by communication. Our experience tells us that the problem is in the communication protocol implementation or the buffer management procedure. In GENESYS, the buffer management procedure has several workers (processes) that handle messages. Once a node receives a message which is

stored in the buffer, the buffer management procedure calls a worker to receive it. The worker then tries to pass the message to the application program. If the program does not need this message at this moment, the worker is occupied by the message. The occupied worker enters the wait state and remains in the wait state until the program wants to receive this message. The trouble occurs when all workers are in the wait state and a new message is coming and this message is expected by the program. In this situation, although the message can enter the buffer, it can not reach the program. The result is that many messages are blocked in buffer and that node enters a deadlock.

After many tries and errors we know that deadlock in our initial program was also caused by using different level communication protocols alternatively. For example, we used a lot of data link communications and some network communications. The network communication is slower than data link communication because network communication calls the routing function. An explicit example is the conflict between the I/O statements and data link communications. I/O statements are implemented by sending the data to the GENESYS on the host using network layer communication. Our program uses only data link layer communication to do computation. Since the I/O statements are inserted in the program to do debugging, the deadlock occurs and the nodes are blocked by the messages which are sent right after I/O statements. The explanation is that a network layer operations may delay the data link layer operation right after it. So, if a node send three messages m1, m2 and m3 to the same destination, and m1 is sent via network layer and m2 and m3 are sent via data link layer. m2 may arrive after m3 because of m1, although m2 is sent before m3.

In our final program, we scheduled sending and receiving of messages carefully so deadlocks would not happen.

5.6.2 Virtual Circuit

Two processes can establish a virtual circuit that will remain in effect for the transmission of several messages. The processes can be on the same node or on different nodes. Virtual circuit sounds good because when it is established it transmits the message with specific event I.D.. However, it may cause deadlocks when used for network message passing if you do not implement the program properly.

When the virtual circuit is established it blocks all message except those with specific event I.D.. Even the system messages cannot pass through. Our experience is that do not use virtual circuit in a long period of time. Usually, set virtual circuit and send several messages then clean virtual circuit. If virtual circuit lasts too long it may conflict with GENESYS system control messages.

5.6 Debug Tools on GENESYS

The debugging tools on GENESYS is the symbolic debugger **tdb**. The **tdb** is not a good debugger compared with **dbx** on Unix.

One main problem with **tdb** is that it cannot debug parallel programs. You can use it to debug a program running on one node. If the program involves 8 nodes, it can do nothing. Although, it provides ability to run several debugger simultaneously on several terminals, the cooperation among these debuggers does not work properly. A debugger may become dead somewhere in the program without any error report.

Because of the problem with debugger, we had many troubles in debugging the program. Every time, before running the program, we had to make sure it works well, otherwise the transputer system may die. So often we had to do what people did many years ago when there is no debugger: insert many printing statements in the program and check the printed message. Then we had the problem of deadlock, since printing message requires communication. This is why we spent several months to complete our transputer sequence alignment program.

CHAPTER 6

EMPIRICAL RESULTS AND DISCUSSIONS

In this chapter, we present the results of our experiment on the transputer system. We have implemented two programs for finding k best non-intersecting local sequence alignments. One is the sequential program using single node and the other is the parallel program using either 8-node or 16-node on the DCSS transputer system. The single node program is based on Huang's program, SIM, for finding k best non-intersecting local sequence alignments. We have run these two programs on several sets of real DNA sequences and artificial sequences, and compared their performance and time efficiency.

6.1 Some Considerations in the Design of the Tests

The tests are designed to achieve the goals. Our first goal is to verify the correctness of the parallel program. The second goal is to determine the real efficiency of the parallel program. As we have the sequential program, SIM, from Huang, we can compare the outputs of SIM with the outputs of our parallel program to show the correctness. To show the efficiency of our parallel program, we try to determine the speed-up of the parallel program over the best sequential program. This is done by comparing the single transputer node program run time with the parallel program run time.

GENESYS provides a function for reading the current value of the system timer in a portable fashion. This function can be called from the OTB and ITB programs when GENESYS is running. To find the run time, we record the program's starting time and ending time. Note that the transputer system is a multi-user system. Recall that our transputer system has four NASs, each has 4 nodes. Thus GENESYS allow four users to use the computing nodes simultaneously. However, users occupy an NAS exclusively, i.e., if we occupy an NAS, say NAS0, the other users can not use NAS0 until we release it. So, we can find the actual run time of our program by simply subtracting the starting time from the ending time. Recall that OTB is the user interface and it starts ITB by sending a message to ITB. We insert the time reading functions in the OTB right after the input is read and before the end of the program. In this way, we can measure the run time of our two programs.

6.2 Testing the k Non-intersecting Local Sequence Alignments Program

To test our parallel transputer program, we selected 3 pairs of real DNA sequences and 4 pairs of pseudo-DNA sequences. Following Huang's papers, we obtained the real DNA sequences from Genbank. The lengths of the real sequences range from 10k to 50k.

The first pair is the beta-like globin cluster of Human and rabbit. (For simplicity we call them the beta-like pair.) This pair of sequences are reported of having high degree of similarity and Huang already has done some test using these sequences. The lengths of the human and rabbit sequences are 73,360 and 44,594, respectively. Since in the DCSS transputer system, nodes do not have sufficiently large memory, we only used a

segment from each beta-like globin cluster. The lengths of the two segments are 21,000 and 42,000, respectively.

The second pair is the alpha globin cluster of Human and rabbit. (Again, we simply call them the alpha-like pair.) This pair are also reported of having high degree of similarity. The lengths of human and rabbit sequences are 19,862 and 10,621 respectively.

Table 6.1 Test results for the 8-node configuration

unit: second

pair name	k	og	eg	m x n	sequencial	parallel	speed-up
tob-liv	5	50	60	38,400x54,000	81270	11463	7.1
tob-liv	1	30	5	24,000x45,540	96684	14370	6.7
beta-like	5	50	60	21,000x42,000	42596	5613	7.6
beta-like	5	30	5	21,000x42,000	77733	12270	6.3
alpha-like	5	50	60	10,621x19,862	10133	1376	7.3
alpha-like	5	30	5	10,621x19,862	15455	2653	5.9
						average	6.8

The third pair is the chloroplast genome of tobacco and liverwort (called the tob-liv pair). Tobacco and liverwort are two plants which shared a common ancestor 500 million years ago. The whole sequences of tobacco and liverwort are 155,844 and 121,024 long respectively. As mentioned above, due to the memory limitation, we only extracted

a pair of segments, one from each sequence, as the test sequences. The lengths of these two segments are 54,000 and 38,400 respectively. During the test, sometimes the single node program run out of memory. Thus, we had to reduce the size of the input sequences to 45,540 and 24,000 respectively, in this case.

We let the score function have 4 values. They are the match score, the mismatch score, the open-gap-penalty, and the extend-gap-penalty. Table 6.1 shows the comparison of the run time of parallel program using 8 nodes and the run time of the single node program. Table 6.2 shows the comparison of the run time of parallel program using 16 nodes with the run time of the single node program. In the Table 6.1, 6.2 and 6.3, "og" stands for the open-gap-penalty and "eg" stands for the extend-gap-penalty. The match score is 10 and mismatch score is -10.

Furthermore, to test the program performance on all kinds of sequences, we generated 4 pairs of sequences. Their similarity varies from the high level to the middle level. Here, the high level means that the length of the optimal alignment of the pair is longer than 10% of the average length of the sequences in the pair. If the length of the alignment is from 5% to 10% of the average length of the sequences, then the pair has a middle level similarity. To make these sequences have the features of real sequences, we simulated the genetic evolution process to produce these sequences. There are three generalized operations in genetic evolution process:

Crossing-over: To reproduce the next generation, parent's genes are crossed over to generate new genes.

Inversion: The order of the nucleotides is changed by inverting a segment of the sequence.

Mutation: Several nucleotides in the sequence are changed to the other nucleotides.

The implementation of the crossing-over is to cut a sequence into two segments and pick one position for each segment randomly. Then exchange the two subsegments that are between the chosen positions and the end of the segments. The implementation of the inversion is to randomly select two positions and reverse the segment between the two positions. The implementation of the mutation is to find several positions randomly and change the symbol to another symbol randomly too.

Table 6.2 Test results for the 16-node configuration

unit: second

pair name	k	og	eg	m x n	sequencial	parallel	speed-up
tob-liv	5	50	60	38,400x54,000	81270	7796	10.4
tob-liv	1	30	5	24,000x45,540	89686	6059	14.8
beta-like	5	50	60	21,000x42,000	42596	2897	14.8
beta-like	5	30	5	21,000x42,000	77733	6421	12.1
alpha-like	5	50	60	10,621x19,862	10133	753	13.4
alpha-like	5	30	5	10,621x19,862	15455	1698	9.1
						average	12.4

To generate a pair of sequences with high degree similarity, we use a sequence, generated by the random number generator, to form two sequences. One is the original sequence and the other is obtained by evolving the original sequence for several generations. Each generation is produced in three steps: crossing-over, inversion and mutation. The more generations the sequence evolves the less similarity the pair has. In our program, each generation consists of one crossing-over, one inversion, and 5% mutations. The symbols in the sequences are A, C, G and T. Table 6.3 shows the test results of finding 5 best non-intersecting local alignments on these pseudo sequences pairs. We selected 4 sets of the score functions as shown in the table.

6.3 Discussions of the Results

First of all, the alignments found by the parallel program are always the same as the alignments found by SIM. From the Table 6.1 and 6.2, we can see that the speed-up of the parallel program is from 5.9 to 7.6 for the 8-node configuration and 9.1 to 14.8 for the 16-node configuration. Table 6.3 shows the general performance of our program on pseudo-DNA sequences. The average speed-up is 6.5 for 8-node configuration and is 11 for 16-node configuration.

The results show that the size of the inputs and the score function affect the parallel program's speed-up. The larger the input sizes, the more speed up we get. The reason is that the dynamic programming section of our program takes a large portion of the total run time and thus, longer sequences lead to less proportion of setup and cleanup time. The score function can affect the lengths of the local alignments found. When the

local alignments are longer, the speed-up decreases due to the communication overhead and the unbalanced computation in some sections of the parallel program.

Table 6.3 Test results of pseudo sequences on both configurations

Unit: second

sizes	og	eg	sequential	8-nodes	16-nodes	speed-up 8	speed-up 16
10,457x12,031	30	5	7083.7	2000	1278.4	3.5	5.5
	50	60	6175.4	889	484.6	6.9	12.8
	30	10	6398	985.2	561.7	6.4	11.4
	50	5	6385.3	1017.2	579.5	6.3	11.0
9,991x9,991	30	5	5317.8	868.7	507.9	6.1	10.5
	50	60	4851.1	707.2	395.4	6.9	12.3
	30	10	5001.1	739.2	422.4	6.8	11.8
	50	5	4970.6	752.2	437.7	6.6	11.4
8,453x9,200	30	5	3909	613.0	350.6	6.4	11.1
	50	60	3728	536.2	310.3	7.0	12.0
	30	10	3825.2	567.5	358.3	6.7	10.7
	50	5	3801.4	566.1	336.0	6.7	11.3
6,895x7,431	30	5	2662.4	448.2	253.8	5.9	10.5
	50	60	2468.9	418.7	268.8	5.8	9.3
	30	10	2510	447.4	305	5.6	8.4
	50	5	2557	400.6	237.4	6.4	10.8
					average speed-up	6.3	10.7

Comparing the run times of the parallel program on the 8-node and 16-node configurations, we can see that the speed-up does not increase as fast as the number of nodes. It shows that the communication overhead still affects the parallel program. Certainly, this is unavoidable and our interest is in to reduce this overhead to its minimum.

The unbalanced computation sections can be found by analyzing Table 6.1 and 6.2. Recall that the large open-gap-penalty and large expend-gap-penalty lead to short

alignments. When open-gap-penalty is 50 and the extend-gap-penalty is 60, the speed-up are larger than the other set of score functions. So, the shorter the alignment, the higher the speed-up is. The unbalanced section must be in the section for displaying the alignments and the section for finding the masked region. To further see this, we can look at the case when k=1 in Table 6.1and 6.2. When k=1, our program does not go through sections for finding the masked region and recomputing the masked region. The speed-ups are the highest on both configurations in this case.

The pseudo sequences are used to evaluate the average performance of our parallel program. There are totally 32 tests in this group. The average speed-up is in our expectation. Several exceptions exist since our program has some drawbacks and the exception cases just touch the drawbacks.

CHAPTER 7

CONCLUSION

In this thesis, the sequence alignment problem and the related algorithms are studied. In particular, the design and implementation of the parallel transputer program for finding k best non-intersecting local alignments is presented. Since the basic problem-solving technique is the dynamic programming, the parallelization of the dynamic programming is discussed thoroughly, including the selection of the network topology and the selection of partition.

Recall that the two aspects of our goal are to find a efficient implementation of the local sequence alignment problem and to show the parallel processing power of the transputer system. Based on the test results in Chapter 6, the first aspect of our goal is achieved. The parallel program's speed up is 6.8 on the average for the 8-node configuration and 11 on the average for the 16-node configuration. Our parallel sequence alignment program on transputer is a successful attempt and it shows that the transputer system can provide an acceptable speed-up in solving the dynamic programming based problems and the communication ability of the transputer system can meet the general requirement of the computation-intensive applications.

The best feature of the transputer system is its flexible connection among nodes. It can be configured as many networks with different topology. This feature provides the

freedom of selecting the best topology suitable for the specific approach for solving a problem.

Certainly, the DCSS transputer system has two drawbacks. They are the small memory space and the lack of customer supporting service. The main problem is the memory size which limits our study. We had to reduce the input size to meet the limitation. The maximum memory size of the MC1000 transputer, TRANSTECH product, is 8 MByte/node at maximum. But the current DCSS system has two kinds of memory configurations: 4 MByte/node or 2 MByte/node. Problems occurred at the 2 MByte memory nodes. The service provided by TRANSTECH is not adequate. We had to solve the problems with the system on our own.

Although the existence of these drawbacks, the transputer system is still a good candidate for the sequence alignment algorithm and the other dynamic programming type applications.

REFERENCES

1. *Academy Backs Genome Project*, Science, 239: 725-726.
2. J. W. Arendt, *Parallel Genome Sequence Comparison Using a Concurrent File System*, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-91-1674 (1991).
3. J. F. Collins and A. F. W. Coulson, *Significance of Protein Sequence Similarities*, Methods in Enzymology, Vol.183, pp.474-487 (1990).
4. E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith, *Parallel Processing of Biological Sequence Comparison Algorithms*, International Journal of Parallel Programming, Vol. 17, No. 3 (1988).
5. D. S Hischberg, *Recent Results on the Complexity of Common-subsequence Problems*, in **Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison**, D. Sankoff and J. B. Kruskal (eds), Addison-Wesley, Reading, MA (1983).
6. X. Huang, *A Space-Efficient Parallel Sequence Comparison Algorithm for a Message-Passing Multiprocessor*, International Journal of Parallel Computing, Vol.18, No.3 (1989).
7. X. Huang, R. C. Hardison and W. Miller, *A Space-Efficient Algorithm for Local Similarities*, CABIOS, Vol.6 No.4, pp. 373-381 (1990).
8. X. Huang and W. Miller, *A Time-Efficient, Linear-Space Local Similarity Algorithm*, Advances in Applied Mathematics, Vol.12, No.3 (1991).
9. X. Huang, W. Miller, and R. C. Hardison, *Parallelization of a Local Similarity Algorithm*, Private paper, 1990.
10. O. Gotoh, *An Improved Algorithm for Matching Biological Sequences*, Journal of Molecular Biology 162: 705-708(1982).

11. O. H. Ibarra, T. Jiang and H. Wang, *String Editing on a One-Way Linear Array of Finite-State Machines*, IEEE Transaction on Computer, Vol.41, No. 1 (1992).

12. O. H. Ibarra, T. Pong, and S. M. Sohn, *String Processing on Hypercube*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 38, No. 1 (1990).

13. INMOS Limited, **The Transputer Databook**, Consolidated Printers, Berkeley, CA.(1989).

14. J. B. Kruskal and D. Sankoff, *An Anthology of Algorithms and Concepts for Sequence Comparison*, in **Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison**, D. Sankoff and J. B. Kruskal(eds), Addison-Wesley, Reading, MA(1983).

15. G. M. Landau, U. Vishkin, and R. Nussinov, *Fast Alignment of DNA and Protein Sequences*, Methods in Enzymology, Vol. 183, pp 487-502 (1990).

16. E. Lander and J. P. Mesirov, *Protein Sequence Comparison on a Data Parallel Computer*, Proceedings of the 1988 ICPP, pp 3:257-263.

17. A. M. Lesk (eds), **Computational Molecular Biology -- Sources and Methods for Sequence Analysis**, Oxford University Press(1988).

18. R. Lipton and D. Lopresti, *Comparing Long Strings on a Short Systolic Array*, in **Systolic Arrays**, W. Moore, A. McCabe, and R. Urquhart(eds), pp.181-190, Adam Hilger (July 1986).

19. R. J. Lipton, T. G. Marr, and J. D. Welsh, *Computational Approaches to Discovering Semantics in Molecular Biology*, Proceedings of The IEEE, Vol.77, No.7 (1989).

20. J. V. Maizel, *Supercomputing in Molecular biology: Applications to Sequence Analysis*, IEEE Engineering in Medicine and Biology, pp. 27-30 (1988).

21. E. W. Myers and W. Miller, *Optimal Alignments in Linear Space*, CABIOS, Vol.4, No.1 pp 11-17 (1988).

22. S. B. Needleman and C. D. Wunsch, *A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins*, Journal of Molecular Biology 48:443-453 (1970).

23. S. Ranka and S. Sahni, *String Editing on an SIMD Hypercube Multicomputer*, Journal of Parallel and Distributed Computing, Vol. 9, pp 411-418 (1990).

24. T. F. Smith and M. S. Waterman, *Identification of Common Molecular Subsequences*, Journal of Molecular biology 147: 195-196 (1981).

25. R. A. Wagner and M. J. Fischer, *The String-to-String Correction Problem*, Journal of the ACM 21(1): 168-173(1974).

26. M. S. Waterman and M. Eggert, *A New Algorithm for Best Subsequence Alignments with Application to tRNA-rRNA Comparisons*, Journal of Molecular biology, 197: 723-728 (1987).

27. M. S. Waterman, **Mathematical Methods for DNA sequences**, CRC Press(1989).

28. W. J. Wilbur and D. J. Lipman, *Rapid Similarity Searches of Nucleic Acid and Protein Data Banks*, in Proceedings of the National Academy of Sciences, Vol.80, pp 726-730 (1983).

29. M. Wu and D. D. Gajski, *Computer-Aided Programming for Message-Passing Systems: Problems and a Solution*, Proceedings of the IEEE, Vol.77, No.12 (1989), pp. 1983-1991.

APPENDEX I: Headfile Listing

```
/* ———————— predefine.h file ———————— */
#include <stdio.h>
#include "net.h"
#include <genesys/events.h>
#include <genesys/malloc.h>
#include <genesys/t_types.h>
#include <genesys/castreq.h>
#define LOAD1 101
#define LOAD 10
#define LOADB 11
#define START1 200
#define START 100
#define NOTE 99
#define p_d 10001
#define c_start 10002
#define c_start1 910002
#define PICK 102
#define BC 9901
#define FLAG1 MYHOLD
#define FLAG DINTDATA
#define whisper_msg 10003
#define FIRSTNODE 0

#define flag_msg 90211
#define OPT 199
#define PASS 31415926

#define c_stt 841128

#define COLLECT 20256
#define AL 42956
#define COST 1128
#define ARRAY 2256

#define rvcmd 950000
#define rvdt 1000000
#define chat_msg 310003

**********************************************************

/* ———————— macro.h file ———————— */

char *AA, *BB;
int nodeid, K, P, desti, SI;
static int (*v)[128];                    /* substitution scores */
static int q, r;                         /* gap penalties */
static int qr;                           /* qr = q + r */
int  *S;
static int CV[128][128];

typedef struct ONE {
```

```
        int ROW ;
        struct ONE  *NEXT ;} pair, *pairptr;
pairptr *col, z;                    /* for saving used aligned pairs */
static short tt;

typedef struct NODE
        { int  SCORE;
          int  STARI;
          int  STARJ;
          int  ENDI;
          int  ENDJ;
/*        int  TOP;
          int  LEFT;    */
          int  BOT;
          int  RIGHT; }  vertex, *vertexptr;

vertexptr  *LIST;                   /* an array for saving k best scores */
vertexptr  low = 0;                 /* lowest score node in LIST */
vertexptr  most = 0;                /* latestly accessed node in LIST */
static int numnode;                 /* the number of nodes in LIST */

static int *CC, *DD;                /* saving matrix scores */
static int *RR, *SS, *EE, *FF;      /* saving start-points */
static int *HH, *WW;                /* saving matrix scores */
static int *UU, *VV;
static int *II, *JJ, *XX, *YY;      /* saving start-points */
static int  m1, mm, n1, nn;         /* boundaries of recomputed area */
static int  lb, tb;                         /* left and top boundaries */
static int  min;                    /* minimum score in LIST */
static short flag;                          /* indicate if recomputation necessary*/

/* DIAG() assigns value to x if (ii,jj) is never used before */
#define DIAG(ii, jj, x, value)                                  \
{ for ( tt = 1, z = col[(jj)]; z != 0; z = z->NEXT )            \
    if ( z->ROW == (ii) )                                       \
      { tt = 0; break; }                                        \
  if ( tt )                                                     \
    x = ( value );                                                      \
}

/* replace (ss1, xx1, yy1) by (ss2, xx2, yy2) if the latter is large */
#define ORDER(ss1, xx1, yy1, ss2, xx2, yy2)                     \
{ if ( ss1 < ss2 )                                              \
    { ss1 = ss2; xx1 = xx2; yy1 = yy2; }                        \
  else                                                          \
    if ( ss1 == ss2 )                                           \
      { if ( xx1 < xx2 )                                        \
            { xx1 = xx2; yy1 = yy2; }                           \
          else                                                  \
            if ( xx1 == xx2 && yy1 < yy2 )              \
              yy1 = yy2;                                        \
```

```
    }                                                    \
}

/* The following definitions are for function diff() */

static int Ii, Jj;                          /* start points of I and J */
int  diff(), display();
static int  zero = 0;                       /* int type zero       */

#define gap(k)  ((k) <= 0 ? 0 : q+r*(k))    /* k-symbol indel score */

static int *sapp;                                /* Current script append ptr */
static int  last;                           /* Last script op appended */

static int I, J;                            /* current positions of A ,B */
static int no_mat;                          /* number of matches */
static int no_mis;                          /* number of mismatches */
static int al_len;                          /* length of alignment */


                                            /* Append "Delete k" op */
#define DEL(k)                                       \
{ I += (k);                                          \
  al_len += (k);                                     \
  if (last < 0)                                      \
    last = sapp[-1] -= (k);                                  \
  else   {                                           \
    last = *sapp++ = -(k);                                   \
    SI++;  }                                         \
}
                                                     /* Append "Insert k" op */
#define INS(k)                                       \
{ J += k;                                            \
  al_len += (k);                                     \
  SI++;                                                      \
  if (last < 0)     {                                \
     sapp[-1] = (k); *sapp++=last;                           \
          }                                          \
  else                                               \
     last = *sapp++ = (k);                                   \
}


                                                     /* Append "Replace" op */
#define REP                                          \
{ last = *sapp++ = 0;                                \
  SI++;                                                      \
  al_len += 1;                                       \
}
```

APPENDIX II: ITB Program Listing

```
/* ———————— ITB program ———————— */
#include "predefine.h"
#define SORT 555;
#define SORTED 5555;
#define S_type 777;
#include "macroh"

double tm1, tm2, tm3;
int mark=0, *pointer, *buffer, *indicator, mash=0x0000ffff;
int low_pos, most_pos, last_rt, found, wi, wcell, nn1, mm1;
struct nmsg mesg, cmdm;
vertex MAX;
/*int low2,low2_pos;*/

/* Add a new node into list. */

#define addnode(cc, cci, ccj, i1, j1, K1, cost, bot, rig, mod)          \
{                                                                       \
  register int dd;                                                      \
  found = 0;                                                            \
  if ( most != 0 && most->STARI == cci && most->STARJ == ccj )          \
    found = 1;                                                          \
  else                                                                  \
    for ( dd= 0; dd< numnode ; dd++ )                                   \
        { most = LIST[dd];                                              \
          if ( most->STARI == cci && most->STARJ ==ccj )                \
            { found = 1;                                                \
              most_pos = dd;                                            \
              break;                                                    \
            }                                                           \
        }                                                               \
  if ( found )                                                          \
    { if ( most->SCORE < cc)                                            \
        {                                                               \
           indicator[most_pos] = 1;                                     \
           most->SCORE =cc;                                             \
         most->ENDI = (i1);                                             \
         most->ENDJ = (j1);                                             \
        }                                                               \
     if (mod==0) {                                                      \
         if ( most->BOT < (i1)) most->BOT = (i1);                       \
         if ( most->RIGHT < (j1)) most->RIGHT = (j1);                   \
         }                                                              \
     else {                                                             \
         if ( most->BOT < bot ) most->BOT = (bot);                      \
         if ( most->RIGHT < rig ) most->RIGHT = (rig);                  \
         }                                                              \
  }                                                                     \
  else                                                                  \
    { if ( numnode < K1 )          /* list is not full */               \
        {                                                               \
          most_pos = numnode;                                           \
```

```
        most = LIST[numnode++];                                    \
        indicator[most_pos] = 1;                                   \
      }                                                            \
    else                                                           \
      {                                                            \
        most = low;             /* list is full        */        \
        indicator[low_pos] = 1;                                    \
        most_pos = low_pos;                                        \
      }                                                            \
    most->SCORE =cc;                                               \
    most->STARI =cci;                                              \
    most->STARJ =ccj;                                              \
    most->ENDI = (i1);                                             \
    most->ENDJ = (j1);                                             \
    if (mod==0) {                                                  \
      most->BOT = (i1);                                            \
      most->RIGHT = (j1);                                          \
      }                                                            \
    else {                                                         \
      most->BOT = bot;                                             \
      most->RIGHT = rig;                                           \
      }                                                            \
  }                                                                \
 if ( numnode == K1 )                                              \
  { if ( low == most || ! low )                                    \
    { for ( low=LIST[0],low_pos=0,dd=1; dd<numnode ;dd++ )         \
      if ( LIST[dd]->SCORE < low->SCORE )                          \
      {  low = LIST[dd];                                           \
         low_pos = dd;}                                            \
      }                                                            \
      cost = low->SCORE;                                           \
  }                                                                \
}

main()
{
    int i, j, nseq, M_I, N_I, si, sj, k, list_length;
    int *fst(), *snd(), findflag();
    vertexptr MAX1, sort();
    pairptr zptr;
    double sttm, commtm, runtm, ttime(), ptime;

    nodeid = getnodeid();
    if (nodeid==3 || nodeid==7 || nodeid==11 || nodeid==15) desti = 3;
      else if (nodeid<3 ||(nodeid>7 && nodeid<11)) desti = 2;
            else desti = 1;

 /*  receive command from HOST   */
    mesg.nh_event = NOTE;
    mesg.nh_type = LOAD;
    mesg.nh_flags = FLAG;
    nrecv(&mesg);
```

```
    M_l = mesg.nh_data[0];
    N_l = mesg.nh_data[1];
    K = mesg.nh_data[2];
    q = mesg.nh_data[4];
    r = mesg.nh_data[5];
    P = mesg.nh_data[6];
    numnode = min = 0;
    qr = q + r;

/*  tprintf("M %d N %d q %d r %d in node %d\n", M_l, N_l, q,r, nodeid);
  create convert table  */
    for (i=0; i<128; i++)
        for (j=0; j<128; j++)
            if (i==j) CV[i][j] = 10;
            else CV[i][j] = mesg.nh_data[3];
    AA = (char *)malloc(M_l+1);
    BB = (char *)malloc(N_l+1);

/*  receive sequence A  */
    mesg.nh_event = START;
    mesg.nh_type = LOAD;
    mesg.nh_length = M_l;
    mesg.nh_flags = 0;
    mesg.nh_msg = &AA[1];
    nrecv(&mesg);

/*  receive sequence B  */
    mesg.nh_event = NOTE;
    mesg.nh_type = LOADB;
    mesg.nh_length = N_l;
    mesg.nh_flags = 0;
    mesg.nh_msg = &BB[1];
    nrecv(&mesg);

/*  allocate memory for working array */
        col = (pairptr *) malloc((N_l+1)*sizeof(pair));
    for (i=1; i<=N_l; i++) col[i] = 0;

/*  create LIST for K best alignments  */

    indicator = (int *)malloc(K * 4);
    LIST = (vertexptr *) malloc(K * sizeof(vertexptr));
    for (i=0; i<K; i++)
        LIST[i] = (vertexptr) malloc(sizeof(vertex));
    sttm = ttime();

    commtm = ttime();
    if(nodeid == 0) {
          cmdm.nh_event = START1;
        cmdm.nh_type = LOAD1;
        cmdm.nh_length = 0;
        cmdm.nh_flags = FLAG;
```

```
            nrecv(&cmdm);
          M_l = cmdm.nh_data[1];   /* length of the array A  */
          N_l = cmdm.nh_data[3];   /* length of the local array B  */
          si = cmdm.nh_data[0];    /* starting position on A */
            sj = cmdm.nh_data[2];    /* startomg position on B */
          }
      ptime = ttime();
      big_pass(AA, BB, M_l, N_l, si, sj, K, nseq);
      printf("run-time of BIG_PASS is %f in node %d\n",ttime()-ptime,nodeid);
      for (k=K-1; k>=0; k--) {
          list_length = k>numnode? k+1 : numnode;
          MAX1 = sort(list_length);
          MAX.SCORE = MAX1->SCORE;
          MAX.STARI = MAX1->STARI;
          MAX.STARJ = MAX1->STARJ;
          MAX.BOT = MAX1->BOT;
          MAX.RIGHT = MAX1->RIGHT;
          MAX.ENDI = MAX1->ENDI;
          MAX.ENDJ = MAX1->ENDJ;

          alignment(MAX.STARI+1,MAX.STARJ+1,MAX.ENDI,MAX.ENDJ,0,P-1,q,q);
          gather();
          update(M_l, N_l);
          if (k == 0) break;
          mark++;
          flag = 0;
          mm = MAX.BOT; nn = MAX.RIGHT;
        ptime = ttime();
          reverse(AA, BB, MAX.BOT, MAX.RIGHT, MAX.STARI, MAX.STARJ);
      printf("run-time of REVS is %f in node %d\n",ttime()-ptime,nodeid);
          if (findflag() > 0)
          {
        ptime = ttime();
              big_pass(AA, BB, MAX.BOT-m1+1, MAX.RIGHT-n1+1, m1, n1, list_length, nseq);
/*      printf("run-time of SMALL_PASS is %f in node %d\n",ttime()-ptime,nodeid); */
          }
       }
       runtm = ttime();
       if (nodeid == P-1) {
          printf("runtm %f \n", runtm - sttm);
          printf("commtm %f \n", commtm - sttm);
       }
      free(AA);
      free(BB);
      kexit();
}

/*  pack two integers into one integer number  */
#define pack(num1, num2, num4)                          \
{   num4 = (num1 <<16) & (~mash)|num2 & mash;       }
```

```
/*  unpack one integer number to two integers  */
#define unpack(num3, derive)                                    \
{       if ((num3&mash)>>15 == 1) derive[1] = num3|(~mash);     \
          else derive[1] = num3 & mash;                         \
          derive[0] = num3>>16&mash;                            \
}


/* function big_pass(A, B, M_l, N_l, si, sj, K, nseq) computes the first
round of dynamic programming task. A and B are two sequences, M_l, N_l are
length of A and B, si and sj are EXACTLY starting points of computing. */


big_pass(A, B, M_l, N_l, si, sj, k, nseq)
char A[], B[];
int k, nseq, M_l, N_l, si, sj;
{
  register int   j;                     /* row and column indices */
  register int   c;                     /* best score at current point */
  register int   f;                     /* best score ending with insertion */
  register int   d;                     /* best score ending with deletion */
  register int   p;                     /* best score at (i-1, j-1) */
  register int   ci, cj;            /* end-point associated with c */
  register int   di, dj;            /* end-point associated with d */
  register int   fi, fj;            /* end-point associated with f */
  register int   pi, pj;            /* end-point associated with p */
  register int     *va;                 /* pointer to v(A[i], B[j]) */
  register int content1[2];
  struct nmsg whisper;
  register int sti, stj, lenj;
  int temp, leni, B_len, infor[8];
  register int i;
  double mtime, ttime();
        /* Compute the matrix and save the top K best scores in LIST
           CC : the scores of the current row
           RR and EE : the starting point that leads to score CC
           DD : the scores of the current row, ending with deletion
           SS and FF : the starting point that leads to score DD       */

#define PAss(package_i,package_o,i0,basej,sizej,kk)     \
{                                                       \
        va = CV[AA[i0]];                                \
        unpack(package_i[6], content1)                  \
        c = content1[0];                                \
        f = content1[1];                                \
        p = package_i[0];                               \
        pi= package_i[1];                               \
        pj= package_i[2];                               \
        ci= package_i[7];                               \
        cj= package_i[3];                               \
        fi= package_i[4];                               \
        fj= package_i[5];                               \
```

```
        for (j = 0; j < sizej; j++)                          \
        { c = c - qr;                                        \
          f = f - r;                                         \
          ORDER(f, fi, fj, c, ci, cj)                        \
          c = CC[j] - qr;                                    \
          ci = RR[j];                                        \
          cj = EE[j];                                        \
          d = DD[j] - r;                                     \
          di = SS[j];                                        \
          dj = FF[j];                                        \
          ORDER(d, di, dj, c, ci, cj)                        \
          c = 0;                                             \
          DIAG(i0, j+basej, c, p + va[BB[j+basej]])   /* diagonal */ \
          if ( c <= 0 )                                      \
            { c = 0; ci = i; cj = j+basej;                   \
                }                                            \
          else                                               \
            { ci = pi; cj = pj;                              \
                }                                            \
          ORDER(c, ci, cj, d, di, dj)                        \
          ORDER(c, ci, cj, f, fi, fj)                        \
          p = CC[j];                                         \
          CC[j] = c;                                         \
          pi = RR[j];                                        \
          pj = EE[j];                                        \
          RR[j] = ci;                                        \
          EE[j] = cj;                                        \
          DD[j] = d;                                         \
          SS[j] = di;                                        \
          FF[j] = dj;                                        \
          if ( c > min )    /* add the score into list */      \
                addnode(c, ci, cj, i, j+basej, kk, min,0,0,0)          \
        }                                                    \
        package_o[0] = p;                                    \
        package_o[1] = pi;                                   \
        package_o[2] = pj;                                   \
        pack(c, f, package_o[6])                             \
        package_o[7] = ci;                                   \
        package_o[3] = cj;                                   \
        package_o[4] = fi;                                   \
        package_o[5] = fj;          }



        /* Initialize the 0 th row */
/*      mark++;  */
    temp = (N_I/P + 17);
    CC = (int *)malloc(temp*24);
    RR = CC + temp;
    SS = RR + temp;
    EE = SS + temp;
    DD = EE + temp;
```

```
FF = DD + temp;
if (nodeid == 0) {
    whisper.nh_dl_event = dl_event(desti);
    whisper.nh_event = p_d;
    whisper.nh_type = c_start;  /* whisper_msg;*/
    whisper.nh_data[1] = N_l;
    whisper.nh_data[2] = sti = si;              /* start point of A */
    whisper.nh_data[3] = M_l;                   /*  length of A   */
    whisper.nh_data[5] = N_l/P;                 /*  length  of B */
    lenj = N_l – (P–1)*whisper.nh_data[5];
  whisper.nh_data[4] = sj+lenj;                 /* start point of B */
    whisper.nh_length = 0;
    whisper.nh_flags = FLAG1;
    whisper.nh_node = nodeid + 1;
  if (P!=1)  dsend(&whisper);
    for (j=0; j<lenj; j++) {
            CC[j] = 0;
            RR[j] = si–1;
            EE[j] = sj+j;
            DD[j] = –q;
            SS[j] = si–1;
            FF[j] = sj+j;
            }
  infor[0] = 0;                             /* p[i–1][j–1] */
  infor[2] = sj–1;              /* pj  */
  infor[3] = sj–1;              /* cj  */
  infor[5] = sj–1;              /* fj  */
  pack(0, –q, infor[6])
                                            /* c[i][j–1] and f[i][j–1] */
    for (i=si; i<si + M_l;i++ )
    {
            infor[1] = i–1;                 /* pi */
            infor[4] = i;                   /* fi  */
            infor[7] = i;                   /* ci  */
            whisper.nh_node = nodeid + 1;
        PAss(infor, whisper.nh_data, i, sj, lenj, k)
            dsend(&whisper);
            }
    }
else {
    whisper.nh_event = p_d;
    whisper.nh_type = c_start; /*whisper_msg;*/
    whisper.nh_length = 0;
    whisper.nh_flags = 0;

      drecv(&whisper);
    B_len = whisper.nh_data[1];
    sti = whisper.nh_data[2];
      stj = whisper.nh_data[4];
      leni = whisper.nh_data[3];
      lenj = whisper.nh_data[5];
```

```
        if (nodeid != P -1 ) {
                whisper.nh_flags = FLAG1;
                whisper.nh_data[4] = stj+lenj;              /* start point */
                whisper.nh_dl_event = dl_event(desti);
                whisper.nh_length = 0;
                whisper.nh_node = nodeid + 1;
                dsend(&whisper);
                }
          for (j=0; j<lenj; j++) {
                CC[j] = 0;
                RR[j] = SS[j] = sti-1;
                EE[j] = FF[j] = stj+j;
                DD[j] = -q;
                }
        for (i=sti; i< sti + leni; i++)
        {
             drecv(&whisper);

                 PAss(whisper.nh_data,whisper.nh_data, i, stj, lenj, k)
             whisper.nh_type = c_start;
             whisper.nh_node = nodeid + 1;
                 if (nodeid != P-1)
                   dsend(&whisper);
         }
    }
    free(CC);
    return(0);
}

/* copy_list() copys all new elements in the LIST to the buffer     */
#define copy_list(l_size, locj)                                      \
  {       for (wi = 0, locj=0; wi<l_size; wi++)                      \
             if (indicator[wi]==1) {                                 \
                  pointer[locj*7] = LIST[wi]->SCORE;                 \
                  pointer[locj*7+1] = LIST[wi]->STARI;               \
                  pointer[locj*7+2] = LIST[wi]->STARJ;               \
                  pointer[locj*7+3] = LIST[wi]->ENDI;                \
                  pointer[locj*7+4] = LIST[wi]->ENDJ;                \
                  pointer[locj*7+5] = LIST[wi]->BOT;                 \
                  pointer[locj*7+6] = LIST[wi]->RIGHT;               \
                  locj++;                                            \
                indicator[wi] = 0;                                   \
             }                                                       \
  }


/* sort() collects all elements in all nodes and sorts lists and send K
larger elements to the node 0   */


vertexptr sort(num)
int num;
```

```
{
        int ID, j, i;
        register int d;
        struct nmsg sort1;
        vertex maxl;
        vertexptr max, findmax();
        if (nodeid==0||nodeid==7||nodeid==8||nodeid==15)
              ID = 0;
        else  if (nodeid==3||nodeid==4||nodeid==11||nodeid==12) ID = 3;
                 else ID = 2;
        pointer = (int *)malloc(K*28);
        buffer = (int *)malloc(K*28);
        if (ID == 0 || ID == 3) {   /* nodes 0,4,8,12,3,7,11 & 15      */

          sort1.nh_dl_event = dl_event(ID? 1 : 2);
          sort1.nh_event = SORT;
          sort1.nh_type = S_type;
          sort1.nh_node = nodeid%4 ? nodeid-1 : nodeid+1;
          sort1.nh_flags = DINTDATA | DINTMSG;
          copy_list(numnode, sort1.nh_data[0])
          sort1.nh_length = 28*sort1.nh_data[0];
          sort1.nh_msg = pointer;
          dsend(&sort1);
        }
        else {
          sort1.nh_event = SORT;
          sort1.nh_type = S_type;
          sort1.nh_flags = 0;
          sort1.nh_msg = buffer;
          sort1.nh_length = K*28;
          drecv(&sort1);
          for (wi=0; wi<sort1.nh_data[0]; wi++)
            if (buffer[wcell=wi*7]>min)
               addnode(buffer[wcell], buffer[wcell+1], buffer[wcell+2], buffer[wcell+3],
                        buffer[wcell+4], num,min,buffer[wcell+5],buffer[wcell+6],1);
          if (nodeid == 2||nodeid==10||nodeid==5||nodeid==13) {
            sort1.nh_dl_event = dl_event(1);
            sort1.nh_node = nodeid%2 ? nodeid+1 : nodeid-1;
            sort1.nh_flags = DINTDATA | DINTMSG;
            copy_list(numnode, sort1.nh_data[0])
            sort1.nh_length = 28*sort1.nh_data[0];
            sort1.nh_msg = pointer;
            dsend(&sort1);
          }
          else  {
            sort1.nh_length = K*28;
            sort1.nh_msg = buffer;
            drecv(&sort1);
            for (wi=0; wi<sort1.nh_data[0]; wi++)
            if (buffer[wcell=wi*7]>min)
               addnode(buffer[wcell], buffer[wcell+1], buffer[wcell+2], buffer[wcell+3],
                        buffer[wcell+4], num, min, buffer[wcell+5], buffer[wcell+6], 1);
```

```
    if (nodeid==6||nodeid==9)  {
          sort1.nh_dl_event = dl_event(nodeid-6);
          sort1.nh_node = nodeid%2 ? nodeid+5 : nodeid-5;
          sort1.nh_flags = DINTDATA | DINTMSG;
          copy_list(numnode, sort1.nh_data[0])
          sort1.nh_length = 28*sort1.nh_data[0];
          sort1.nh_msg = pointer;
          dsend(&sort1);
        }
        else    {
          sort1.nh_length = K*28;
         sort1.nh_msg = buffer;
          drecv(&sort1);
          for (wi=0; wi<sort1.nh_data[0]; wi++)
            if (buffer[wcell=wi*7]>min)
                addnode(buffer[wcell], buffer[wcell+1], buffer[wcell+2], buffer[wcell+3],
                        buffer[wcell+4], num, min, buffer[wcell+5], buffer[wcell+6], 1);
          if (P==16) {
            if (nodeid==14) {
              sort1.nh_dl_event = dl_event(3);
                sort1.nh_node = 1;
              sort1.nh_flags = DINTDATA | DINTMSG;
                copy_list(numnode, sort1.nh_data[0])
                sort1.nh_length = 28*sort1.nh_data[0];
              sort1.nh_msg = pointer;
              dsend(&sort1);
            }
            else {
                sort1.nh_event = SORT;
                sort1.nh_type = S_type;
                sort1.nh_flags = DINTDATA | DINTMSG;
                sort1.nh_length = K*28;
                sort1.nh_msg = buffer;
                drecv(&sort1);
             for (wi=0; wi<sort1.nh_data[0]; wi++)
                 if (buffer[wcell=wi*7]>min)
                        addnode(buffer[wcell], buffer[wcell+1], buffer[wcell+2],
                                buffer[wcell+3], buffer[wcell+4], num, min, buffer[wcell+5],
                                buffer[wcell+6], 1);
            }
          }
  if (nodeid==1) {
        sort1.nh_dl_event = dl_event(1);
        sort1.nh_event = SORTED;
        sort1.nh_node = 0;
        sort1.nh_flags = DINTDATA | DINTMSG;
        copy_list(numnode, sort1.nh_data[0])
        sort1.nh_length = 28*sort1.nh_data[0];
        sort1.nh_msg = pointer;
        dsend(&sort1);}
  }
}
```

```
    }

      if (nodeid == 0) {
        sort1.nh_event = SORTED;
        sort1.nh_type = S_type;
        sort1.nh_length = K*28;
        sort1.nh_msg = buffer;
        sort1.nh_flags = 0;
        drecv(&sort1);
        for (wi=0; wi<sort1.nh_data[0]; wi++)
          if (buffer[wcell=wi*7]>min)
            addnode(buffer[wcell], buffer[wcell+1], buffer[wcell+2], buffer[wcell+3],
                    buffer[wcell+4], num, min, buffer[wcell+5], buffer[wcell+6], 1);
      free(pointer);
      free(buffer);
      max = findmax();
      maxl.SCORE = sort1.nh_data[0] = max->SCORE;
      maxl.STARI = sort1.nh_data[1] = max->STARI;
      maxl.STARJ = sort1.nh_data[2] = max->STARJ;
      maxl.ENDI = sort1.nh_data[3] = max->ENDI;
      maxl.ENDJ = sort1.nh_data[4] = max->ENDJ;
      maxl.BOT = sort1.nh_data[5] = max->BOT;
      maxl.RIGHT = sort1.nh_data[6] = max->RIGHT;
      sort1.nh_data[7] = min;
      sort1.nh_dl_event = dl_event(desti);
    sort1.nh_event = PICK;
    sort1.nh_type = LOAD1;
      sort1.nh_length = 0;
      sort1.nh_flags = FLAG;
      sort1.nh_node = nodeid + 1;
      dsend(&sort1);                          /*send*/
      }

/* passing globle max and update the LIST */
else {
    free(pointer);
      free(buffer);
      sort1.nh_event = PICK;
      sort1.nh_type = LOAD1;
      sort1.nh_length = 0;
      sort1.nh_flags = FLAG;
      drecv(&sort1);
      maxl.SCORE = sort1.nh_data[0];
      maxl.STARI = sort1.nh_data[1];
      maxl.STARJ = sort1.nh_data[2];
      maxl.ENDI = sort1.nh_data[3];
      maxl.ENDJ = sort1.nh_data[4];
      maxl.BOT = sort1.nh_data[5];
      maxl.RIGHT = sort1.nh_data[6];
      min = sort1.nh_data[7];

      sort1.nh_dl_event = dl_event(desti);
```

```
        sort1.nh_event = PICK;
        sort1.nh_type = LOAD1;
        sort1.nh_length = 0;
        sort1.nh_flags = FLAG;
        sort1.nh_node = nodeid + 1;
        if (nodeid != P-1) {
          dsend(&sort1);
        }

    for ( d = 0; d < numnode ; d++ )
      {
        max = LIST[d];
        if ((max1.STARI == max->STARI) && (max1.STARJ == max->STARJ))
        {
          numnode--;
          if (d != numnode) {
              LIST[d] =LIST[numnode];/* LIST[numnode-1]=LIST[numnode]; */
              LIST[numnode] = max;
              if (low_pos==numnode) low_pos=d;
              }
          most = LIST[0];
          if (low == max) { low = LIST[0]; low_pos = 0;}
          break;  }
      }
     }
    return(&max1);
}

vertexptr findmax()
{
  register int i, j;
  vertexptr cur;
  for ( j = 0, i = 1; i < numnode ; i++ )
    if ( LIST[i]->SCORE > LIST[j]->SCORE )
      j = i;
    else if (LIST[i]->SCORE==LIST[j]->SCORE && LIST[i]->STARJ<LIST[j]->STARJ)
        j = i;
  cur = LIST[j];
  if ( j != --numnode )
    { LIST[j] = LIST[numnode];
      LIST[numnode] =  cur;
    }
  most = LIST[0];
  if ( low == cur ) {
        low = LIST[0];
        low_pos =0;
    }
  return (cur);
}
```

```
alignment(stti, sttj, endi, endj, headnd, endnd, tb, te)
int stti, sttj, endi, endj, headnd, endnd, tb, te;
{
    int midnd, mid_i, mid_j,i,oldnd, temp;
    int *cont, id, *fst(), *snd(), newmem;

    mid_i = stti + (endi - stti)/2;
    temp = 2*(endj-sttj)/(endnd-headnd+1)+16;
    newmem = temp*4;
    CC = (int *)malloc(newmem*6);
    RR = CC + temp;
    SS = RR + temp;
    EE = SS + temp;
    DD = EE + temp;
    FF = DD + temp;
    if (endnd!=headnd && nodeid <= (midnd = headnd+(endnd-headnd)/2))
        {
        oldnd=endnd;
        cont = fst(stti,mid_i,sttj,endj,oldnd,headnd,endnd=midnd, tb, te);
        }
    else
        {
        oldnd=headnd;
        cont = snd(mid_i+1,endi,sttj,endj,oldnd,headnd=midnd+1,endnd,tb,te);
        }
    mid_j = cont[0];
    id = cont[1];
     free(CC);

/*      recursion on computing node    */
    if (headnd != endnd)
        if (id == 1)                /*      type 1  */
           if (nodeid > midnd) {
              alignment(mid_i+1, mid_j+1, endi, endj, headnd, endnd, q, te);}
           else
              alignment(stti, sttj, mid_i, mid_j, headnd, endnd, tb, q);
        else  {                     /*      type 2  */
           if (nodeid > midnd)
              alignment(mid_i+2, mid_j+1, endi, endj, headnd, endnd, 0, te);
           else {
              alignment(stti, sttj, mid_i-1, mid_j, headnd, endnd, tb, 0);
              if (nodeid == endnd) DEL(2);
            }
          }
/*      recursion on each computing node       */
    else
        {
        al_len = last = SI = 0;
        no_mis = no_mat = 0;
        I = J = 0;
        if (id == 1)
           if (nodeid%2 == 0) {
```

```
temp = mid_j-sttj+1;
newmem = temp *4 + 4;
S = sapp = (int *)malloc(newmem);
CC = (int *)malloc(newmem*6);
RR = CC + temp +1;
SS = RR + temp+1;
EE = SS + temp+1;
DD = EE + temp+1;
FF = DD + temp+1;
        Ii = stti -1; Jj = sttj -1;
        recu_al(&AA[Ii], &BB[Jj],mid_i-stti+1,temp, tb, q);
        }
   else {
temp = endj-mid_j;
newmem = temp *4 + 8;
S = sapp = (int *)malloc(newmem);
CC = (int *)malloc(newmem*6);
RR = CC + temp+2;
SS = RR + temp+2;
EE = SS + temp+2;
DD = EE + temp+2;
FF = DD + temp+2;
        Ii =mid_i; Jj = mid_j;
        recu_al(&AA[Ii], &BB[Jj], endi-mid_i, temp, q, te);
        }
else
   if (nodeid%2 == 0) {
temp = mid_j-sttj+1;
newmem = temp *4 + 4;
S = sapp = (int *)malloc(newmem);
CC = (int *)malloc(newmem*6);
RR = CC + temp+1;
SS = RR + temp+1;
EE = SS + temp+1;
DD = EE + temp+1;
FF = DD + temp+1;
        Ii = stti -1; Jj = sttj -1;
        recu_al(&AA[Ii], &BB[Jj], mid_i-stti, temp, tb, 0);
        DEL(2);
    }
   else  {
temp = endj-mid_j;
newmem = temp *4 + 8;
S = sapp = (int *)malloc(newmem);
CC = (int *)malloc(newmem*6);
RR = CC + temp+2;
SS = RR + temp+2;
EE = SS + temp+2;
DD = EE + temp+2;
FF = DD + temp+2;
        Ii =mid_i+1; Jj = mid_j;
        recu_al(&AA[Ii], &BB[Jj],endi-mid_i-1,temp,0,te);
```

```
            }
    free(CC);
        }
}


/* recu_al(A,B,M,N,tb,te) returns the score of an optimum conversion between
   A[1..M] and B[1..N] that begins(ends) with a delete if tb(te) is zero
   and appends such a conversion to the current script.                  */

recu_al(A,B,M,N,tb,te) char *A, *B; int M, N; int tb, te;

{ int   midi, midj, type;  /* Midpoint, type, and cost */
  int  midc;

{ register int   i, j;
  register int c, e, d, s;
        int t, *va;
  pairptr  z1;

/* Boundary cases: M <= 1 or N == 0 */

  if (N <= 0)
    { if (M > 0) DEL(M)
      return - gap(M);
    }
  if (M <= 1)
    { if (M <= 0)
        { INS(N);
          return - gap(N);
        }
      if (tb > te) tb = te;
      midc = - (tb + r + gap(N) );
      midj = 0;
      va = CV[A[1]];
      for (j = 1; j <= N; j++)
        {  for ( tt = 1, z = col[Jj+J+j]; z != 0; z = z->NEXT )
             if ( z->ROW == Ii+I+1 )
                  { tt = 0; break; }
           if ( tt )
            { c = va[B[j]] - ( gap(j-1) + gap(N-j) );
              if (c > midc)
               { midc = c;
                 midj = j;
              }
              }
        }
      if (midj == 0)
        { INS(N) DEL(1) }
      else
        { if (midj > 1) INS(midj-1)
          REP
```

```
        if ( A[1] == B[midj] )
          no_mat += 1;
        else
          no_mis += 1;
        /* mark (A[I],B[J]) as used: put J into list row[I] */
      I++; J++;
        z1 = ( pairptr ) malloc(sizeof(pair));
      z1->ROW = I+Ii;
      z1->NEXT = col[Jj+J];
        col[Jj+J] = z1;
      if (midj < N) INS(N-midj)
     }
   return midc;
  }

/* Divide: Find optimum midpoint (midi,midj) of cost midc */

  midi = M/2;                    /* Forward phase:                    */
  CC[0] = 0;                     /*   Compute C(M/2,k) & D(M/2,k) for all k */
  t = -q;
  for (j = 1; j <= N; j++)
    { CC[j] = t = t-r;
      DD[j] = t-q;
    }
  t = -tb;
  for (i = 1; i <= midi; i++)
    { s = CC[0];
      CC[0] = c = t = t-r;
      e = t-q;
      va = CV[A[i]];
      for (j = 1; j <= N; j++)
        { if ((c = c - qr) > (e = e - r)) e = c;
          if ((c = CC[j] - qr) > (d = DD[j] - r)) d = c;
            DIAG(i+I+Ii, j+J+Jj, c, s + va[B[j]])
          if (c < d) c = d;
          if (c < e) c = e;
          s = CC[j];
          CC[j] = c;
          DD[j] = d;
        }
    }
  DD[0] = CC[0];

  RR[N] = 0;                     /* Reverse phase:                    */
  t = -q;                        /*   Compute R(M/2,k) & S(M/2,k) for all k */
  for (j = N-1; j >= 0; j--)
    { RR[j] = t = t-r;
      SS[j] = t-q;
    }
  t = -te;
  for (i = M-1; i >= midi; i--)
    { s = RR[N];
```

```
      RR[N] = c = t = t-r;
      e = t-q;
      va = CV[A[i+1]];
      for (j = N-1; j >= 0; j--)
        { if ((c = c - qr) > (e = e - r)) e = c;
          if ((c = RR[j] - qr) > (d = SS[j] - r)) d = c;
            DIAG(i+1+I+Ii, j+1+J+Jj, c, s+va[B[j+1]])
          if (c < d) c = d;
          if (c < e) c = e;
          s = RR[j];
          RR[j] = c;
          SS[j] = d;
        }
    }
  SS[N] = RR[N];
  midc = CC[0]+RR[0];              /* Find optimal midpoint */
  midj = 0;
  type = 1;
  for (j = 0; j <= N; j++)
    if ((c = CC[j] + RR[j]) >= midc)
      if (c > midc || CC[j] != DD[j] && RR[j] == SS[j])
        { midc = c;
          midj = j;
        }
  for (j = N; j >= 0; j--)
    if ((c = DD[j] + SS[j] + q) > midc)
      { midc = c;
        midj = j;
        type = 2;
      }
}

/* Conquer: recursively around midpoint */

  if (type == 1)
    { recu_al(A,B,midi,midj,tb,q);
      recu_al(A+midi,B+midj,M-midi,N-midj,q,te);
    }
  else
    { recu_al(A,B,midi-1,midj,tb,zero);
      DEL(2);
      recu_al(A+midi+1,B+midj,M-midi-1,N-midj,zero,te);
    }
  return midc;
}


/************************************/
int *fst(sti, eni, stj, enj, odnd, head, end, tb, te)
int sti, eni, stj, enj, odnd, head, end, tb, te;
{
   struct nmsg whisper1, score;
```

```
  int in_a[6], j, i, *va;
  int t, sj, leni, lenj, P_g, content[3];
  register int e,s,c,d;     /* e --- l[i][j-1]    */
                            /* s --- C[i-1][j-1] */
                            /* c --- C[i][j-1]   */

#define fst_pass(in, baj, bai, N, head, out)                    \
{                                                               \
     s = in[0];                                                 \
     c = in[1];                                                 \
     e = in[2];                                                 \
     va = CV[AA[bai]];                                          \
     for (j=((nodeid==head)? 1:0); j < N; j++)                  \
       { if ((c = c - qr) > (e = e - r)) e = c;                 \
         if ((c = CC[j] - qr) > (d = DD[j] - r)) d = c;         \
           DIAG(bai, j+baj, c, s+va[BB[j+baj]])                 \
           if (c < d) c = d;                                    \
         if (c < e) c = e;                                      \
         s = CC[j];                                             \
         CC[j] = c;                                             \
         DD[j] = d;                                             \
       }                                                        \
     out[0] = s;                                                \
     out[1] = c;                                                \
     out[2] = e;                                                \
}


  if (enj-stj<=0) {
    score.nh_data[0] = stj-1;
    score.nh_data[1] = 1;
    return(score.nh_data);
  }
  P_g = end-head+1;
  leni = eni-sti+1;                                 /*  length    */
  if (nodeid == head) {
        lenj = (enj-stj+2)/P_g;
        if ((enj-stj+2)%P_g != 0) ++lenj;
    if (head!=end) {
        whisper1.nh_dl_event = dl_event(desti);
        whisper1.nh_event = c_stt+end;
        whisper1.nh_type = whisper_msg;
        whisper1.nh_data[4] = stj+lenj-1;                   /* stt point */
        whisper1.nh_data[5] = lenj;                 /*  length of j */
        whisper1.nh_length = 0;
        whisper1.nh_flags = FLAG1;
        whisper1.nh_node = nodeid + 1;
      dsend(&whisper1);
     }
        t = -q;
        CC[0] = 0;
        for (j=1; j<=lenj; j++) {
```

```
                CC[j] = t = t-r;
                DD[j] = t-q;
                }
        t = -tb;
        for (i=0; i<leni; i++) {
            in_a[0] = CC[0];                    /* s -- C[i-1][j-1] */
                in_a[1] = CC[0] = t = t-r;      /* c -- C[i][j-1]  */
                in_a[2] = t-q;                  /* e -- I[i][j-1] */
                fst_pass(in_a, stj-1, sti+i, lenj, head, content)
                if (head!=end) {
            for (j=0; j<4; j++) whisper1.nh_data[j] = content[j];
                whisper1.nh_node = nodeid + 1;
                dsend(&whisper1);
            }
        }
       DD[0] = CC[0];
}
else {
       whisper1.nh_event = c_stt+end;
      whisper1.nh_type = whisper_msg;
      whisper1.nh_length = 0;
      whisper1.nh_flags = 0;
      drecv(&whisper1);
        t=-q-(whisper1.nh_data[4]-stj)*r;
        sj = whisper1.nh_data[4];
        lenj = whisper1.nh_data[5];
      if (nodeid != end) {
                whisper1.nh_dl_event = dl_event(desti);
                whisper1.nh_event = c_stt+end;
                whisper1.nh_type = whisper_msg;
                whisper1.nh_data[4] = sj+lenj;          /* stt point */
                whisper1.nh_data[5] = lenj;             /* length */
                whisper1.nh_length = 0;
                whisper1.nh_flags = FLAG1;
                whisper1.nh_node = nodeid + 1;
                dsend(&whisper1);
                }
        else     lenj = enj-sj+1;
        for (j=0; j<lenj; j++) {
                CC[j] = t = t-r;
                DD[j] = t-q;
                }
      for (i=0; i<leni; i++)
      {
          whisper1.nh_type = p_d;
          drecv(&whisper1);
          fst_pass(whisper1.nh_data, sj, sti+i, lenj, head, content)
              if (nodeid != end) {
                whisper1.nh_type = p_d;
                whisper1.nh_node = nodeid + 1;
                for (j=0; j<4; j++) whisper1.nh_data[j] = content[j];
                dsend(&whisper1);
```

```
                }
            }
        }
/*      Send CC and DD to the correspondent node */
    score.nh_event = COST;
    score.nh_type = ARRAY;
    score.nh_length = lenj*4;
    score.nh_flags = DINTMSG;
    score.nh_msg = CC;
    score.nh_node = odnd-nodeid+head;
    nsend(&score);
    score.nh_event = COST+1;
    score.nh_msg = DD;
    nsend(&score);

/*      Receive the node on optimal path        */

    score.nh_event = ARRAY;
    score.nh_type = COST;
    score.nh_length = 0;
    score.nh_flags = DINTDATA;
    nrecv(&score);

    return(score.nh_data);
}

/************************************/
int *snd(sti, eni, stj, enj, odnd, head, end, tb, te)
int sti, eni, stj, enj, odnd, head, end, tb, te;
{
  struct nmsg whisper1, score1;
  int in_a[6], j, i, midj, midc, type, t,sj, *result, *find_mid();
  int leni, lenj, P_g, temp_cost, *va, sjj;

/* snd_pass(in, baj, bai, N) compute the locall RR and SS. in ins a input
   array, baj (base of j) and bai (base of i) and N (range of j).    */
   register int e,              /* I[i][j-1]    */
                s,              /*  C[i-1][j-1] */
                c,              /*  C[i][j-1]   */
                d;
#define snd_pass(in, baj, bai, N, head, out)            \
{                                                       \
     s = in[0];                                         \
     c = in[1];                                         \
     e = in[2];                                         \
     va = CV[AA[bai]];                                  \
     for (j = N-1; j >= 0; j--)                         \
         { if ((c = c - qr) > (e = e - r)) e = c;       \
        if ((c = RR[j] - qr) > (d = SS[j] - r)) d = c;  \
          DIAG(bai, j+baj, c, s+va[BB[j+baj]])          \
        if (c < d) c = d;                               \
        if (c < e) c = e;                               \
```

```
        s = RR[j];                                          \
        RR[j] = c;                                          \
        SS[j] = d;                                          \
       }                                                    \
     out[0] = s;                                            \
     out[1] = c;                                            \
     out[2] = e;                                            \
}

  if (enj-stj<=0) {
    score1.nh_data[0] = stj-1;
    score1.nh_data[1] = 1;
    return(score1.nh_data);
  }
  P_g = end-head+1;
  leni = eni-sti+1;                                         /* length    */
  if (nodeid == head) {
        whisper1.nh_data[5]=lenj = (enj-stj+2)/P_g;
        if ((enj-stj+2)%P_g != 0) {
                whisper1.nh_data[5] = ++lenj;
                lenj = enj-stj+2-lenj*(P_g - 1);            /* length of j */
        }
        t = -q;
        for (j=lenj-2; j>=0; j--) {
                RR[j] = t = t-r;
                SS[j] = t-q;
                }
    if (head!=end) {
        whisper1.nh_dl_event = dl_event(desti);
        whisper1.nh_event = p_d+end;
        whisper1.nh_type = whisper_msg;
        whisper1.nh_data[4] = enj-lenj+1;       /* end position */
        whisper1.nh_data[3] =  t;
        whisper1.nh_length = 0;
        whisper1.nh_flags = FLAG1;
        whisper1.nh_node = nodeid + 1;
     dsend(&whisper1);
     }
        sj = enj - lenj+2;
        RR[lenj-1] = 0;
        t = -te;
        for (i=leni-1; i>=0; i--) {
            in_a[0] = RR[lenj-1];                   /* s -- C[i-1][j-1] */
                in_a[1] = RR[lenj-1] = t = t-r;             /* c -- C[i][j-1]  */
                in_a[2] = t-q;                      /* e -- I[i][j-1] */
                snd_pass(in_a, sj, sti+i, lenj-1, head, whisper1.nh_data)
                if (head!=end) {
                whisper1.nh_type = c_stt;
                whisper1.nh_node = nodeid + 1;
                dsend(&whisper1);
                }
          }
```

```
      SS[lenj-1] = RR[lenj-1];
    }
    else {
        whisper1.nh_event = p_d+end;
        whisper1.nh_type = whisper_msg;
          whisper1.nh_flags = 0;
        whisper1.nh_length = 0;
        drecv(&whisper1);
          t = whisper1.nh_data[3];
          enj = whisper1.nh_data[4];
          lenj = whisper1.nh_data[5];
          for (j=lenj-1; j>=0; j--) {
                RR[j] = t = t-r;
                SS[j] = t-q;
                }
        sj = enj - lenj+1;
          if (nodeid != end) {
                whisper1.nh_dl_event = dl_event(desti);
                whisper1.nh_event = p_d+end;
                whisper1.nh_type = whisper_msg;
                whisper1.nh_data[3] = t;
                whisper1.nh_data[4] = enj-lenj;          /* stt point */
                whisper1.nh_data[5] = lenj;              /* length */
                whisper1.nh_length = 0;
                whisper1.nh_flags = FLAG1;
                whisper1.nh_node = nodeid + 1;
                dsend(&whisper1);
                }
        for (i=leni-1; i>=0; i--)
        {
            whisper1.nh_type = c_stt;
            drecv(&whisper1);
            snd_pass(whisper1.nh_data,sj,sti+i,lenj,head,whisper1.nh_data)
                if (nodeid != end) {
                  whisper1.nh_type = c_stt;
                  whisper1.nh_node = nodeid + 1;
                  dsend(&whisper1);
                }
          }
        }

/*      receive CC and DD from the correspondent node */
    score1.nh_event = COST;
    score1.nh_type = ARRAY;
    score1.nh_flags = DINTMSG;
    score1.nh_length = lenj*4;
    score1.nh_msg=CC;
    nrecv(&score1);

    score1.nh_event = COST+1;
    score1.nh_msg=DD;
    nrecv(&score1);
```

```
/*      find the node on optimal path    */

    midc = CC[0]+RR[0];                    /* Find optimal midpoint */
    midj = sj-1;
    type = 1;
    for (j = 0; j < lenj; j++)
        if ((temp_cost = CC[j] + RR[j]) >= midc)
            if (temp_cost > midc || CC[j] != DD[j] && RR[j] == SS[j])
        { midc = temp_cost;
                midj = j+sj-1;
              }
    for (j = lenj-1; j >= 0; j--)
        if ((temp_cost = DD[j] + SS[j] + q) > midc)
            { midc = temp_cost;
              midj = j+sj-1;
              type = 2;
             }

  if (end-head>0) {
    result = find_mid(head, end, midc, midj, type);
    midc = result[0];
    midj = result[1];
    type = result[2];
        }
/*      distribute the mid_i, mid_j.       */

    score1.nh_event = ARRAY;
    score1.nh_type = COST;
    score1.nh_length = 0;
    score1.nh_flags = DINTDATA;
    score1.nh_data[0] = midj;
    score1.nh_data[1] = type;
    score1.nh_node = end-nodeid+odnd;
    nsend(&score1);
    return(score1.nh_data);
}



/*      findmid() return a pointer with point to an three element array
        which contains mid_cost, mid_j, type.    */

int *find_mid(hdnd, tlnd, mid_c, md_j, type)
int hdnd, tlnd, mid_c, md_j, type;
{
    struct nmsg  max_element;
    int i, n, d;

    if (nodeid == hdnd) {        /* headnode send its max  */
        max_element.nh_dl_event = dl_event(desti);
        max_element.nh_event = PASS;
        max_element.nh_type = OPT;
```

```
        max_element.nh_length = 0;
        max_element.nh_flags = FLAG;
        max_element.nh_data[0] = mid_c;                 /* max cost      */
        max_element.nh_data[1] = md_j;                  /* j_index of max cost */
        max_element.nh_data[2] = type;                  /* type of connection  */
        max_element.nh_node = nodeid + 1;
        dsend(&max_element);
        }
    else
        {     /* the other nodes receive and compare, sent new max */
        max_element.nh_event = PASS;
        max_element.nh_type = OPT;
        max_element.nh_length = 0;
        max_element.nh_flags = 0;
        drecv(&max_element);
        if (max_element.nh_data[0] < mid_c)
                {
                max_element.nh_data[0] = mid_c;
                max_element.nh_data[1] = md_j;
                max_element.nh_data[2] = type;
                }
        max_element.nh_dl_event = dl_event(desti);
        max_element.nh_length = 0;
        max_element.nh_flags = FLAG;
        max_element.nh_node = nodeid + 1;
        if (nodeid == tlnd) {
                max_element.nh_type = BC;
                max_element.nh_node = hdnd;
                nsend(&max_element);}                                   /*send*/
        else dsend(&max_element);
        }

/*      passing globle max and add node(i,j) in the memo structure  */
    max_element.nh_event = PASS;
    max_element.nh_type = BC;
    max_element.nh_length = 0;
    max_element.nh_flags = 0;
    nrecv(&max_element);
    max_element.nh_dl_event = dl_event(desti);
    max_element.nh_length = 0;
    max_element.nh_flags = FLAG;
    max_element.nh_node = nodeid + 1;
    max_element.nh_event = PASS;
    max_element.nh_type = BC;
    if (nodeid != tlnd) dsend(&max_element);
    return(max_element.nh_data);
}
```

```
/* gather() send the conversion of the optimal alignment to the HOST  */
gather()
{
        int i;
        struct nmsg updt1;
        updt1.nh_event = COLLECT+nodeid;
        updt1.nh_type = AL;
        updt1.nh_flags = DINTMSG|DINTDATA;
        updt1.nh_node = 100;
        updt1.nh_length = Sl*4;                         /* length of al. */
        updt1.nh_msg = S;
/*      if (nodeid==15) tprintf("S = %d\n",S); */
        updt1.nh_data[0] = Ii;                  /* i of A */
        updt1.nh_data[1] = Jj;                  /* j of B */
        updt1.nh_data[3] = I;
        updt1.nh_data[4] = J;
        updt1.nh_data[5] = al_len;
        updt1.nh_data[2] = MAX.SCORE;
        updt1.nh_data[6] = no_mat;
        updt1.nh_data[7] = no_mis;
        nsend(&updt1);
/*      if (nodeid==2) for (i=0; i<Sl; i++) tprintf("S[%d]=%d\n",i,S[i]);
        tprintf("Sl =%d results of node %d out\n", Sl, nodeid);*/
        free(S);
        return(0);
}

/*      update(Mn) updates the used_pair table in each node. It is done by
        passing a token in among nodes. The holder of the token send its
        used_pair table to the others which update their used_pair table
        based on the received message.                  */
update(Mn, Nn)
int Mn, Nn;             /* length of the message       */

{
        int  *block;
        int i,j, h, index, t_flag;
        pairptr zz;
        struct nmsg updt;
/*  construct message block     (j,i) (j,i) ...  */

        block = (int *)malloc(Nn*2);
        index = 0;
        for (j=Jj; j<=Jj+J+1; j++)
                if (col[j]!=NULL) {
                        block[index++] = j;
                        block[index++] = col[j]->ROW;
                }
        updt.nh_dl_event = dl_event(desti);
        updt.nh_event = AL;
        updt.nh_type = COLLECT;
        updt.nh_data[0] = index;
```

```
        updt.nh_flags = DINTDATA|DINTMSG;
        updt.nh_msg = block;
        updt.nh_node = (nodeid + 1)%P;
        updt.nh_length = index * 4;
        nsend(&updt);

        for (i=0; i<P-1; i++) {
                                        /*  receive block          */
                updt.nh_event = AL;
                updt.nh_type = COLLECT;
                updt.nh_flags = 0;
                updt.nh_length = Nn*2;
                updt.nh_msg = block;
                nrecv(&updt);
                index = updt.nh_data[0];
/*          update the used_pair table  */

                for (h=0; h<updt.nh_data[0]; h++) {
                   for (z=col[block[h]],t_flag=1; z!=0; z=z->NEXT)
                        if (z->ROW==block[h+1]) {
                                t_flag=0; break; }
                   if (t_flag==1) {
                        zz = (pairptr) malloc(sizeof(pair));
                        zz->NEXT = col[block[h]];
                        col[block[h]]=zz;
                        zz->ROW = block[h+1];
                        }
                   h++;
                }
                updt.nh_dl_event = dl_event(desti);
                updt.nh_event = AL;
                updt.nh_type = COLLECT;
                updt.nh_msg = block;
                updt.nh_length = index*4;
                updt.nh_data[0] = index;
                updt.nh_node = (nodeid + 1)%P;
                updt.nh_flags = DINTDATA|DINTMSG;
                if (i!=P-2) nsend(&updt);

        }

        free(block);
        return(0);
}


/* function reverse(A, B, sri, srj, eri, erj) modifies the alignment LIST
using reversed dynamic programming. A and B are two sequences, sri and srj
are EXACTLY starting points, and eri and erj are ending point of computing.
 Note: sri > eri and srj > erj      */

static int tflag, lflag;      /*  top flag and left flag    */
```

```
reverse(A, B, sri, srj, eri, erj)
char A[], B[];
int sri, srj, eri, erj;
{
    struct nmsg chat;
    int  M_l, N_l, infor[8];
    int stri, strj, leni, lenj, B_len;
    int temp, inc, enri, enrj;
    long i;
    int rl, cl;     /*      rl — row limit, cl — colume limit */
                    /*      tb — top boundary, lb — left boundary    */
  register int   j, jj1;            /* row and column indices */
  register int   c;                         /* best score at current point */
  register int   f;                         /* best score ending with insertion */
  register int   d;                         /* best score ending with deletion */
  register int   p;                         /* best score at (i-1, j-1) */
  register int   ci, cj;            /* end-point associated with c */
  register int   di, dj;            /* end-point associated with d */
  register int   fi, fj;            /* end-point associated with f */
  register int   pi, pj;            /* end-point associated with p */
  register int    *va;                      /* pointer to v(A[i], B[j]) */
  int   bj;                                 /* function for inserting a node */
  register int    content1[2];

        /* Compute the matrix and save the top K best scores in LIST
           CC : the scores of the current row
           RR and EE : the starting point that leads to score CC
           DD : the scores of the current row, ending with deletion
           SS and FF : the starting point that leads to score DD       */
#define rvpass(package_i, package_o, i0, basej, sizej)                  \
{                                                                       \
        va = CV[AA[i0]];                                                \
        p = package_i[0];                                               \
        pi= package_i[1];                                               \
        pj= package_i[2];                                               \
        unpack(package_i[6], content1)                                  \
        c = content1[0];                                                \
        f = content1[1];                                        \
        ci= package_i[7];                                               \
        cj= package_i[3];                                               \
        fi= package_i[4];                                               \
        fj= package_i[5];                                               \
        tflag = 0;                                                      \
        bj = basej - sizej+1;                                           \
        for ( j=jj1=(nodeid==P-1 ? basej : sizej-1); j >=jj1-sizej+1; j--)  \
          {                                                             \
          f = f - r;        /* r, q, N_l and min are globles */         \
          c = c - qr;               \
          ORDER(f, fi, fj, c, ci, cj)                                   \
          c = CC[j] - qr;                                               \
          ci = RR[j];                                                   \
          cj = EE[j];                                                   \
```

```
  d = DD[j] − r;                                          \
  di = SS[j];                                                \
  dj = FF[j];                                                \
  ORDER(d, di, dj, c, ci, cj)                                \
  c = 0;                                                  \
  if (nodeid == P−1)  DIAG(i0, j, c, p + va[BB[j]])  /* diagonal */ \
  else                                                       \
    DIAG(i0, j+bj, c, p + va[BB[j+bj]])                   /* diagonal */ \
  if ( c <= 0 )                                              \
    { c = 0; ci = i0; cj = (nodeid==P−1 ? j : j+bj);  \
          }                                                  \
  else                                                       \
    { ci = pi; cj = pj;                                   \
          }                                                  \
  ORDER(c, ci, cj, d, di, dj)                                \
  ORDER(c, ci, cj, f, fi, fj)                                \
  p = CC[j];                                                 \
  CC[j] = c;                                                 \
  pi = RR[j];                                                \
  pj = EE[j];                                                \
  RR[j] = ci;                                                \
  EE[j] = cj;                                                \
  DD[j] = d;                                                 \
  SS[j] = di;                                                \
  FF[j] = dj;                                                \
  if ( c > min )   /* add the score into list */          \
          flag = 1;                                       \
  if (tflag==0&&(ci>tb && cj>lb || di>tb && dj>lb || fi>tb && fj>lb))\
    { tflag = 1; mm1=i0; }                                   \
}                                       /* K is globle too */      \
if (nodeid != P−1) {                                       \
   package_o[0] = p;                                       \
   package_o[1] = pi;                                      \
   package_o[2] = pj;                                      \
   pack(c, f, package_o[6])                                \
   package_o[7] = ci;                                      \
   package_o[3] = cj;                                      \
   package_o[4] = fi;                                      \
   package_o[5] = fj;                                      \
}                                                          \
else {                                                     \
   if (!lflag&&(ci>tb && cj>lb || di>tb && dj>lb || fi>tb && fj>lb))\
          { lflag = 1; nn1 = j; }                          \
    HH[i0] = p;                                            \
    II[i0] = pi;                                           \
    JJ[i0] = pj;                                           \
    pack(c, f, UU[i0])                                     \
    VV[i0] = ci;                                           \
    WW[i0] = cj;                                           \
    XX[i0] = fi;                                           \
    YY[i0] = fj;                                           \
}                                                          \
```

```
}


    M_l = sri – eri + 1;
    N_l = srj – erj + 1;
    temp = (srj – erj+1)/(P–1);
    enri = eri; enrj = erj;
    if (nodeid < P–1) {
        CC = (int *)malloc(temp*24);
        DD = CC + temp;
        RR = DD + temp;
        EE = RR + temp;
        FF = EE + temp;
        SS = FF + temp;
        }
    else {
        temp = 16+erj;
        CC = (int *)malloc(temp*24);
        DD = CC + temp;
        RR = DD + temp;
        EE = RR + temp;
        FF = EE + temp;
        SS = FF + temp;

        temp = sri;
        UU = (int *)malloc(temp*32);
        VV = UU + temp;
        HH = VV + temp;
        II = HH + temp;
        JJ = II + temp;
        WW = JJ + temp;
        XX = WW + temp;
        YY = XX + temp;
        }

    if (nodeid == 0) {
        chat.nh_dl_event = dl_event(desti);
        chat.nh_event = rvcmd;
        chat.nh_type = chat_msg;
        chat.nh_data[1] = N_l;
        chat.nh_data[2] = sri;                  /* start point of A */
        chat.nh_data[3] = M_l;                  /*  length of A   */
        lenj = (N_l–1)/(P–1);                   /*  length of B   */
      chat.nh_data[4] = srj–lenj;               /* start point of B */
        chat.nh_data[5] = lenj;
        chat.nh_length = 0;
        chat.nh_flags = FLAG1;
        chat.nh_node = nodeid + 1;
      dsend(&chat);
        for (j=srj, jj1=lenj; j>srj–lenj; j—, jj1—) {
                CC[jj1] = 0;
```

```
                RR[jj1] = sri+1;
                EE[jj1] = j;
                DD[jj1] = -q;
                SS[jj1] = sri+1;
                FF[jj1] = j;

                }
    infor[0] = 0;                       /* p[i-1][j-1] */
    infor[2] = srj+1;                   /* pj  */
    infor[3] = srj+1;                   /* cj  */
    infor[5] = srj+1;                   /* fj  */
    pack(0, -q, infor[6])               /* c[i][j-1] and f[i][j-1] */
      for (i=sri; i>sri - M_l;i-- )
      {
                infor[1] = i+1;                 /* pi */
                infor[4] = i;                   /* fi  */
                infor[7] = i;                   /* ci  */
                chat.nh_event = rvdt + i;
                chat.nh_type = c_start;
                chat.nh_length = 0;
                chat.nh_node = nodeid + 1;
                rvpass(infor, chat.nh_data, i, srj, lenj)
                dsend(&chat);
                }
      }
else {
      chat.nh_event = rvcmd;
      chat.nh_type = chat_msg;
      chat.nh_length = 0;
      chat.nh_flags = 0;
        drecv(&chat);
        B_len = chat.nh_data[1];
        stri = chat.nh_data[2];
        strj = chat.nh_data[4];
        leni = chat.nh_data[3];
        lenj = chat.nh_data[5];
      if (nodeid != P -1 ) {
                chat.nh_dl_event = dl_event(desti);
                chat.nh_data[1] = B_len;                /* start point */
                chat.nh_data[2] = stri;                 /* start point */
                chat.nh_data[3] = leni;                 /* length */
                chat.nh_data[4] = strj-lenj;            /* start point */
                chat.nh_data[5] = lenj;                 /* length */
                if (nodeid == P-2) {
                   chat.nh_data[5] = B_len - (P-1)*lenj;
                }
                chat.nh_event = rvcmd;
                chat.nh_type = chat_msg;
                chat.nh_length = 0;
                chat.nh_flags = FLAG1;
                chat.nh_node = nodeid + 1;
                dsend(&chat);
```

```
    }
      for (j=strj, jj1=nodeid==P-1 ? strj : lenj-1; j>strj-lenj; j--,jj1--)
       {
            CC[jj1] = 0;
            RR[jj1] = stri + 1;
            FF[jj1] = EE[jj1] = j;
            DD[jj1] = -q;
            SS[jj1] = stri + 1;
            }
    for (i=stri; i>= enri; i--)
    {
            chat.nh_event = rvdt+i;
        chat.nh_flags = FLAG1;

        chat.nh_type = c_start;
        chat.nh_length = 0;
        drecv(&chat);

            rvpass(chat.nh_data, chat.nh_data, i, strj, lenj)
            if (nodeid != P-1) {
                    chat.nh_dl_event = dl_event(desti);
              chat.nh_flags = FLAG1;
              chat.nh_type = c_start;
                    chat.nh_event = rvdt+i;
              chat.nh_length = 0;
              chat.nh_node = nodeid + 1;
              dsend(&chat);
            }
    }
}

if (nodeid == 0) {
lflag = tflag = 1;
for (tb=rl=mm1=enri+1, lb=cl=nn1=enrj+1; ;)
{
  if ( lb==1 && tb==1 || ex_limit()) break;
     for (; (tflag && enri > 1) || (lflag && enrj > 1); )
     {
       if (tflag && enri > 1)
            rl = (rl - 512>0 ? rl -512 : 1);
       if (lflag && enrj > 1)
            cl = (cl - 512>0 ? cl - 512 : 1);
       chat.nh_dl_event = dl_event(desti);
       chat.nh_event = rvcmd;
       chat.nh_type = chat_msg;
       chat.nh_data[0] = 111;
       chat.nh_data[2] = rl;
       chat.nh_data[1] = cl;                    /*  */
       chat.nh_data[3] = tb;
       chat.nh_data[4] = lb;                    /*  */
       chat.nh_length = 0;
       chat.nh_flags = FLAG1;
```

```
        chat.nh_node = nodeid + 1;
      dsend(&chat);
        lflag = 0;
      infor[0] = 0;                          /* p[i-1][j-1] */
      infor[2] = srj+1;                      /* pj  */
      infor[3] = srj+1;                      /* cj  */
      infor[5] = srj+1;                      /* fj  */
      pack(0, - q, infor[6])      /* c[i][j-1] and f[i][j-1] */
        for (i=enri-1; i>=rl;i— )
         {
        infor[1] = i+1;                      /* pi */
        infor[4] = i;                        /* fi  */
        infor[7] = i;                        /* ci  */
              chat.nh_event = rvdt;
        chat.nh_type = c_start1;
          chat.nh_length = 0;
          chat.nh_node = nodeid + 1;
          rvpass(infor, chat.nh_data, i, srj, lenj)
          dsend(&chat);
         }
      /* receive lflags and tflags        */
      for (i = 1; i<P; i++) {
              chat.nh_event = p_d;
              chat.nh_type = flag_msg;
              chat.nh_length = 0;
              chat.nh_flags = FLAG;
              nrecv(&chat);
              if (chat.nh_data[0]==1) lflag = 1;
              if (chat.nh_data[1]==1) tflag = 1;
              if (chat.nh_data[2]<mm1) mm1 = chat.nh_data[2];
              nn1 = chat.nh_data[3];

              }
      tprintf("tflag %d  lflag %d rl %d cl %d tb %d lb %d\n",tflag, lflag,rl,cl,tb,lb);
      enri = m1 = rl;   enrj = n1 = cl;
      }
      tprintf("rl = %d   cl = %d \n", m1, n1);
      }
/*    send stop command     */
      chat.nh_dl_event = dl_event(desti);
      chat.nh_event = rvcmd;
      chat.nh_type = chat_msg;
      chat.nh_data[0] = 0;
      chat.nh_data[1] = m1 =mm1;
      chat.nh_data[2] = n1 =nn1;
      chat.nh_length = 0;
      chat.nh_node = 1;
    dsend(&chat);

      free(CC);
      return(1);
  }
```

```
else if (nodeid == P-1)
{
  while(1) {
     chat.nh_dl_event = dl_event(desti);
   chat.nh_event = rvcmd;
   chat.nh_type = chat_msg;
   chat.nh_length = 0;
   chat.nh_flags = FLAG1;
     drecv(&chat);
   if (chat.nh_data[0] == 0) {
          m1 = chat.nh_data[1];
          n1 = chat.nh_data[2];
          free(UU);
          free(CC);
          return(1);
     }
     cl = chat.nh_data[1];
   rl = chat.nh_data[2];
     tb = chat.nh_data[3];
   lb = chat.nh_data[4];
   for (j=enrj-1; j>=cl; j--) {  /*   j should not ended at 0  */
                CC[j] = 0;
                RR[j] = sri + 1;
                EE[j] = j;
                DD[j] = -q;
                SS[j] = sri + 1;
                FF[j] = j;
                }
     inc = enrj-cl;
     lflag = 0;
   for (i=sri; i>=enri && cl <= enrj; i--)
   {
                chat.nh_data[0] = HH[i];
                chat.nh_data[1] = II[i];
                chat.nh_data[2] = JJ[i];
                chat.nh_data[6] = UU[i];
                chat.nh_data[7] = VV[i];
                chat.nh_data[3] = WW[i];
                chat.nh_data[4] = XX[i];
                chat.nh_data[5] = YY[i];
                rvpass(chat.nh_data, chat.nh_data, i, enrj-1, inc)
                 }
   for (i=enri-1; i>= rl; i--)
   {
                chat.nh_event = rvdt;
          chat.nh_flags = FLAG1;
          chat.nh_type = c_start1;
          chat.nh_length = 0;
          drecv(&chat);
               rvpass(chat.nh_data, chat.nh_data, i, strj, lenj+inc)
                                  /*          inc used          */
        }
```

```
  chat.nh_event = p_d;
  chat.nh_type = flag_msg;
  chat.nh_length = 0;
  chat.nh_flags = FLAG;
  chat.nh_data[0] = lflag;
chat.nh_data[1] = tflag;
chat.nh_data[2] = mm1;
chat.nh_data[3] = nn1;
  chat.nh_node = 0;
  nsend(&chat);
  enri = rl;        enrj = cl;        lenj=lenj+inc;
   }
  }
  else    /*        nodes from 1 to P-2     */
  {
  while (1) {
      chat.nh_dl_event = dl_event(desti);
    chat.nh_event = rvcmd;
    chat.nh_type = chat_msg;
    chat.nh_length = 0;
    chat.nh_flags = FLAG1;
      drecv(&chat);

      cl = chat.nh_data[1];
    rl = chat.nh_data[2];
      tb = chat.nh_data[3];
    lb = chat.nh_data[4];
      chat.nh_dl_event = dl_event(desti);
    chat.nh_data[1] = cl;          /*        colume limit  */
    chat.nh_data[2] = rl;          /*         row limit    */
      chat.nh_data[3] = tb;
    chat.nh_data[4] = lb;
      chat.nh_length = 0;
    chat.nh_node = nodeid + 1;
    dsend(&chat);

    if (chat.nh_data[0] == 0) {
          m1 = chat.nh_data[1];
          n1 = chat.nh_data[2];
          free(CC);
          return(1);
      }

      for (i=enri-1; i>= rl; i—)
    {
          chat.nh_event = rvdt;
      chat.nh_flags = FLAG1;

      chat.nh_type = c_start1;
      chat.nh_length = 0;
      drecv(&chat);
```

```
                rvpass(chat.nh_data, chat.nh_data, i, strj, lenj)
            chat.nh_length = 0;
            chat.nh_node = nodeid + 1;
            dsend(&chat);
                }

/*      send flags to node 0    */
        chat.nh_event = p_d;
        chat.nh_type = flag_msg;
        chat.nh_length = 0;
        chat.nh_flags = FLAG;
        chat.nh_data[0] = 0;
      chat.nh_data[1] = tflag;
      chat.nh_data[2] = mm1;
      chat.nh_data[3] = 0;
        chat.nh_node = 0;
        nsend(&chat);
        enri = rl;
        }
    }

}



ex_limit()
{
        vertexptr       cur;
        register long   i;
        for (i=0; i<numnode; i++) {
           cur = LIST[i];
           if (cur->STARI <= mm && cur->STARJ <= nn && cur->BOT >= mm1
                && cur->RIGHT >= nn1 && (cur->STARI<tb || cur->STARJ<lb))
        {
                if (cur->STARI < tb) {tb = cur->STARI; tflag = 1; }
                if (cur->STARJ < lb) {lb = cur->STARJ; lflag = 1;}
                flag = 1;
                break;
           }
        }
        if (i == numnode) return 1;
        else return 0;
}

findflag()
{
    struct nmsg  flg_element;

    if (nodeid == 0) {      /* node 0 send its flg  */
        flg_element.nh_dl_event = dl_event(desti);
        flg_element.nh_event = PICK;
        flg_element.nh_type = LOAD1;
```

```
        flg_element.nh_length = 0;
        flg_element.nh_flags = FLAG;
        flg_element.nh_data[0] = flag;
        flg_element.nh_node = nodeid + 1;
        dsend(&flg_element);
        }
    else
        {     /* the other nodes receive and compare, sent new flg */
        flg_element.nh_event = PICK;
        flg_element.nh_type = LOAD1;
        flg_element.nh_length = 0;
        flg_element.nh_flags = FLAG;
        drecv(&flg_element);
        flg_element.nh_data[0] = flg_element.nh_data[0] + flag;

        flg_element.nh_dl_event = dl_event(desti);
        flg_element.nh_length = 0;
        flg_element.nh_flags = FLAG;
        flg_element.nh_node = nodeid + 1;
        if (nodeid == P-1) flg_element.nh_node = 0;
        dsend(&flg_element);                              /*send*/
        }

       /* passing globle flg and update the LIST */
    flg_element.nh_dl_event = dl_event(desti);
    flg_element.nh_event = PICK;
    flg_element.nh_type = LOAD1;
    flg_element.nh_length = 0;
    flg_element.nh_flags = FLAG;
    drecv(&flg_element);
    flag = flg_element.nh_data[0];
    flg_element.nh_node = nodeid + 1;
    if (nodeid != P-1) dsend(&flg_element);
    return(flag);
}
```

APPENDEX III OTB Program Listing

```
/* ———————— OTB Program ———————— */
#include "predefine.h"

char *A, *B;
struct nmsg mesg, cmdm;
int set[16], rfun=1, M, N, P;

main(argc,argv)
int argc;
char *argv[];

{
    int i, j, n, go, gx, ms, K, iptr,cstid;
    char symb;
    double start_time, end_time, ttime();
    FILE *Aseq, *copen();
    if (argc != 3 ) err_rept("Incorrect number of arguements", argv[0]);
    Aseq = copen(argv[1], "r");
    for (M =0; (symb = getc(Aseq)) != EOF;)
        if (symb != '\n')  ++M;
    fclose(Aseq);
    A = (char*) malloc((M+1) * sizeof(char));
    Aseq = copen(argv[1], "r");
    for (M =0; (symb = getc(Aseq)) != EOF;)
        if (symb != '\n')  A[M++] = symb;
    fclose(Aseq);

    Aseq = copen(argv[2], "r");
    for (N =0; (symb = getc(Aseq)) != EOF;)
        if (symb != '\n')  ++N;
    fclose(Aseq);
    B = (char*) malloc((N+1) * sizeof(char));
    Aseq = copen(argv[2], "r");
    for (N =0; (symb = getc(Aseq)) != EOF;)
        if (symb != '\n')  B[N++] = symb;
    fclose(Aseq);

/*  read in number of alignments, mismatch penalty,
    gap-open penalty and gap-extend penalty          */

    printf("Number of Computing Nodes (positive integer):\n");
    scanf("%d", &P);
    printf("Number of Alignments (positive integer):\n");
    scanf("%d", &K);
    printf("Matching Score is 10. \n");
    printf("Mismatching Penalty(negtive integer): \n");
    scanf("%d", &ms);
    printf("Gap-open Penalty(positive integer): \n");
    scanf("%d", &go);
    printf("Gap-expend Penalty(positive integer): \n");
    scanf("%d", &gx);
```

```
    printf("Match          Mismatch        Open-gap-penalty      Extend-gap-penalty\n");
    printf(" 10              %d                    %d                    %d \n\n", ms, go, gx);
    printf("       Upper Sequence : %s\n", argv[1]);
    printf("                length : %d\n", M);
    printf("       Lower Sequence : %s\n", argv[2]);
    printf("                length : %d\n", N);

    kinit();
/*
    cstid = 117;
    for (i=0; i<P; i++)  set[i] = i;
    if (rcast(ORIGIN, cstid, rfun, set, P)!=0)
        err_rept("Incorrect cast call", argv[0]);
*/
    mesg.nh_event = NOTE;
    mesg.nh_type = LOAD;
    mesg.nh_length = 0;
    mesg.nh_flags = DINTDATA;
    mesg.nh_data[0] = M;        /* size of the A */
    mesg.nh_data[1] = N;        /* size of the B */
    mesg.nh_data[2] = K;        /* number of alignmant */
    mesg.nh_data[3] = ms;       /* mismatch penalty    */
    mesg.nh_data[4] = go;       /* gap open penalty    */
    mesg.nh_data[5] = gx;       /* gap extend penalty  */
    mesg.nh_data[6] = P;        /* number of computing node  */
  for (i=0;i<P;i++) {
    mesg.nh_node = i;
    nsend(&mesg);
   }

    start_time = ttime();

/* broadcast the array A */
    mesg.nh_event = START;
    mesg.nh_type = LOAD;
    mesg.nh_length = M;
    mesg.nh_flags = DRAWMSG;
    mesg.nh_msg = A;
    for (i=0; i<P; i++){
        mesg.nh_node = i;
        nsend(&mesg);
        }
/* broadcast the array B */
    mesg.nh_event = NOTE;
    mesg.nh_type = LOADB;
    mesg.nh_length = N;
    mesg.nh_flags = DRAWMSG;
    mesg.nh_msg = B;
    for (i=0; i<P; i++) {
        mesg.nh_node = i;
        nsend(&mesg);}
```

```
/* issue a command to node 0   */
    cmdm.nh_event = START1;
    cmdm.nh_type = LOAD1;
    cmdm.nh_length = 0;
    cmdm.nh_flags = DINTDATA;
    cmdm.nh_node = FIRSTNODE;
    cmdm.nh_data[0] = 1;
    cmdm.nh_data[1] = M;
    cmdm.nh_data[2] = 1;
    cmdm.nh_data[3] = N;
    nsend(&cmdm);

/*   receive results
        for (i=0;i<N;i++) printf("B[%d] = %c\n", i, B[i]);
        rmelt(ORIGIN,cstid);  */
    for (i=0; i<K; i++)
        display(i);

    end_time = ttime();
    printf("start_time is %f \n", start_time);
    printf("run time is %f \n", end_time - start_time);
    kexit(0);
    free(A);
    free(B);
}


err_rept(err_msg, val)
char *err_msg, *val;
{
        fprintf(stderr, err_msg, val);
        putc('\n', stderr);
        exit(1);
}
FILE *copen(name, mode)
char *name, *mode;
{
        FILE  *fp;
        if ((fp = fopen(name, mode)) == NULL)
                err_rept("Can not open file %s \n",name);
        return(fp);
}


/* Alignment display routine */

static char ALINE[51], BLINE[51], CLINE[51];

/* long display(A1,B1,M,N,S,AP,BP) char A1[], B1[]; long M, N; long S[], AP, BP; */
display(ctl)
```

```
int ctl;
{
  struct nmsg result;
  register char *a, *b, *c;
  register int   i,  j, op;
  int   h, lines, ap, bp, *zz, M1, N1, data[16][6];
  int  *S[16], *s, no_mat, no_mis, al_len, score;
  char  *AA1, *BB1;

  for (i=0; i<P; i++) {
        S[i] = (int *)calloc((M+1+N/P)/2, sizeof(int));
        result.nh_event = COLLECT+i;
        result.nh_type = AL;
        result.nh_flags = DINTDATA | DINTMSG;
        result.nh_length = (N + M)*4;
        result.nh_msg = (char *)S[i];
        nrecv(&result);
/*  printf("In the HOST program!\n"); */
        data[i][0]=i;                              /* nodeid        */
        data[i][1]=result.nh_data[0];              /* Ii            */
        data[i][2]=result.nh_data[1];              /* Jj            */
        data[i][3]=result.nh_data[3];              /* I             */
        data[i][4]=result.nh_data[4];              /* J             */
        al_len = al_len + result.nh_data[5];       /*   al_len      */
        no_mat = no_mat + result.nh_data[6];       /* no. of match */
        no_mis = no_mis + result.nh_data[7];       /* no. of mismatch */
        if (i==0) score = result.nh_data[2];
    }


        a = ALINE;
        b = BLINE;
        c = CLINE;
        lines = 0;
        op = 0;
        ap = data[0][1]+1;
        bp = data[0][2]+1;
  for (h=0; h<P; h++) {
        i = j = 0;
        s = S[h];
        AA1 = &A[data[h][1]-1];
        BB1 = &B[data[h][2]-1];
        M1 = data[h][3];
        N1 = data[h][4];
        while (i < M1 || j < N1)
          { if (op == 0 && *s == 0)
              { op = *s++;
                *a = AA1[++i];
                *b = BB1[++j];
/*              printf("AA1[%d] =%c    ", i-1, AA1[i-1]);*/
                *c++ = (*a++ == *b++) ? '|' : ' ';
```

```
                }
              else
              { if (op == 0)
             op = *s++;
                if (op > 0)
                     { *a++ = ' ';
                       *b++ = BB1[++j];
                       op—;
                     }
                  else
                    { *a++ = AA1[++i];
                      *b++ = ' ';
                      op++;
              }
                    *c++ = '–';
               }
          if (a >= ALINE+50 || (h==P–1 && i >= M1 && j >= N1))
                { *a = *b = *c = '\0';
                  printf("\n%5d ",50*lines++);
                  for (b = ALINE+10; b <= a; b += 10)
                  printf("    .    :");
                  if (b <= a+5)
                    printf("    .");
                 printf("\n%5d %s\n      %s\n%5d %s\n",ap,ALINE,CLINE,bp,BLINE);
                    ap = data[h][1] + i + 1;
                    bp = data[h][2] + j + 1;
                  a = ALINE;
                  b = BLINE;
                  c = CLINE;
                }
          }
/*        printf("%s         \n", ALINE);  */
    }
    for (i=0; i<P; i++)  free(S[i]);
    return(0);
}
```