Parallel Pairwise Sequence Alignment Algorithm Based on Longest Common Subsequence

Pich Tantichukaitikul[1], Sattara Hattirat[1] and Jonathan H. Chan[1,2,3*]

[1]Bioinformatics and Systems Biology Program,
[2]Data and Knowledge Engineering Laboratory (D-Lab),
[3]School of Information Technology,
King Mongkut's University of Technology Thonburi, Bangkok, Thailand
*Corresponding Email: jonathan@sit.kmutt.ac.th

ABSTRACT – There is an emerging paradigm in the field of computing towards parallelism at increasing levels. Among these, multi-core processors are fast becoming the norm in the world of modern computers. The potential enhancement in performance would allow certain fundamental procedures in molecular biology, such as biological sequence alignments of DNA and protein sequences, to be done faster, paving the way for more efficient multiple genome comparison. However, in order to harness the full power of multi-core processors, effective parallel algorithms are needed. This work aimed to develop a suitable parallel longest common subsequence (LCS) algorithm for pairwise sequence alignment. The proposed parallel LCS (PLCS) performed approximately 23-30% better than the traditional serial LCS, when using the median run-time as the measure.

KEYWORDS – Parallel algorithm; longest common subsequence; biological sequence alignment; computational biology; multi-core processing

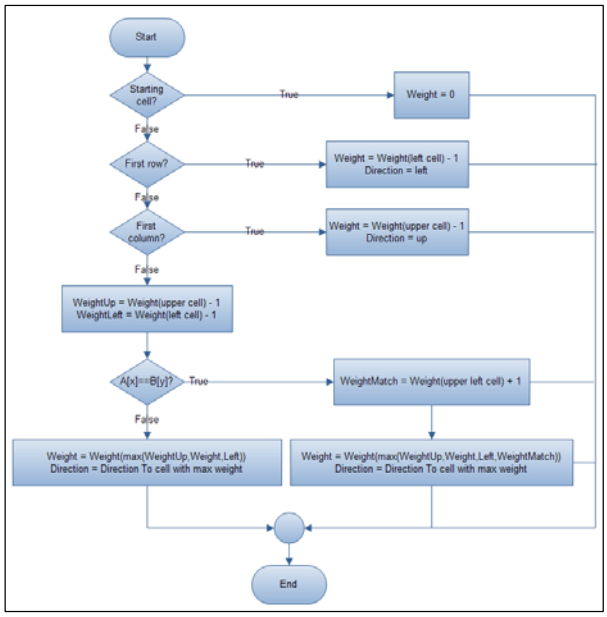
1. Introduction

The recent trend in parallelism has been moving towards ubiquitous and everyday computing devices. Implementations range from single-instruction multiple-data (SIMD) techniques at bit-level by vector processing units, to thread-level shared memory in multi-core processors, to distributed memory parallel systems in clusters and supercomputers, to massively parallel systems in clouds and server farms that may span globally. These emerging multi-core devices have been used increasingly for various scientific computing purposes [1]. For example, biological sequence analysis has been studied extensively by numerous researchers in the field of biology and bioinformatics. In order to determine the degree of similarity or homology among different organisms, sequence alignment is routinely performed using DNA and protein sequences [2]. Similarities in two or more different genomes may indicate conserved biological functions and structures [3]. That is, conserved patterns found in different biological sequences could reveal the ecological niche as well as the evolution process of these organisms [4].

To align biological sequences, it is necessary to have a scoring measure of the closeness of the sequences. The simplest measure for alignment is to use the maximum number of consecutive identities among the sequences. This is termed the longest common subsequence (LCS). There are different algorithms for solving the LCS problem. For two sequences (2LCS), this can be solved in O(mn) by using the dynamic programming technique where m and n are the lengths of the two input sequences [5]. Common algorithms for solving LCS problems include dynamic programming [6] and Hidden Markov Model [7]. The former is more accurate and would provide the optimal solution; however, it requires longer run-time as well as more computational resources. Whereas the latter runs faster and uses less resource and can produce acceptable results. In this work, we focus on the dynamic programming algorithm for LCS and aim to take advantage of the current multi-cored architectures to reduce the run-time of dynamic programming LCS.

Current computers face physical limitations such as power consumption and heat dissipation [8]. Thus, it is necessary for manufacturers to turn to building chips with multiple processor cores to increase the computational power. However, single processor

usage in a multi-core processor may not perform as fast as the latest designed single-core models. Nonetheless, they can improve overall performance by processing more work in parallel [9]. It has been found that a dual-core chip running multiple applications is typically about one and a half times faster than a comparable single core chip [9].




```
filltable(X,Y)
  if(X == 0 and Y == 0)
    WeightTable[X][Y] <- 0
  else if(X == 0)
    WeightTable[X][Y] <- WeightTable[X][Y-1] - 1
    DirTable[X][Y] <- UP
  else if(Y == 0)
    WeightTable[X][Y] <- WeightTable[X-1][Y] - 1
    DirTable[X][Y] <- left
  else
    weightUP = WeightTable[X][Y-1] - 1
    weightLEFT = <- WeightTable[X-1][Y] - 1
    if(A[X]==B[Y])
      weightM = WeightTable[X-1][Y-1] + 1
      if(weightUP > weightLEFT and weightUP > weightM)
        WeightTable[X][Y] = weightUP
        DirTable[X][Y] = UP
      else if(weightLEFT > weightUP and weightLEFT >
weightM)
        WeightTable[X][Y] = weightLEFT
        DirTable[X][Y] = LEFT
      else if(weightM > weightLEFT and weightM > weightUP)
        WeightTable[X][Y] = weightM
        DirTable[X][Y] = M
end
```

Figure 1. Flowchart and pseudo code of score filling in traditional LCS.

Moreover, there seems to be a slight lag on the development on parallelizing compilers. This could be due to the increased complications in designing a parallel algorithm. Also, parallelism works only for a restricted class of problems [8]. Consequently, many applications are not designed or have not been rewritten to run in parallel architectures. Therefore, even though we are in the multi-core era, the true advantage of the multiple execution units from the presence of multiple processors has not yet been fully realized.

Despite its importance in the biological arena, there have been limited studies for the parallel sequence alignment problem until recently. Biological sequence alignment with LCS dynamic algorithm could take longer time when the sequences become longer. This study aims to experiment on parallel pairwise alignment with LCS dynamic programming algorithm to find out if the problem of biological sequence alignment could be rewritten with parallelism and still yield optimal results. PLCS in this paper refers to parallel LCS.

The organization of the rest of the paper is as follows. Section 2 illustrates the problem formulation for both the traditional LCS problem and the parallelized version. Section 3 briefly describes the methodology used in this work. This is followed by a presentation of the results as well as relevant discussions in Section 4. Then conclusions and future works are provided in the last section of this paper.

2. Problem Formulation

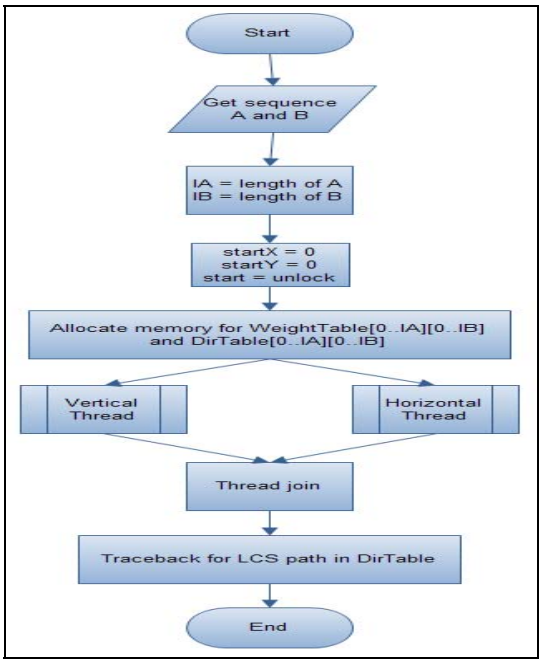
Many common methods for dealing with the LCS problem can be expressed in form of standard algorithms for pattern matching, text, and string searching, as well as those for sequence comparison in molecular biology [10].

2.1 Traditional LCS Problem

In the traditional LCS alignment of a pair of biological sequences, the symbols which represent either amino acids or bases may be shifted in either direction to align as many identical letters as possible. Gaps, or blank symbols that are often denoted with the symbol '-', often need to be inserted into the sequences to obtain improved alignment [2]. A suitable scoring function is used to measure the degree of matching between each pair of symbols. An optimal alignment is one with the highest cumulative score. Note that it is possible for multiple alignments to have the same optimal

score, but this becomes increasingly unlikely for longer and longer sequences. For biological sequences, it is possible to have lengths of over 200 million.

The flowchart and pseudo code in Figure 1 show how traditional LCS fills similarity scores in the edit table during the dynamic programming process. Note that vertical and horizontal cells are filled simultaneously in a sequential manner.




```
PLCS(string A, string B)
lA = length(A)
lB = length(B)
startX = 0
startY = 0
start = unlock
alocate memory for WeightTable[0..lA][0..lB]
alocate memory for DirTable[0..lA][0..lB]
start thread(Vertical)
start thread(Horizontal)
waiting for thread Vertical and Horizontal done
make traceback for LCS path in DirTable
end
```

Figure 2. Parallel LCS flowchart and pseudo code.

2.2 PLCS Algorithm

Important factors when creating a serial algorithm are the accuracy of the program results and the resources (time and memory) required for the program to run [11]. In the case of parallel computing, the accuracy of the alignment results has to be taken even more into consideration due to the data dependency between multi-concurrent threads [8]. For example, it may be possible that one processor would retrieve the values for further calculation before the result is actually ready, or before the job of the other processor is completed. Furthermore, deadlocks could occur when the first thread waits for the second thread while the second thread also waits for the first one, thus causing the program to run in an infinite loop.

Another challenge in parallel programming is to balance the work load between CPUs in order to gain the optimal CPU allocation and an improved run-time. That is, ideally, the CPU would not need to wait for the results from other processes before carrying out the next operation.

Taking the above into consideration, a parallel LCS algorithm was developed. This is described and shown in Figure 2 in form of a flowchart and the corresponding pseudo code.

Referring to Figure 2, sequences A and B have length *lA* and *lB*, respectively. The starting points to fill the edit table are *startX* and *startY*, both with an initial value of zero. The variable *start* is for confirmation of the readiness of *startX* and *startY*. The status '*lock*' means the thread is in critical session. The initial status of *start* is '*unlock*'. Memory on RAM is allocated for WeightTable (for assigning the scores in each cell) and DirTable (for determining the direction of table filling). The vertical and horizontal threads are created using different filling patterns. When the processes in both threads are completed, every cell in the table is filled. A trace back operation is then performed to obtain the optimal path for LCS.

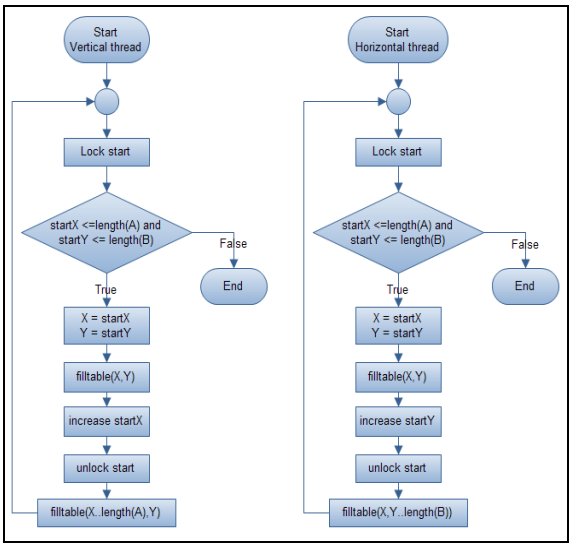


Figure 3. Flowchart of the score filling procedure in parallel LCS.

The flowchart in Figure 3 and the pseudo code in Figure 4 show how the table is filled by threads in the horizontal and vertical directions. Thread Vertical fills the similarity scores for the cells along the vertical direction; whereas Thread Horizontal fills in the cells along the horizontal direction. Both threads would operate until the start point is outside the table reference position. Then they begin filling by changing the variable *start* from the *unlock* state to *lock* state. Next, the values *startX* and *startY* are used as reference variables for filling the table, followed by a shift in the working position. In particular, Thread Vertical shifts to X+1 position while Y remains unchanged. Also, Thread Horizontal shifts to Y+1 while X remains unchanged. Finally, variable *start* changes to *unlock*, and both threads would start filling the scores in their own directions, until the stopping criteria have been reached.

```
thread Vertical
 while X <= length(A) and Y <= length(B)
  lock start
  X = startX
  Y = startY
  filltable(X,Y)
  increase X
  unlock start
  while(Y<=length(B))
    filltable(X,Y)
    increase Y
  end
 end
end
```

```
thread Horizontal
 while X <= length(A) and Y <= length(B)
  lock start
  X = startX
  Y = startY
  filltable(X,Y)
  increase Y
  unlock start
  while(X<=length(A))
    filltable(X,Y)
    increase X
  end
 end
end
```

Figure 4. Pseudo code to fill the vertical and horizontal threads in the proposed parallel LCS.

Figures 5 and 6 illustrate the topology of the order of data filling for a sample pairwise DNA sequence alignment using LCS and PLCS, respectively. In both figures, the arrows show the possible direction of table filling. The vertical and horizontal threads for PLCS are represented by the vertical and horizontal rectangles in Figure 6. In this case, the arrows denote the order of simultaneous data filling of the edit table from the top left to the bottom right by the two parallel threads.

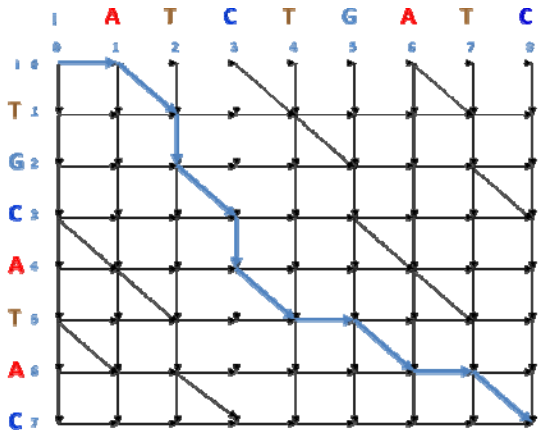


Figure 5. Topology of the data filling procedure for the traditional LCS method.

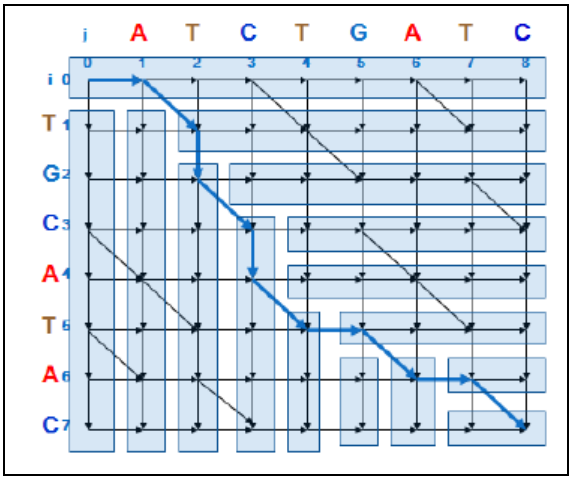


Figure 6. Topology of the data filling procedure for the parallel LCS method.

3. Methodology

To evaluate the performance of the developed parallel LCS algorithm in comparison to traditional LCS algorithm, it is necessary to be able to control the factors which could affect the running time of the processes. Both algorithms were implemented in C and were compiled using GNU Complier Collection (GCC). For the parallel version, the *pthread* library was needed. The source codes for these algorithms are provided in the Appendix.

The random sequences of A, T, C, and G nucleotides were generated for testing inputs of each program. Each program was then executed on the same computer, equipped with 2 Itanium CPUs

which have a clock rate of 1.33GHz each. The run-time of each algorithm was measured for 100 times.

4. Experimental Results and Discussion

The means, standard deviations, minimal values and maximal values of the calculation time for LCS and PLCS are shown in Table 1. The percentages of improvement are calculated from the median values, which are also shown in Table 1. The reduced running time is clearly shown in Figures 7 and 8 for 1,000 bases sequence alignment and 10,000 bases sequence alignment, respectively. The boxplot shown on the left of the figures represent the running time for the serial LCS while the right boxplot represents that of the parallel LCS. The boxes are very thin so they appear like grey horizontal lines. The whiskers represent standard deviations and the black dots are outliers.

Table 1. Comparison of the running time of traditional and parallel LCS algorithms

	1,000 bases length		10,000 bases length	
	Serial	Parallel	Serial	Parallel
mean	0.0871	0.0691	7.505	5.208
SD	0.00743	0.01275	0.04139	0.09739
min	0.084	0.064	7.434	5.007
max	0.138	0.161	7.717	5.51
Q1	0.085	0.065	7.480	5.137
median	0.085	0.065	7.499	5.199
Q3	0.086	0.066	7.524	5.262

It was found that the accuracy of parallel LCS and traditional LCS is the same. That is, the alignments from both algorithms are the same. However, the calculation time was improved by about 23% and 30% when tested with randomly generated 1,000-bases sequence alignment and 10,000-bases sequence alignment, respectively.

The standard deviations varied because of the random workload of the cluster computer used for running the programs. As the cluster is a public cluster and can be accessed by other remote users simultaneously, the system could be running other jobs while performing the LCS and PLCS tasks, resulting in CPU sharing and different run-time results. The standard deviations of LCS performance were higher, as well as the spread of the outlier range. This was expected since in the case of parallel computing, all CPUs were used. CPU sharing, thus, affected the performance of PLCS more significantly. However, it could be seen that PLCS still performed faster regardless of the random incremental workloads by other users of the cluster.

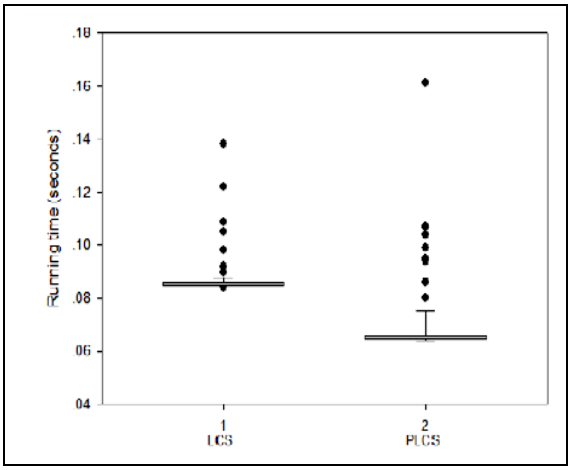


Figure 7. Boxplot comparing run time of serial LCS and parallel LCS for 1,000-bases sequence alignment.

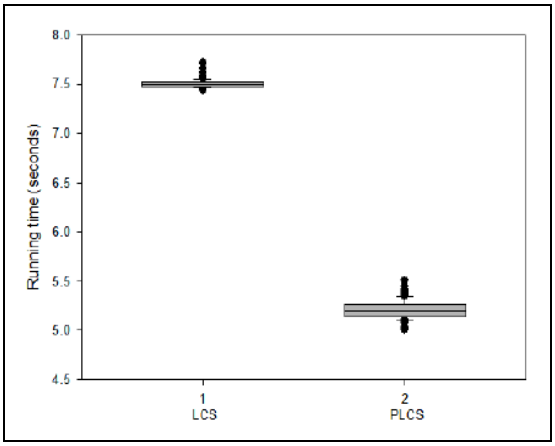


Figure 8. Boxplot comparing run time of serial LCS and parallel LCS for 10,000-bases sequence alignment.

The performance of PLCS was not much improved in 1,000-bases sequence alignment than in 10,000-bases alignment. The reason being that the time taken for the sequences in both alignment jobs to be uploaded into the cluster was not different, providing a constant initial running time for both LCS and PLCS. As can be seen from the results, PLCS showed greater improved performance when

the running time was sufficiently long. That is, pairwise alignment of longer sequences showed improved performance when comparing PLCS to traditional LCS.

5. Conclusions and Future Work

This work has supported the findings that parallel LCS could perform faster than LCS while yielding the same optimal solutions. There have been increasing studies on comparative genomics [12] which require longer running time for alignment due to the use of whole genome sequences. Such analyses, which require multiple genome comparison, have a wider range of applications [4]. Thus, there should be further development on parallel multiple biological sequence alignments to realize the increased power of multi-core computing. Moreover, in order to further test the proposed algorithm, comparison runs will be made using sequences from each chromosome of the human genome with length varying from 47 to 247 Mbps. The new HPC OCEAN cluster at NECTEC will be used for this future work.

Acknowledgments

The authors would like to thank Itanium cluster, Large Scale Simulation Research Laboratory, NECTEC for the use of their cluster for this work. Thanks are also due to the CBS Cluster, Center for Biological Sequence Analysis, Technical University of Denmark (DTU), for their support in part of this work.

References

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. & Dev.*, vol. 49, no. 4/5, pp. 589-604, 2005.

[2] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter and I. Parsons, "FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment," *Algorithmica*, vol. 45, pp. 337-335, 2006.

[3] K.-M. Chao and L. X. Zhang, *Sequence Comparison: Theory and Method*. Springer-Verlag, 2009.

[4] C. M. Fraser, J. Eisen, R. D. Fleischmann, K. A. Ketchum, and S. Peterson, "Comparative genomics and understanding of microbial biology," *Emerging Infectious Diseases*, vol. 6, no. 5, pp. 505–512, 2000.

[5] S. J. Shyu and C. Y. Tsai, "Finding the longest common subsequence for multiple biological sequences by ant colony optimization," *Computers and Operations Research*, vol. 36, pp. 73–91, 2009.

[6] 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*, vol. 48, no. 3, pp. 443-453, 1970.

[7] S. R. .Eddy, "What is a hidden Markov model?" *Nat Biotech*, vol. 22, no. 10, pp. 1315-1316, 2004.

[8] A. Buttari, J. Langou, J. Kurzak and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35. pp. 38-53, January 2009.

[9] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11-13, 2005.

[10] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," *Proc. 7th International Symposium on String Processing Information Retrieval (SPIRE'00)*, Spain, 2000, pp. 39-48.

[11] M. A. Weiss, *Data Structures and algorithm analysis in C*, 2nd Ed. Addison-Wesley, 1997.

[12] T. T. Binnewies, Y. Motro, P. F. Hallin, O. Lund, D. Dunn, T. La, D. J. Hampson, M. Bellgard, T. M. Wassenaar and D. W. Ussery, "Ten years of bacterial genome sequencing: comparative-genomics based discoveries," *Functional and Integrative Genomics*, vol. 6, pp. 165-185, 2006.

Appendix

The following are the source codes for the two algorithms demonstrated in this work:

lcs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void strre(char *str)
{
        char *s;
        char ch;
        s=str;
        while(*s!=0)s++;
        s--;
        while(s>str)
        {
                ch = *s;
                *s = *str;
                *str = ch;
```

```
		str++;
		s--;
	}
}

long lcs(char *str1,char *str2)
{
	long n1,n2;
	long **weight;
	int **dir;
//direction(s1)	0=match/mismatch	1=delete(i)	2=insert(j)
	long i,j,w1,w2,w3;
	char *s1,*s2,*out1,*out2;
	n1 = strlen(str1);
	n2 = strlen(str2);
	s1 = (char*)malloc(sizeof(char)*(n1+2));
	s2 = (char*)malloc(sizeof(char)*(n2+2));
	out1 = (char*)malloc(sizeof(char)*(n1+n2+2));
	out2 = (char*)malloc(sizeof(char)*(n1+n2+2));
	sprintf(s1," %s",str1);
	sprintf(s2," %s",str2);
	n1++;
	n2++;
	weight = (long**)malloc(sizeof(long*)*n1);
	dir = (int**)malloc(sizeof(int*)*n1);
	for(i=0;i<n1;i++)
	{
		weight[i] = (long*)malloc(sizeof(long)*n2);
		dir[i] = (int*)malloc(sizeof(int)*n2);
	}
//Start lcs
	for(i=0;i<n1;i++)	{weight[i][0] = 0-i; dir[i][0]=1;}
	for(i=1;i<n2;i++)	{weight[0][i] = 0-i; dir[0][i]=2;}

	for(i=1;i<n1;i++)
	{
		for(j=1;j<n2;j++)
		{
			if(s1[i]==s2[j])
			{
				w1 = weight[i-1][j-1] + 1;
			}
			else
			{
				w1 = weight[i-1][j-1] - 1;
			}
			w2 = weight[i-1][j]-1;
			w3 = weight[i][j-1]-1;
			if(w1>=w2 && w1>=w3) {weight[i][j] = w1; dir[i][j]=0;}
			else if(w2>=w1 && w2>=w3) {weight[i][j] =w2; dir[i][j]=1;}
			else {weight[i][j] = w3; dir[i][j]=2;}
		}
	}
//	for(i=0;i<n1;i++)
//	{
//		for(j=0;j<n2;j++)
//			printf("%2d ",weight[i][j]);
//		printf("\n");
//	}
	i=n1-1;
	j=n2-1;
	w1=0;
	w2=0;
	while(i>0||j>0)
	{
		if(dir[i][j]==0)
		{
			out1[w1++]=s1[i--];
			out2[w2++]=s2[j--];
		}
		else if(dir[i][j]==1)
		{
			out1[w1++]=s1[i--];
			out2[w2++]='-';
		}
		else
		{
			out1[w1++]='-';
			out2[w2++]=s2[j--];
		}
	}
	out1[w1]=0;
	out2[w2]=0;
//	printf("%s\n%s\n",out1,out2);
	strre(out1);
	strre(out2);
	printf(">%s\n>%s\n",out1,out2);
	return weight[n1-1][n2-1];
}

int main()
{
	char s1[1000000],s2[1000000];
	scanf("%s %s",s1,s2);
//	printf(">%s\n>%s\n",s1,s2);
	printf("weight = %ld\n",lcs(s1,s2));
//	printf("weight = %ld\n",lcs("abc","cabd"));
	return 0;
}
```

plcs.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

pthread_mutex_t loc;
long n1,n2;
long **weight;
int **dir;
long si,sj;
char *s1,*s2;

//direction(s1)        0=match/mismatch   1=delete(i)
2=insert(j)

void strre(char *str)
{
        char *s;
        char ch;
        s=str;
        while(*s!=0)s++;
        s--;
        while(s>str)
        {
                ch = *s;
                *s = *str;
                *str = ch;
                str++;
                s--;
        }
}

void *f1(void * x)
{
        printf("start f1\n");
        long i,j,w1,w2,w3;
        while(1)
        {
                pthread_mutex_lock(&loc);
                if(si==n1||sj==n2)
                {

        pthread_mutex_unlock(&loc);
                        break;
                }
                i=si;
                j=sj;
                if(s1[i]==s2[j])
         {
                        w1 = weight[i-1][j-1] +
1;
                }
         else
         {
               w1 = weight[i-1][j-1] - 1;
         }
         w2 = weight[i-1][j]-1;
         w3 = weight[i][j-1]-1;
         if(w1>=w2 && w1>=w3) {weight[i][j] =
w1; dir[i][j]=0;}
         else      if(w2>=w1      &&      w2>=w3)
{weight[i][j] =w2; dir[i][j]=1;}
         else {weight[i][j] = w3; dir[i][j]=2;}
                si++;
                pthread_mutex_unlock(&loc);
                j++;
                while(j<n2)
                {
                        if(s1[i]==s2[j])
                        {
                                w1 = weight[i-
1][j-1] + 1;
                        }
                        else
                        {
                                w1 = weight[i-1][j-1] -
1;
                        }
                        w2 = weight[i-1][j]-1;
                        w3 = weight[i][j-1]-1;
                        if(w1>=w2      &&      w1>=w3)
{weight[i][j] = w1; dir[i][j]=0;}
                        else  if(w2>=w1  &&  w2>=w3)
{weight[i][j] =w2; dir[i][j]=1;}
                         else     {weight[i][j]     =     w3;
dir[i][j]=2;}
                        j++;
                }
        }
        pthread_exit(NULL);
}
void *f2(void *x)
{
        printf("start f2\n");
        long i,j,w1,w2,w3;
        while(1)
        {
                pthread_mutex_lock(&loc);
                if(si==n1||sj==n2)
                {

        pthread_mutex_unlock(&loc);
                        break;
                }
                i=si;
                j=sj;
                if(s1[i]==s2[j])
         {
                        w1 = weight[i-1][j-1] +
1;
                }
         else
         {
               w1 = weight[i-1][j-1] - 1;
         }
```

```
		w2 = weight[i-1][j]-1;
		w3 = weight[i][j-1]-1;
		if(w1>=w2 && w1>=w3) {weight[i][j] = w1; dir[i][j]=0;}
		else if(w2>=w1 && w2>=w3) {weight[i][j] =w2; dir[i][j]=1;}
		else {weight[i][j] = w3; dir[i][j]=2;}
			sj++;
			pthread_mutex_unlock(&loc);
			i++;
			while(i<n1)
			{
				if(s1[i]==s2[j])
				{
					w1 = weight[i-1][j-1] + 1;
				}
				else
				{
					w1 = weight[i-1][j-1] - 1;
				}
				w2 = weight[i-1][j]-1;
				w3 = weight[i][j-1]-1;
				if(w1>=w2 && w1>=w3) {weight[i][j] = w1; dir[i][j]=0;}
				else if(w2>=w1 && w2>=w3) {weight[i][j] =w2; dir[i][j]=1;}
				 else {weight[i][j] = w3; dir[i][j]=2;}
					i++;
				}
		}
		pthread_exit(NULL);
}

long lcs(char *str1,char *str2)
{
		pthread_t threads[2];
		pthread_attr_t attr;
		pthread_attr_init(&attr);
		pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
		pthread_mutex_init(&loc,NULL);

		long i,j,w1,w2;
		char *out1,*out2;
		n1 = strlen(str1);
		n2 = strlen(str2);
		s1 = (char*)malloc(sizeof(char)*(n1+2));
		s2 = (char*)malloc(sizeof(char)*(n2+2));
		out1 = (char*)malloc(sizeof(char)*(n1+n2+2));
		out2 = (char*)malloc(sizeof(char)*(n1+n2+2));
		sprintf(s1," %s",str1);
		sprintf(s2," %s",str2);
		n1++;
		n2++;
		weight = (long**)malloc(sizeof(long*)*n1);
		dir = (int**)malloc(sizeof(int*)*n1);
		for(i=0;i<n1;i++)
		{
			weight[i] = (long*)malloc(sizeof(long)*n2);
			dir[i] = (int*)malloc(sizeof(int)*n2);
		}
//Start lcs

		for(i=0;i<n1;i++) {weight[i][0] = 0-i; dir[i][0]=1;}
		for(i=1;i<n2;i++) {weight[0][i] = 0-i; dir[0][i]=2;}
		si=1;
		sj=1;
		printf("start thread\n");
		pthread_create(&threads[0],NULL,f1,(void*)si);
		pthread_create(&threads[1],NULL,f2,(void*)si);
		pthread_join(threads[0],NULL);
		pthread_join(threads[1],NULL);

//		for(i=0;i<n1;i++)
//		{
//			for(j=0;j<n2;j++)
//				printf(" %2d ",weight[i][j]);
//			printf("\n");
//		}
		i=n1-1;
		j=n2-1;
		w1=0;
		w2=0;
		while(i>0||j>0)
		{
			if(dir[i][j]==0)
			{
				out1[w1++]=s1[i--];
				out2[w2++]=s2[j--];
			}
			else if(dir[i][j]==1)
			{
				out1[w1++]=s1[i--];
				out2[w2++]='-';
			}
			else
			{
				out1[w1++]='-';
				out2[w2++]=s2[j--];
			}
		}
		out1[w1]=0;
		out2[w2]=0;
```

```
//          printf("%s\n%s\n",out1,out2);
            strre(out1);
            strre(out2);
            printf(">%s\n>%s\n",out1,out2);
            return weight[n1-1][n2-1];
}

int main()
{
            char s1[1000000],s2[1000000];
            scanf("%s %s",s1,s2);
//          printf(">%s\n>%s\n",s1,s2);
            printf("weight = %ld\n",lcs(s1,s2));
//          printf("weight                              =
%ld\n",lcs("abc","cabd"));
            return 0;
}
```