International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 03 Issue: 06 | June-2016 www.irjet.net p-ISSN: 2395-0072

"AN OPTIMIZED PARALLEL ALGORITHM FOR LONGEST COMMON SUBSEQUENCE USING OPENMP"

[1]HanokPalaskar, [2]Prof.TausifDiwan

[1] M.Tech Student, CSE Department, Shri Ramdeobaba College of Engineering and Management, Nagpur, India
[2]Assistant Professor, CSE Department, ShriRamdeobaba College of Engineering and Management, Nagpur,India

---

Abstract - ***Finding the Longest Common Subsequence has many applications, such as in the field of bioinformatics and computational genomics. The LCS problem consist of an optimal substructure and overlapping sub problems, problems which have such properties can be solved using dynamic programming problem solving technique. Due to the enormous growth in the database sizes, it becomes difficult to solve problems like LCS in less amount of time using classical sequential algorithms. To overcome this problem parallel algorithms are the best solution. In this paper we have presented an optimized parallel LCS algorithm, using OpenMP, for Multi-Core architectures. We have optimized our parallel algorithm by load balancing among the threads, we have divided the score matrix into three parts; growing region, stable region and shrinking region depending upon the number of subproblems and have schedule the subproblems effectively to processing cores for optimal utilization of multicore technology. We realize the implementations of our optimized parallel LCS algorithm on Intel Dual-Core and Quad-Core processors using OpenMP with different scheduling policies. Our optimized parallel algorithm achieves approximately 2.5 speedup factor over the conventional sequential algorithm approach on Intel Quad-Core.***

Key Words:*LCS, Dynamic Programming, Parallel Algorithm, OpenMP.*

1. INTRODUCTION

Dynamic programming is widely used for discrete and combinatorial optimization problems. Dynamic programming is based on storing all intermediate results in a tabular form, so as to utilize it for further computations. Due to its amenable computational approach, this technique has been largely adopted for solving various optimization problems, including matrix chain multiplication, longest common subsequence, binary knapsack, travelling salesman problem and so on.

The LCS problem deals with comparing two or more sequence and finding the maximum length subsequence which is common to two or more given sequences. The LCS algorithm is widely used in many areas, which includes the field of gene engineering to compare DNA of patients with that of healthy ones. Also, due to the digitization of information plagiarism has become more convenient, which is evidently displayed in lots of research work. By comparing the similarity between different texts, the detection technique may be realized by LCS algorithm. Apart from this LCS also has application in the areas of speech recognition, file comparison and in the field of bioinformatics.

In the field of bioinformatics most of the common studies have evolved towards a more large scale, for e.g., study and analysis of proteome/ genome instead of a single protein/gene. Therefore, it has become more and more difficult to perform these analyses using sequential algorithms on a single computer. For these kind of massive computations bioinformatics now requires parallel algorithms. Unlike serial algorithm, parallel algorithms can be executed a part at a time on different processing devices and these parts at the end can be combined to get the correct result. Because of the spread of multithreading processors and the multicore machines in the marketplace, it is now possible to create parallel programs for uniprocessors also, and can be utilize to solve the large scale problems like LCS. To perform the parallel processing on multicore machines, lots of shared memory API tools are available; one of such is OpenMP which provides the various constructs which can be added to sequential programs written in C/C++, Fortran. Using various Scheduling constructs of OpenMP we can balance the load among the threads thus allowing us to schedule the sub problems of dynamic programming effectively to processing cores for optimal utilization of multicore processors.

In this paper we have performed parallelization of Positional_LCS algorithm using OpenMP constructs, as the Positional_LCS algorithm focuses only on the matched position it has less execution time as compared to other DP-LCS algorithms. We have also performed the optimization on our parallel LCS algorithm by using different Scheduling constructs on computation regions of the score matrix. Our paper is organized as follows: section 2 deals with the problem definition and the related work. Section 3 describes the background of Positional_LCS algorithm and OpenMP tool. Section 4 explains our proposed parallel algorithm and its optimization. In section 5 and section 6 we have presented our experimental results and conclusion respectively.

2. Related Work

Most of the existing algorithms use DP technique to compute LCS. The main concept of DP is to compute current state from previous state. Let x1 and x2 are two given sequences and L[i,j] is the score matrix computed using recursion equation defined in equation (1). Scanning the score matrix L[i,j] gives us the required LCS.

$$L[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0, \\ L[i-1,j-1] + 1 & \text{if } x1[i] = x2[j], \\ \text{Max}(L[i,j-1], L[i-1,j]) & \text{if } x1[i] \neq x2[j] \end{cases} \quad \text{Eq.(1)}$$

For the example x1=CTGCTCACCG and x2=CTTCTCAAAT the score matrix is computed using Eq.(1). Table 1 shows the score matrix for given two sequences.

Table (1) Score Matrix

	j	0	1	2	3	4	5	6	7	8	9	10
i			C	T	T	C	T	C	A	A	A	T
0		0	0	0	0	0	0	0	0	0	0	0
1	C	0	1	1	1	1	1	1	1	1	1	1
2	T	0	1	2	2	2	2	2	2	2	2	2
3	G	0	1	2	2	2	2	2	2	2	2	2
4	C	0	1	2	2	3	3	3	3	3	3	3
5	T	0	1	2	3	3	4	4	4	4	4	4
6	C	0	1	2	3	4	5	5	5	5	5	5
7	A	0	1	2	3	4	5	5	6	6	6	6
8	C	0	1	2	3	4	5	5	6	6	6	6
9	C	0	1	2	3	4	5	5	6	6	6	6
10	G	0	1	2	3	4	5	5	6	6	6	6

Scanning of the score matrix gives us the required LCS which is CTCTCA in this case. In DP technique main computational process to get the required LCS is the computation of the score matrix and scanning of the entire score matrix. For the two input sequence of size n n2 calculations are required to compute the score matrix and n2 time is required to scan the score matrix to get the required LCS.

In [1] authors proposed parallelization of LCS problem using Dynamic Programming technique. They have presented parallelization approach for solving LCS problem on GPU, and using CUDA and OpenCL they have implemented their proposed algorithm on NVIDIA platform. Computation of score matrix is carried out in anti-diagonal fashion in order to eliminate dependency. Using OpenMP API they have implemented their proposed algorithm on CPU.

Instead of dynamic programming technique in [2] authors have proposed an algorithm based on the Dominant Point approach which make use of the divide-and-conquer technique. They have divided the score matrix into the size of LCS and have used two algorithms Union() and Find() to compute the output. This algorithm is also suitable for the Multiple LCS.

In [3] author proposed a parallel algorithm for LCS which is based on the calculation of the relative positions of the characters. This algorithm recognizes and rejects all those subsequences which fail to generate the next character of LCS. Drawback of this algorithm is that it requires having the knowledge of number of characters being used in the sequence in advance. Parallelization of this algorithm uses the multiple processors where number of processors should be equal to number of characters.

Instead of computing the entire score matrix of n×n, in [4] authors have used the optimized technique of the theorems which calculates the score matrix of order p×n where p is less than n. For the computation of the LCS they have devised a formula which gives the required LCS from the score matrix without backtracking. Parallelization of this algorithm is done by using OpenMP constructs for the Multi-Core CPUs

In [4] authors have proposed a new algorithm for LCS, Postional_LCS. Instead of focusing on both matched and unmatched positions of sequences this algorithm focus only on matched positions and stores those positions in a separate array. To compute the required LCS this algorithms does not use backtracking on the score matrix instead, it access the array which have the positions of

matched characters. In this paper we have presented the optimized parallel version of this Positional_LCS algorithm.

3. BACKGROUND

3.1 Positional LCS

In DP approach of finding LCS, to find the value of current position L[i, j] in score matrix its three adjacent cells namely L[i-1,j], L[i,j-1] and L[i-1,j-1] values are used. Table 2 provides the adjacent positions of the score matrix L[i,j].

Left Top	Right Top
L[i-1,j-1]	L[i-1,j]
L[i,j-1]	L[i,j]
Left Bottom	Current Position

Table (2) Adjacent Positions for Score Matrix

If a matching occurs at ith position of the sequence x1 and jth position of the sequence x2, then value of current position L[i,j] is computed by incrementing the value of its left top position in score matrix by one, i.e. L[i,j] = L[i-1, j-1] + 1.Otherwise, the value of current position L[i,j] is computed as maximum value of its left bottom position L[i,j-1] right top position L[i-1,j]. So value of the current position L[i,j] is max (L[i-1,j] and L[i,j-1]).

LCS always occurs only at matching positions of the sequences. Last occurrence of the matching position is the left top position. Positional_LCS use an array namely Parent[i,j] to store this left top position L[i-1, j-1]. Whenever we get a new matching at the position (i,j), the value of current position L[i,j] is computed by adding one with its left top position value and current LCS position is stored in Parent[i,j].

By maintaining this parent array, scanning the entire score matrix (both matched and unmatched positions), to get the required LCS, is eliminated. Parent matrix gives the required LCS. Parent matrix stores the left top positions, the first time where LCS matching occurred. This algorithm compares the current score value and the maximum score value. If the maximum score value is less than the current score value, then next new LCS matching occurs.

The current score value at this point is stored in the Parent matrix. In other words, the Parent matrix contains the positions of the characters of required LCS. Parent matrix itself produces the required LCS, instead of scanning entire matrix.

3.2 OpenMP

OpenMP is one of the favorite Application Programming Interface used for parallelization on the shared memory architecture, adopted by a majority of high performance community due to its higherprogramming efficiency. OpenMP is shared memory programming fork join model that provides various directives and library functions for creating and managing a team of threads. Various synchronization and work sharing constructs are provided by OpenMP, using which we automatically or manually divide the task among threads. OpenMP provides four different types of scheduling for assigning the loop iterations to different threads: static, dynamic, guided and runtime.Schedule clause is provided for specifying schedule and numbers of iterations i.e. chunk size. In static scheduling, chunks are assigned to processing cores in round robin fashion. It is the simplest kind of scheduling with minimum overhead. In dynamic scheduling, thread requests for new chunk as it finishes the assigned chunk. In the guided scheduling thread request for newer chunks, but chunk size is calculated as the number of unassigned iterations divided by the total number of threads in the team. Guided scheduling seems to be more efficient scheduling, but involves a little bit of overheads in the calculation of chunk size. In runtime scheduling, schedule and optional chunk size are set with the help of environment variables. The details of scheduling techniques are discussed in [8, 9].

4. PROPOSED PARALLEL ALGORITHM

In this section we present the parallel version on Positional_LCS for Multi-Core CPUs using OpenMP. First we discuss the parallelization approach we used to parallelize the algorithm and then we present the optimization which we have applied to our parallel algorithm.

4.1 The Parallelization Approach

When we observe the construction of score matrix using dynamic programming approach we can see the value of current element L[i,j] is depends on the three entries in the score matrix; L[i-1, j-1], L[i-1,j], L[i,j-1]. In other words, L[i,j] depends on data in the same column and same row. That implies, we cannot compute the cells in the same row or column in parallel.

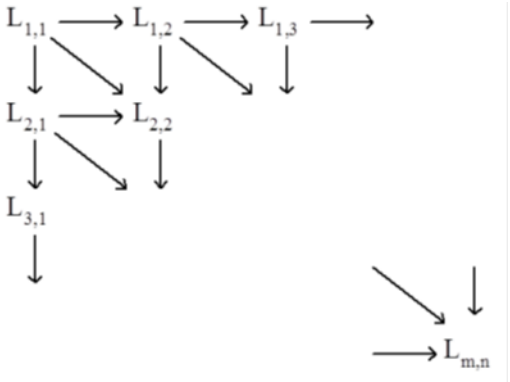

Fig. 1 Data Dependency in Score Matrix

To compute the elements of score matrix in parallel, we start computing from L1,1 then L2,1 and L1,2 at the same time and so on. We can see that computation of elements which are in the same diagonal can be done in parallel. To perform the computation of score matrix in parallel we are computing elements of score matrix in diagonal manner instead of row wise (see Fig. 2).

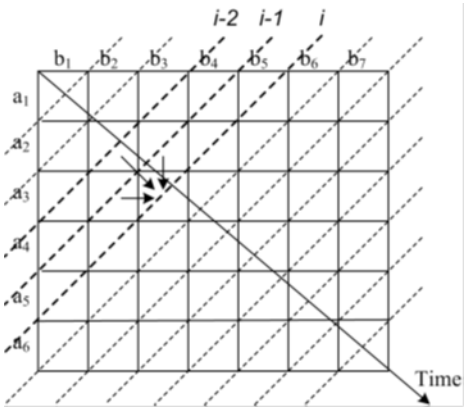

Fig. 2 The Parallelization Approach

4.2 Parallel Algorithm

In this paper we have use OpenMP and C language for the parallelization of the sequential algorithm. For the parallelization we have computed the score matrix diagonally, and for computation of elements in the same diagonal we have applied OpenMP constructs on the inner for loop (see Fig.3). We have also restricted the number of threads equals to the number of cores in the machine to avoid the computation overhead.
Pseudo-code for our proposed parallel algorithm can be given as follows:

ALGORITHM L(A,B)
INPUT STRING A AND STRING B
OUTPUT LCS OF A AND B
Begin
(1) Initialization of matrix elements L(m,n)

(2) Computation of elements of score matrix diagonally in parallel manner and maintaining the parent array to store the matched positions

(3) Printing LCS from parent matrix

End

```
// following code calculates the growing region of score matrix diagonally in parallel manner
 for(i=1;i<=m;i++)
 {

        #pragma omp parallel for private(j,k)

        for(j=1;j<=i;j++)
        {
                k=i-(j-1);
                if(y[k-1]==x[j-1])
                {
                        c[k][j]=c[k-1][j-1]+1;
                        {
                                if(c[k][j]>Lmax)
                                {
                                        Lmax=c[k][j];
                                        parent[s]=k-1;
                                        s++;

                                        count++;

                                }


                        }
                }


                else if(c[k-1][j]>=c[k][j-1])
                {
                        c[k][j]=c[k-1][j];
                }
                else
                {
                        c[k][j]=c[k][j-1];
                }
```

Fig. 3 Parallel Region in Code

4.3 Optimization

Due to non-uniformity in the inherent dependence in dynamic programming algorithms, it becomes necessary to schedule the sub problems of dynamic programming effectively to processing cores for optimal utilization of multicore technology. For the optimization of our parallel algorithm we have used the load balancing. We have divided the score matrix of LCS in three parts; growing region, stable region and shrinking region depending on whether the number of sub problems increases, remain stable or decreases uniformly phase by phase respectively. Fig. 4 represents the region wise partition and

arrows indicate direction of parallelization strategies for the LCS.

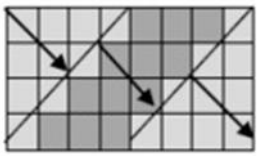

Fig. 4 Parallelization Strategy and Region Wise Distribution of Score Matrix

For each phase, the numbers of subproblems are assigned to the threads which are handled by the chunk size parameter in OpenMP and finally threads execute those assigned subproblems over physical cores which are handled by a scheduling policy in OpenMP. We have performed the experiments using three different scheduling policies on each phase and based on experimental results we have chosen best policy for each phase in order to get the optimized results.

5. RESULT AND COMPARISON

5.1 Experimental Results of Parallel Algorithm

We have evaluated the performance of our Optimized Parallel LCS on Intel Dual Core processor with CPU clock 2.00 GHz, 2 CPU cores, 4 GB of RAM and Intel Quad Core processor with CPU clock 2.3GHz, 4 CPU cores, 4 GB of memory. The operating system used for performance evaluation is Obuntu 14.1 32-bit Linux with GNU GCC compiler 4.8.3 with OpenMP.

Speed up is computed by taking ratio of time taken by serial algorithm to time taken by parallel algorithm. For the Dual-Core processor 1.42 speedup is computed, while on Quad Core processor 2.15 speed up is computed. As all speedups are greater than one, OpenMP performs better as compared to the sequential algorithm. Table 3 shows the experimental results of serial algorithm and parallel algorithm and Fig. 5 represents the graphical representation of comparison of serial and parallel algorithms for different input size.

Time (ms)	Number of Characters			
	N=500	N=1000	N=6000	N=10000
Quad-Core	15.52	20.48	383.7	835.5
Dual-Core	14.86	31.29	682.7	1476.0
Serial	12.37	30.92	935.7	2259.6

Table 3 Execution Time Comparison

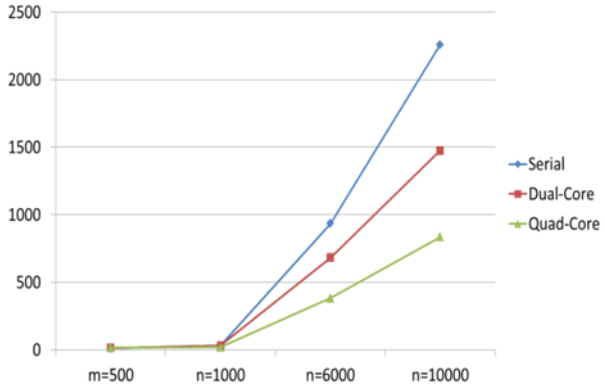

Fig 5

5.2 Experiment Results of Optimization

In this section we present the experiment results of our parallel algorithm after the optimization. We have divided the score matrix computation into growing phase, stable phase and shrinking phase and have applied the Static, Dynamic and Guided constructs of OpenMP to get the optimized result. Table 4 and 5 shows the experimental results of different scheduling schemes on Dual-Core and Quad-Core processors respectively and Fig. 6 shows the graphical representation of comparison of different scheduling policies. Time is in seconds.

Input Size	Scheduling Type		
	Static	Dynamic	Guided
6000	0.691	1.370	0.682
8000	1.123	2.076	0.994
10000	1.473	3.104	1.445

Table 4 Scheduling on Dual-Core

Input Size	Scheduling Type		
	Static	Dynamic	Guided
6000	0.386	0.788	0.404
8000	0.573	1.180	0.593
10000	0.843	1.758	0.860

Table 5 Scheduling on Quad-Core

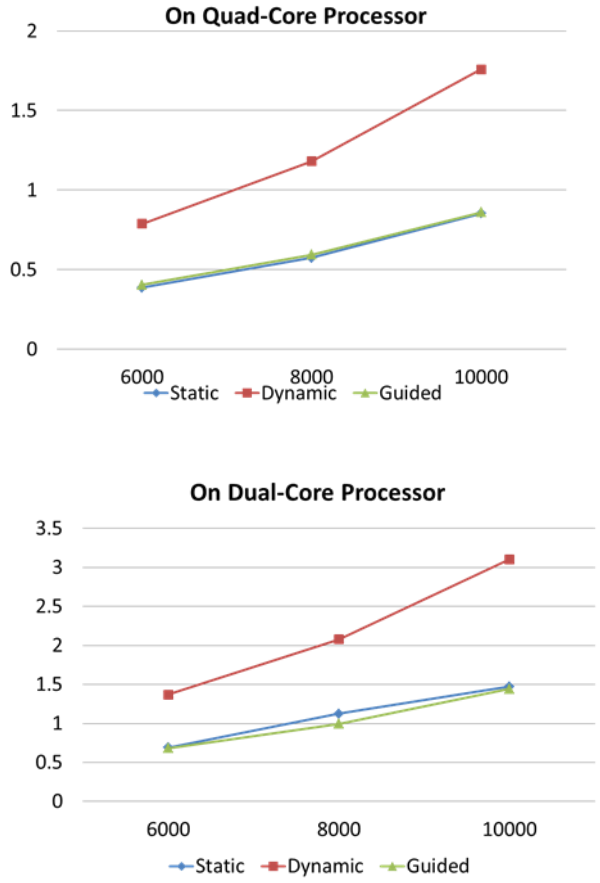

Fig. 6 Scheduling on Quad-Core and Dual-Core

5.3 Result Analysis

From the experimental results we have observed that when the input size is greater than 1000 parallel algorithm takes less time than the serial one. But when input size is small then the sequential algorithm is faster than the parallel algorithm. For this we have set one threshold value in our algorithm if the input size is less than threshold value our algorithm will execute serially and if input size is greater than threshold algorithm will execute parallel. Guided Scheduling is the best for the Dual-Core processor when applied on the shrinking region and Static Scheduling on the growing region. Whereas, Static Scheduling is best for the Quad-Core processor when applied on the shrinking region as well as on growing region.

6. CONCLUSION AND FUTURE WORK

Problem of LCS have the variety of applications in the domain of bioinformatics, pattern recognition and data mining. Due to the recent developments in the multi-core CPUs, for the problems having large size input parallel algorithms using OpenMP are one of the best ways to solve these problems. In this paper we have presented an Optimized Parallel algorithm using OpenMP for Multi-Core CPUs.We have observed from our experimental results that our optimized parallel algorithm is faster than the sequential LCS algorithm for the large input size. We also conclude that Guided scheduling is best for the Dual-CoreProcessors while Static scheduling is best for Quad-Core processors. In future we can extend this algorithm to support Multiple Longest Common Subsequence; also we can implement this algorithm on Graphical Processing Unit (GPU) using CUDA or OpenCL and study the performance. This algorithm can also be implemented on distributed memory architecture using hybrid of OpenMP and MPI.

7. REFERENCES

[1] Amine Dhraief, RaikIssaoui, AbdelfettahBelghith, "Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability", The First International Conference on Advanced Communications and Computation, 2011.

[2] Quingguo Wang, Dmitry Korkin, Yi Shang, "A Fast Multiple Longest Common Subsequnce (MLCS)Algorithm",IEEE transaction on knowledge and data engineering, 2011.

[3] AmitShukla, SuneetaAgrawal, "A Relative Position based Algorithm to find out the Longest Common Subsequence from Multiple Biological Sequnces ", 2010 International Conference on Computer and Communication Technology, pages 496 – 502.

[4]R. Devika Ruby, Dr. L. Arockiam, "Positional LCS: A position based algorithm to find Longest Common Subsequence (LCS) in Sequence Database (SDB)", IEEE International Conference on Computational Intelligence and Computing Research, 2012.

[5] Jiamei Liu, Suping Wu "Research on Longest Common Subsequence Fast Algorithm", 2011 International Conference on Consumer Electronics, Communications and Networks, pages 4338 – 4341.

[6] ZhongZheng, Xuhao Chen, Zhiying Wang, Li Shen, Jaiwen Li "Performance Model for OpenMP Parallelized Loops ", 2011 International Conference on Transportation, Mechanical and Electrical Engineering (TMEE), pages 383-387.

[7] Rahim Khan, Mushtaq Ahmad, Muhammad Zakarya, "Longest Common Subsequence Based Algorithm for Measuring Similarity Between Time Series: A New Approach" World Applied Sciences Journal, pages 1192-1198.

[8] Jiaoyun Yang, Yun Xu,"A Space-Bounded Anytime Algorithm for the Multiple Longest Common Subsequence Problem", IEEE transaction on knowledge and data engineering, 2014.

[9] I-Hsuan Yang, Chien-Pin Huang, Kun-Mao Chao, "A fast algorithm for computing a longest common increasing subsequence", Information Processing Letters, ELSEVIER, 2004.

[10] Yu Haiying, Zhao Junlan, Application of Longest Common Subsequence Algorithm in Similarity Measurement of Program Source Codes. Journal of Inner Mongolia University, vol. 39, pp. 225–229, Mar 2008.

[11] KrsteAsanovic, RasBodik, Bryan, Joseph, Parry, Samuel Williams, "The Landscape of Parallel Computing Research: A view from Berkeley" Electrical Engineering and Computer Sciences University of California at Berkeley, December 2006.

[12] Barbara Champman, Gabriel Jost, Ruud Van Der Pas "Using OpenMP", 1-123