Longest Common Subsequence - OpenMP Implementation Group 10

André Filipe da Silva Bispo
ist166941
andrebispo@ist.utl.pt

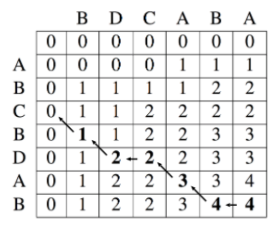
Ricardo Moreira Silva Neves
ist167072
ricardo.neves@ist.utl.pt

Diogo Filipe Reis Pinto
ist169905
diogo.reis.pinto@ist.utl.pt

1. INTRODUCTION

The purpose of this project is to gain experience in parallel programming using OpenMP, achieving an efficient parallelization of a serial version of longest common subsequence problem solution.

It is expected for us to produce a serial implementation, as efficient as possible, and then parallelize it to achieve a good speedup. For solving this problem, we will use the dynamic programming approach to fill up the score matrix, achieved by the algorithm presented in figure 1.



		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	**1**	1	2	2	3	3
D	0	1	**2**	← **2**	2	3	3
A	0	1	2	2	**3**	3	4
B	0	1	2	2	3	**4**	← **4**

Figure 1: Dynamic programming algorithm.

This computes the optimum LCS solution by considering one extra element of the input sequence at a time. If there is a match, we increase the count. If there is a mismatch, we consider the best subsequence found so far. The length of the LCS is stored in position [m, n]. For obtaining the LCS solution, we just need to use a backtracking method.

A serial implementation is easy to achieve using this algorithm, but parallelize it would be extremely difficult due to data dependencies. Using this algorithm as-is, when filling up the matrix in position [i,j], we would always depend of positions [i-1,j], [i,j-1] e [i-1,j-1]. For this reason we had to look for approaches to remove this dependencies and be able to parallelize the algorithm with a good load-balancing.

2. PARALLELIZATION APPROACHES

To solve this problem, we take in account two approaches: one known as anti-diagonals approach[1] and another that makes use of an auxiliary matrix to remove some data dependencies[2].

2.1 Anti-diagonals approach

When filling up the score matrix we were faced with the data dependencies presented in figure 2. To avoid this, we can fill the score matrix by diagonals, being able to parallelize the processing in each diagonal, as presented in fig 3. This solution removes the problem of data dependencies but is not efficient regarding load balancing.

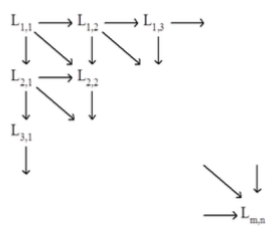


Figure 2: Data dependencies in score matrix

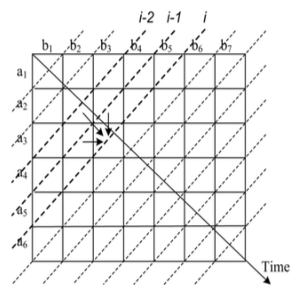

i-2
i-1
i
b_1 b_2 b_3 b_4 b_5 b_6 b_7
a_1 a_2 a_3 a_4 a_5 a_6
Time

Figure 3: Anti-diagonals approach.

To be able to adjust the efficiency of this approach, we would need to increase the number of cells processed in each iteration by having bigger diagonals, but this is not possible without stepping in some data races.

2.2 Auxiliary Table approach

To face the unbalancing problem described in the anti-diagonal approach we need a way to rearrange the data dependencies in order to enable the parallelization by columns or rows, making all processors work under the same load.

For solving this problem, we will use an auxiliary matrix composed by the working alphabet, l, and one of the strings, m. This matrix, $P_{[m][l]}$, will show for each l what are the indexes where it occurs in string m. After the computation of this matrix, we will use it in addiction with two unsigned bits operations to expedite the process of string comparison. To better explain this process the pseudo-code is presented in figure 4.

3. EXPERIMENTAL RESULTS

After the implementation of the algorithm using the auxiliary matrix approach, we will present the experimental results we obtained. All the tests were performed 5 times allowing us to have accurate results by using an average. We used an Intel Core i5 760 (Quad-core) processor, equipped with 8GB of RAM and running Debian 7.0 operating system.

	Matrix Size (m*n)										
	600	2.4k	3.75k	30k	4500k	24000k	72000k	306000k	360000k	1440000k	1800000k
Sequential Time (s)	0,0001	0,0008	0,0012	0,0107	1,7792	8,0857	24,1945	102,4512	120,6533	482,6970	606,9738
Parallel Time (s)	0,0003	0,0005	0,0007	0,0038	0,4929	2,2944	6,8266	28,3978	33,1763	131,3451	165,1186
Speedup	0,7532	1,5548	1,7830	2,7892	3,6078	3,5274	3,5442	3,6078	3,6369	3,6751	3,6760

Table 1: Execution times and speedups with 4 threads.

```
1. For $i = 1$ to $l$ par-do
     For $j = 1$ to $m$
       Calculate $P[i, j]$ according to Eq. (6)
     End for
   end for

2. For $i = 1$ to $n$
     For $j = 1$ to $m$ par-do
       $t =$ the sign bit of $(0 - P[c, j])$.
       $s =$ the sign bit of $(0 - (S[i-1, j] - t * S[i - 1, P[c, j] - 1]))$.
       $S[i, j] = S[i-1, j] + t * (s \oplus 1)$.
     End for
   End for
```

Figure 4: Pseudo-code for Auxiliary Matrix approach.

3.1 Execution Time Performance

In order to evaluate the execution performance we ran 11 tests, with different matrix sizes. As shown on figure 5, we can reduce almost 4 times the sequential execution time by using all the processing power available on the quad-core machine. With an increasing size of the score matrix this result is more clear.

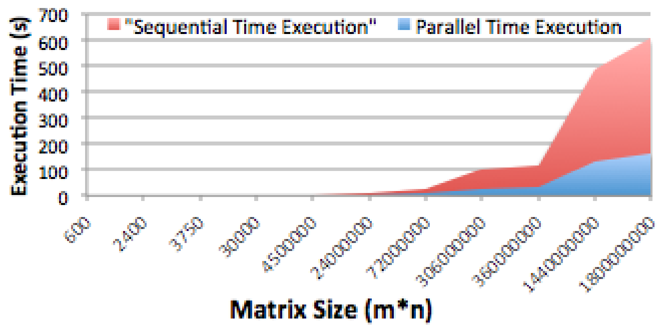


Figure 5: Sequential and Parallel execution times.

3.2 Speedup Comparisons

As shown on table 1, presenting the speedups for the 11 tests performed, we obtained better speedups with an increasing matrix size. For really small matrixes, with less than 2000 cells, the sequential version grants better results, because of the overheads incurred on thread managing. In matrixes bigger than 2000 cells, the speedup averages range about 1.55 and 3.67, becoming more stable above 4500k cells.

3.3 Thread Number Influence

As shown on figure 6, that presents the speedups using 2,4,8 and 16 threads for parallelization, the optimum number of running threads is 4, because we are running on a quad-core machine, taking advantage of all the processors. The 2 threads test is only using half of the total processing power, while the 8 and 16 threads test incur on additional overheads scheduling the exceeding threads.

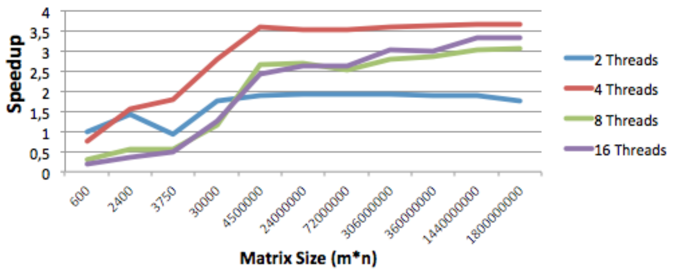


Figure 6: Speedup comparison varying number of threads.

4. CONCLUSIONS

After evaluating all the results achieved, we can conclude that the auxiliary matrix approach grants better speedups compared to the anti-diagonal one. This is due to the fact that load is more balanced by parallelizing columns or lines instead of variable size diagonals. we can also conclude that the optimum number of threads running in parallel is equal to the machine's number of cores, because additional threads will incur on overheads scheduling them. With the implementation of this algorithm, we achieved the maximum speedup of 3.67. Taking in account that, in our case, the optimal speedup is 4.0, this is a very good result due to the overhead created by the parallelization.

5. REFERENCES

[1] R. I. Amine Dhraief and A. Belghith. Parallel computing the longest common subsequence (lcs) on gpus: Efficiency and language suitability. *The International Conference on Advanced Communications and Computation (INFOCOMP 2011)*, October 2011.

[2] Y. X. Jiaoyun Yang and Y. Shang. An efficient parallel algorithm for longest common subsequence problem on gpus. *Proceedings of the World Congress on Engineering*, July 2010.