Longest Common Subsequence - OpenMP and MPI 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 and MPI, 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.

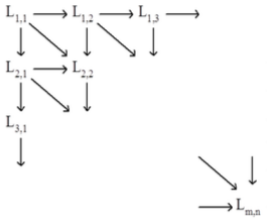

Figure 2: Data dependencies in score matrix

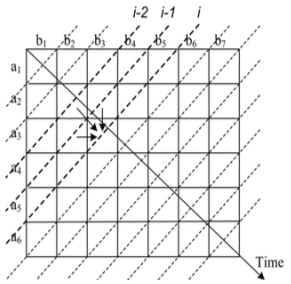

Figure 3: Anti-diagonals approach.

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. OPENMP IMPLEMENTATION

For implementing the openMP parallel version of our project we used the pseudo-code presented in figure 4 as-is, parallelizing the inner cycle (corresponding to processing an entire line of the matrix), by dividing the line into the same number of divisions as the number of cores we have available on the machine

Using the algorithm presented in figure 4, it was possible

	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.

to implement this version without any trouble, in the sense we only depend the last line of the matrix to compute the current line, making us able to fully parallelize each line.

3.1 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.

3.1.1 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 clearer.

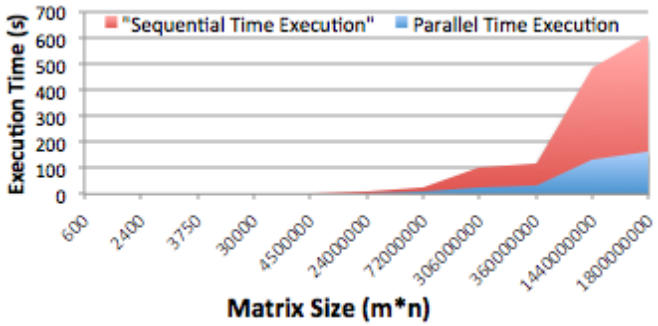

Figure 5: Sequential and Parallel execution times.

3.1.2 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.

3.1.3 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.

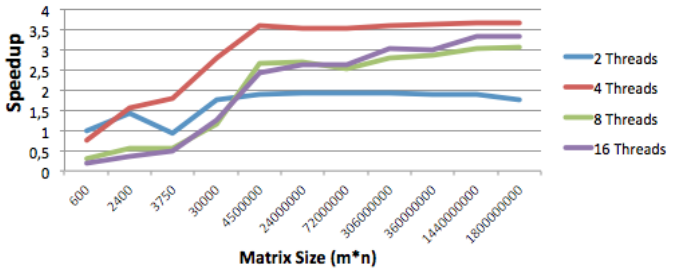

Figure 6: Speedup comparison varying number of threads.

4. MPI IMPLEMENTATION

For implementing the MPI version, we had a new constraint. The score matrix can be bigger than the available memory of one machine. To be able to respect this constraint, we had to adapt the algorithm presented in section 2.2 to be able to achieve an implementation where each one of the machines would have only one subset of the whole matrix. Furthermore, we also need to adapt the backtracking process, being a distributed process in the new implementation. For achieving this, we came to several approaches, which we will explain next.

4.1 First approach

The first approach was made considering minor changes of the openMP implementation. We would divide the total number of lines to process by the machines available and

each one of the machines would store that subset of the matrix.

For a set of lines, a machine is assigned to store them. For each line of the matrix, we would divide the processing for the machines available. The machine storing the line being processed would gather all the distributed computation of other machines.

This first approach was too bad in terms of performance because we were using too much communication for processing one line. For each line, the machine holding its previous line already computed had to broadcast it to all other machines. Furthermore, each one of the machines, after computing its part of the line, had to send it to the machine that would store it.

4.2 Second approach

The second approach was made for reducing the communication costs between machines. For this, we make the matrix division by columns, so each machine would store a columns' subset of the whole matrix. Each machine would compute the first line of its columns, and send it to the machine holding the adjacent columns, that will store it as the previous line computed and start its processing.

This approach reduces significantly the number and size of communications, in the way there is no need to send processed cells to machines holding the line because each machine computes the cells it will store. Also, each machine will only communicate to the machine that has adjacent columns to it, reducing the number of communications.

With this approach we were able to improve our speedup results, but was not the result we were expecting, in the sense we were spending too much computational time waiting for the machine that holds the adjacent lines. Also, we were making too much communications.

4.3 Third approach

The third approach, and the adopted one, was considering how we could reduce the time each machine is waiting for the previous line and also for reducing communication overheads. For this, we return to the division of the whole matrix by lines, assigning a subset of the lines to each machine.

Considering that each machine holds a set of lines, it just needs the last line computed by the previous machine. For being able to achieve a higher scale of parallelization, we divide the processing of each group of lines in blocks.

Each block will be a subset of the total columns each machine holds. This division by blocks will allow to decrease the time each machine waits to start processing each block. Furthermore, this approach will greatly reduce the communication comparing to our previous solution. This approach is detailed graphically in figure 7

Another important aspect of this approach was to define the optimal value for the processing division into blocks. We conduct a series of experiments for being able to define the optimal value. This value needs to be dynamic, taking in account the matrix size and the number of machines we have

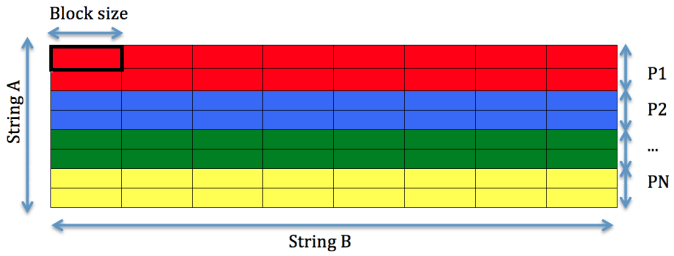

Figure 7: Matrix Division for the third approach

available. We obtain, for the optimal formula to compute the number of blocks:

$$NumBlocks = 0,0025 * lenB * p$$

Taking in account this formula, we are considering a block with 0,25% of the total number of columns, also weighted with the number of machines available.

4.4 Experimental Results

we will present next the experimental results we obtained. All the tests were performed with machines on lab13 and lab14 at RNL, all equipped with Intel Core i5 760 (Quadcore) processors, with 8GB of RAM and running Debian 7.0 operating system. We also need to take in account that the machines were not completely available, making our tests not really accurate. This type of tests should be ran in a controlled and closed environment where all the processing power of each machine is available.

4.4.1 Speedups and efficiency comparison

In table 2 we find the speedups obtained for our implementation, for different number of machines and different tests sizes. The results are good for tests that require a bigger score matrix. We obtained the best results for the 18k.17k and 48k.30k tests, where for 2, 4 and 8 machines we obtained speedups close to the ideal. For more machines we have a deterioration of results, caused by bigger overheads in communication among machines (more machines implies more communication). For small tests, like 150.200, the results are, as we expected, not good in the sense it is not profitable to parallelize such small computation.

We also present, in table 2 the efficiency calculated by the formula:

$$Efficiency = \frac{Obtainedspeedup}{OptimalSpeedup} * 100$$

The efficiency variation follows the same behavioral pattern as speedups. The speedup results are also represented graphically in figure 8 where we have the speedup variation by matrix size and number of machines.

		Matrix Size(String A size x String B size)				
		150x200	3Kx8K	18Kx17K	48Kx30K	60Kx30K
2 Machines	Speedup	1,39	1,92	1,91	1,91	1,86
	Efficiency(%)	69,7	96,1	95,4	95,7	93,2
4 Machines	Speedup	1,57	3,64	3,77	3,86	3,62
	Efficiency(%)	39,3	91,1	94,2	96,5	90,5
8 Machines	Speedup	1,46	4,84	7,39	7,43	7,29
	Efficiency(%)	18,3	60,5	92,4	92,9	91,1
16 Machines	Speedup	1,08	12,29	13,89	14,20	14,10
	Efficiency(%)	6,8	76,8	86,8	88,8	88,2

Table 2: Speedups and efficiency obtained for 2, 4, 8 and 16 machines.

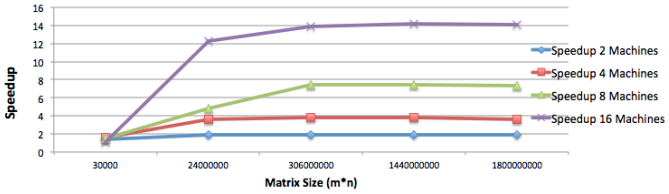

Figure 8: Speedups obtained for 2, 4, 8 and 16 machines.

5. MPI WITH OPENMP IMPLEMENTATION

When talking about parallelization, we want to take the more computation power as we can get from our resources. Using only openMP we are maximizing the CPU power of one machine, using all the processors available. Using MPI we are maximizing the power we can get from a set of computers in a cluster, but only using one processor of each.

Our objective for using openMP and MPI together is to take the full power of the resources available, using all machines and all the processors in each machine.

For achieving this, we implemented an openMP-MPI version from the MPI version presented in section 4, where we used openMP to parallelize the processing in each block. One important aspect to take in account was the number of blocks. Considering we have more processes working under the same machine, we would maximize performance if we consider bigger blocks. The new computed value for the number of blocks is:

$$NumBlocks = 0,0025 * lenB$$

As we can see, comparing to the equation for the number of blocks in an MPI implementation, we are decreasing the number of blocks in function with the number of machines.

5.1 Experimental Results

we will present next the experimental results we obtained for the MPI with openMP implementation. All the tests were once again performed with machines on lab13 and lab14 at RNL, all equipped with Intel Core i5 760 (Quad-core) processors, with 8GB of RAM and running Debian 7.0 operating system.

Taking in account that this machines are being constantly used by the RNL Cluster, to test the influence of openMP with our MPI implementation we considered the number of threads for OMP as two. This will only use two processors of the all four available, giving more reliable results. Once again, the tests should had been ran in a controlled environment.

5.1.1 *Speedups and efficency comparison*

As expected, the MPI with openMP implementation follows the same behavior as we saw on MPI only, being the tests with bigger score matrixes the ones we obtained better speedups and efficiency, as seen in figure 9 and table 3

On a first perception of the problem, we expect to obtain better efficiency for MPI with openMP than with MPI only, considering the same number of processes. For example with 8 machines on an MPI implementation we expect a less efficient result than with 4 machines and 2 threads on an MPI with openMP implementation, because we are reducing communication overheads on 8 machines to 4 machines.

Considering the presented results, we conclude that this does not happen. We can justify this by taking in account all the overheads with threads creation and management on OMP that can overcome the communication overheads of MPI. Also, when we performed the tests for MPI with OMP, the RNL cluster was more overloaded, making our results less accurate.

		Matrix Size(String A size x String B size)				
		150x200	3Kx8K	18Kx17K	48Kx30K	60Kx30K
2 Machines 2 Threads	Speedup	2,03	3,37	3,51	3,50	3,57
	Efficiency(%)	50,6	84,3	87,8	87,6	89,3
4 Machines 2 Threads	Speedup	2,03	5,99	6,60	6,73	6,88
	Efficiency(%)	25,4	74,9	82,5	84,1	86,0
8 Machines 2 Threads	Speedup	1,76	10,27	12,87	12,70	12,92
	Efficiency(%)	11,0	64,2	80,4	79,4	80,7
16 Machines 2 Threads	Speedup	1,07	15,43	20,53	22,69	23,05
	Efficiency(%)	3,3	48,2	64,2	70,9	72,0

Table 3: Speedups and efficency obtained for 2, 4, 8 and 16 machines.

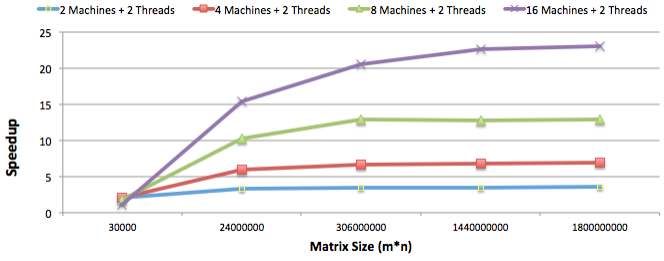

Figure 9: Speedups and efficiency obtained for 2, 4, 8 and 16 machines.

6. CONCLUSIONS

After evaluating all the results and parallelization approaches presented, we can conclude that the auxiliary matrix approach grants better speedups compared to the anti-diagonal one, in openMP. 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 openMP, 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

The division by lines making use of blocks of computation is the better approach when applied to MPI, because the overheads of MPI are associated with the costs of communication between machines. We also applied openMP to our MPI program to take complete advantage of all the computational resources.

As expected, the results' efficiency for MPI implementation are better when applied to bigger problems, overcoming the communication costs with more processing per machine. However, we were expecting better results when adding OMP to MPI program, but the results were not so evident. We can explain it by threads' creation and management overheads incurred in openMP.

7. 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.