A parallel approach to the longest common subsequence problem

Francesco Pham[1,✉]

[1]Student id. 1234004 University of Padua

This report covers my final project for the parallel computing course. In this project I explored some approaches to parallelizing the longest common subsequence problem and some experimental results achieved with an MPI implementation of the parallel algorithm designed for the LCS problem.

Introduction

The longest common subsequence problem is the following: given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “acf”, “cef”, ”acefg” are subsequences of “abcdefg”. This is a classic problem in computer science and has several applications such as in molecular biology or file comparison.
This problem can be efficiently solved in a dynamic programming approach. The idea is based on the following optimal substructure property: let the input sequences be $X_{0..m-1}$ and $Y_{0..n-1}$ of lengths m and n respectively. And let $L(X_{0..m-1}, Y_{0..n-1})$ be the length of LCS of the two sequences X and Y.

- If last characters of both sequences match $X_{m-1} = Y_{n-1}$ then $L(X_{0..m-1}, Y_{0..n-1}) = 1 + L(X_{0..m-2}, Y_{0..n-2})$
- If last characters of both sequences do not match $X_{m-1} \neq Y_{n-1}$ then $L(X_{0..m-1}, Y_{0..n-1}) = max(L(X_{0..m-2}, Y_{0..n-1}), L(X_{0..m-1}, Y_{0..n-2}))$
- In the base case we have that if X or Y is an empty string, then the LCS is also empty.

These property leads immediately to a dynamic programming solution. Here we focus on the bottom-up approach in which a matrix of size $m \times n$ is evaluated, whose entry (i, j) is the length of LCS of $X_{0..i}$ and $Y_{0..j}$. It is not hard to see that each entry of the table depends only on the upper neighbor, the upper-left neighbor, and the left neighbor.
Usually the table is evaluated simply in row major order. This poses a problem for parallelization, as within a row, each entry depends on the previous entry. However, if we process the table in “diagonal-major” order, then new doors are opened for parallel approaches.

Sequential algorithm. The first task was to implement a sequential algorithm of the diagonal-order evaluation of the DP table. The time complexity is clearly $\Theta(m * n)$

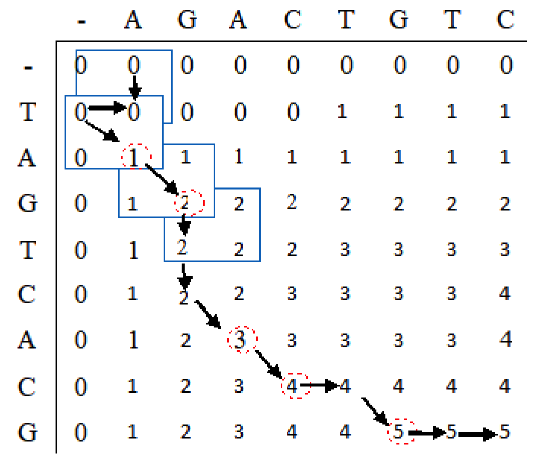
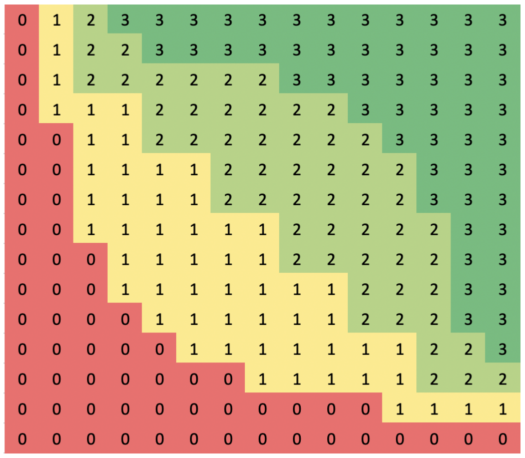


Fig. 1. Example of the computation of LCS of two strings

An observation we can make is that each entry depends only on the previous two diagonals, therefore we can reduce the space complexity by throwing away previous diagonals once they are not required. This way, if we force the second input to be the smaller string, the space requirement is reduced to $\Theta(min(m, n))$

The parallel approach

The principal idea is that by computing the table in diagonal-major order, each element of a diagonal can be computed independently given that the previous diagonals are already computed.
To evenly distribute the load to the P processes available, each diagonal is divided into P blocks of the same size (or similar, when its length is not divisible by P).
Figure 2 shows an example of how the computation of the table is distributed among the processes.
However, the algorithm would not work without some communications among the processes. The problem appear when we evaluate an entry at the boundary between two processes. See figure 3 as an example: the entries at the boundary depend on entries assigned to adjacent processes in the previous diagonals.
To fix this issue, the solution is the following: when a process evaluates its own block of a diagonal, it also receives the last element of the previous process and the first element of the following process. Symmetrically, it also has to send the extremes of its own block to the adjacent processes. For example process 1 would send the first entry of its own block to process 0 and the last element to process 2, then it would



0	1	2	3	3	3	3	3	3	3	3	3	3	3	3
0	1	2	2	3	3	3	3	3	3	3	3	3	3	3
0	1	2	2	2	2	2	2	3	3	3	3	3	3	3
0	1	1	1	2	2	2	2	2	2	3	3	3	3	3
0	0	1	1	2	2	2	2	2	2	2	3	3	3	3
0	0	1	1	1	1	2	2	2	2	2	2	3	3	3
0	0	1	1	1	1	2	2	2	2	2	2	3	3	3
0	0	1	1	1	1	1	1	2	2	2	2	2	3	3
0	0	0	1	1	1	1	1	2	2	2	2	2	3	3
0	0	0	1	1	1	1	1	1	1	2	2	2	3	3
0	0	0	0	1	1	1	1	1	1	2	2	2	3	3
0	0	0	0	0	1	1	1	1	1	1	1	2	2	3
0	0	0	0	0	0	0	1	1	1	1	1	2	2	2
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 2. Computation distribution of the DP table between the parallel processes. The values represent the rank of the process the entry has been assigned to. In this case the inputs consists of 2 strings of 14 characters and 4 parallel processes have been assigned.

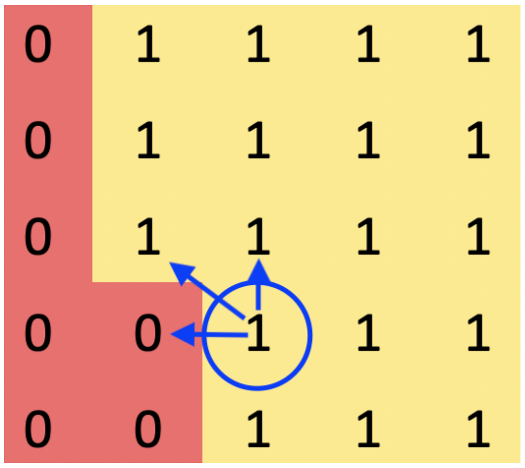


Fig. 3. Computation of an entry at the boundary. The entry circled in blue is assigned to process 1 but depends on an entry of the previous diagonal which has been assigned to process 0.

receive the last element from process 0 and the first element from process 2. This way, each process receives some information at the boundary of adjacent blocks.
The space complexity can also be reduced the same way as before, by storing just the previous two diagonals.

Results

The following results are the running time measured on the Power7 cluster provided by the university. Several test instances were executed with different combinations of input size and number of parallel processes (pairs of string of 100/10k/100k/500k characters each and 8/16/32 processes). Tables 1 and 2 display the time of execution on both parallel and sequential algorithms.

	100 chars	**10k chars**	**100k chars**	**500k chars**
8 tasks	0,0014	0,19	11	259
16 tasks	0,0018	0,19	9,1	203
32 tasks	0,0035	0,22	7,8	172

Table 1. Parallel execution time in seconds varying input size and number of processes (-O3 optimization)

	100 chars	**10k chars**	**100k chars**	**500k chars**
8 tasks	0,000055	0,49	49	1227
16 tasks	0,000087	0,79	55	1229
32 tasks	0,000098	0,79	80	2020

Table 2. Sequential execution time in seconds (-O3 optimization)

As we can see, the running time is typically lower on the parallel implementation except for small input size such as 100 characters. This is expected because of the overhead of parallelizing the job and passing messages between adjacent processes. At larger inputs this overhead matters less and the advantage of parallelizing becomes more significant.
Another observation we can make is that the sequential running time is slightly longer when more tasks are allocated, this is because just one of the allocated processes is used for running the sequential algorithm.
Table 3 shows more clearly the speedup, which is calculated as sequential running time over parallel running time.

Speedup	**100 chars**	**10k chars**	**100k chars**	**500k chars**
8 tasks	0,04	2,58	4,45	4,74
16 tasks	0,05	4,16	6,04	6,05
32 tasks	0,03	3,59	10,26	11,74

Table 3. Speedup (sequential time / parallel time)

The result of the speedup measurement conformed to our expectations: the parallel algorithm is more advantageous at larger input size and larger number of parallel processes. At very small input size the speedup is smaller than 1 which comply to our previous observation.
Table 4 shows the results obtained using different optimization flags. We notice that level 3 optimization performs better on both 8 and 16 processes, therefore this is the flag I used for the previous measurements.

	-O0	**-O2**	**-O3**
8 tasks	56 sec	16 sec	11 sec
16 tasks	38 sec	10 sec	9.1 sec

Table 4. Parallel execution time using different optimization flags (input size 100k characters)

Conclusions

In conclusion, the results achieved are very satisfying. The speedup is up to 60% of the theoretical upper bound which is the number P of parallel tasks. Considering the communication overhead between the processes, the speedup values are indeed non expected to be as high as P, so these results seem fairly good to me.
The only problems I encountered on testing the implementation on the Power7 cluster are the following:

- with very big input sizes such as 1 million characters for each input string, the program crashes giving a segmentation fault error during the loading of the input. I suppose that particular permissions are required in order to allocate a lot of memory.
- I could not run the program on 64 parallel processes. When I try to run the job, it remains indefinitely in queue and never gets executed by the machine.

There is only one downside that I could think of this parallel implementation. The previous results are based on the assumption that only the length of the LCS is needed, but if we also had to reconstruct the longest common subsequence, then the information needed would be distributed across the processes and it's computational expensive to transfer all the data to the master process. However, for most of the applications such as in molecular biology, the length of the LCS is enough.
To sum up, this parallel approach to the longest common subsequence is based on a simple idea, however the implementation proved to be very challenging especially at managing the indexes and rounding problems but at the end the outcome is very satisfactory.