Longest Common Subsequence problem

Definition: Let $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ be two sequences, where each $x_i, y_i \in$ alphabet Σ. We say that a sequence $Z = (z_1, \ldots, z_t)$ is a subsequence of X if there exists a strictly increasing sequence $(i_1, \ldots, i_t)$ of indices of X such that $z_j = x_{i_j}$, for all $1 \leq j \leq t$. The Longest Common Subsequence (LCS) problem consists in finding a common subsequence Z of *both* X and Y, of maximum length.
Without loss of generality, we assume $m \leq n$ in the following. We first describe a sequential algorithm for the LCS problem, and then we move to the more interesting parallel case, for which we propose an algorithm that borrows its structure from the sequential one.

Sequential algorithm

There's a well-known algorithm (see [1] or [2]) based on dynamic programming, that we propose here for the sequential case, which exploits the optimal substructure of the problem. Let M be an $(m+1) \times (n+1)$ matrix, where entry $M[i,j]$ represents the length of an LCS of the sequences X_i and Y_j, where X_i is the i-th prefix of X, i.e. $(x_1, \ldots, x_i)$ for $i > 0$ while X_0 is the empty string, and similarly for Y_j. It holds that:

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ M[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(M[i, j-1], M[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

From this simple recurrence relation, it's easy to design a sequential algorithm that solves the LCS problem, filling each row one at a time. It follows that the length of an LCS is stored in $M[m,n]$ and we don't need the rest of the matrix. We can reduce the space requirements from $\theta(mn)$ to $\theta(n)$ by observing that by computing the entries of M row by row, the algorithm only needs the current row and the previous row. If, however, not only the length of the LCS is required, but also the actual subsequence, we need to store the whole matrix M. To re-construct the LCS of sequences X_i, Y_j from M the procedure is to start at entry $M[i,j]$ and follow at each step the previous entry which led to the computation of the current entry.

Parallel algorithm

We will exploit the previous recurrence relation, trying to find a way to parallelize the computation. Let us first define what we mean by principal diagonal of M. Here M is the same as in the previous paragraph, with the first column and the first row removed (which don't require to be computed at all, since they consist of zeros). Hence $M[i,j]$ contains the length of an LCS of X_{i+1} and Y_{j+1}.

Definition: The M's **principal diagonal** of index d, for $0 \le d \le m+n-2$, is the *ordered* set of entries

$$D(d) = \begin{cases} \{M[0,d], M[1,d-1], \dots, M[d,0])\} & \text{if } 0 \le d < m \\ \{M[0,d], M[1,d-1], \dots, M[m-1, d-m+1])\} & \text{if } m \le d < n \\ \{M[d-n+1, n-1], M[d-n+2, n-2], \dots, M[m-1, d-m+1])\} & \text{if } d \ge n \end{cases}$$

Note how entries in each $D(d)$ will only depend on entries belonging to $D(d-1)$ and $D(d-2)$. In fact each element only depends on three elements from the two previous principal diagonals. This suggests a way to parallelize our initial algorithm: by looking at the CDAG of the computation, each diagonal is a level of the greedy schedule. Hence each entry in each diagonal can be computed in parallel, as long as entries from the previous diagonals have already been computed. From the previous definition, we define $L(d)$ as the length of the principal diagonal d:

$$L(d) = |D(d)| = \begin{cases} d+1 & \text{if } 0 \le d < m \\ m & \text{if } m \le d < n \\ m+n-1-d & \text{if } d \ge n \end{cases}$$

or, more concisely, $L(d) = \min\{d+1, m, m+n-1-d\}$.

Optimal execution order

We have to assign an order of execution to compute every entry in the LCS matrix. The assignment we are looking for has to:

- maximize concurrent computation
- minimize communication costs

To maximize concurrent computation we can look at the CDAG of the matrix, where we compute each cell that has required variables ready as soon as possible; let $P_{\max} > 0$ be the number of processors at our disposal. Let's see how many processors we need to assign to a given principal diagonal d whose length is $L = L(d)$. Notice how if $L < P_{\max}$, $P_{\max} - L$ processors will not work at all, since each entry can be computed in parallel by L processors. So we put $P = \min\{L(d), P_{\max}\}$. Thus we can assign processors as follows:

- $\lceil L/P \rceil$ cells to processor i for $0 \le i < (L \mod P)$
- $\lfloor L/P \rfloor$ cells to processor j for $(L \mod P) \le j < P$

We now focus on the permutations of this sequence, in order to minimize the communication between processors. The intuitive way to achieve this is by having processors assigned to contiguous cells of the diagonal of the matrix and in the same order for each diagonal: this way we increase the probability for each processor to have the required variables from the previous diagonal already stored in its memory. Let D be the principal diagonal of index d; processor i will have to compute entries from $D[s]$ to $D[e-1]$, where:

$$s = \begin{cases} i\left\lceil \frac{L(d)}{P} \right\rceil & \text{if } i < L(d) \mod P \\ (L(d) \mod P) \cdot \left\lceil \frac{L(d)}{P} \right\rceil + (i - (L(d) \mod P)) \cdot \left\lfloor \frac{L(d)}{P} \right\rfloor & \text{otherwise} \end{cases}$$

$$e = \begin{cases} s + \left\lceil \frac{L(d)}{P} \right\rceil & \text{if } i < L(d) \mod P \\ s + \left\lfloor \frac{L(d)}{P} \right\rfloor & \text{otherwise} \end{cases}$$

Or more intuitively using an algorithm:

```python
def diag_start_end(d: int, i: int) -> Tuple[int, int]:
    """
    Parameters:
        - d: int Diagonal index.
        - i: int Processor index.
    Returns:
        A pair (s: int, e: int) indicating the starting/ending cell on th
        diagonal d assigned to processor i.
        Index e is exclusive, while index s in inclusive.
    """
    L_d = diag_length(d)
    # Avoid unnecessary processors
    p = min(P, L_d)
    # Number of cells for the first L_d % p processors
    ceil_size = math.ceil(L_d / p)
    # Number of cells for the other p - L_d % p processors
    floor_size = math.floor(L_d / p) # or ceil_size
    rem = L_d % p
    if i < rem:
        start = i * ceil_size
        end = start + ceil_size
    else:
        start = rem * ceil_size + (i-rem) * floor_size
        end = start + floor_size
    return (start, end)
```

As an example of optimal assignment, we show two possible matrices (5×8 and 4×6)

p0	p0	p0	p0	p0	p0	p0	p0
p1	p1	p0	p0	p0	p0	p0	p0
p2	p1	p1	p1	p1	p1	p0	p0
p2	p1	p1	p1	p1	p1	p1	p0
p2	p2	p2	p2	p2	p2	p1	p0

p0	p0	p0	p0	p0	p0
p1	p0	p0	p0	p0	p0
p1	p1	p1	p1	p0	p0
p1	p1	p1	p1	p1	p0

the left one is computed by 3 processors, while the other by 2.

Bound on the number of messages

Let us define $P(i, j)$ as the index of the processor assigned to entry (i, j), according to the previous scheme.

Remark: It's quite easy to see that during the computation of entry (i, j) processor $P(i, j)$ already has the value of cell $(i-1, j-1)$ stored in its memory, for $i, j \geq 1$. First notice that at least one of $(i, j-1)$ or $(i-1, j)$ is assigned to p: in fact, let's say cell (i, j) lies on diagonal d; then if $L(d-1) = L(d)$, it follows that $P(i, j) = P(i, j-1)$; it's just slightly more difficult to see that if $L(d-1) = L(d) \pm 1$, the statement is still valid. Hence, in either case, cell $(i-1, j-1)$ is known to p, since its value was fetched by p in the previous diagonal.

A performance metric we use for the assignment is the number of messages exchanged by the processors. The exact measure for variable n, m and $P_{\max}$ is hard to obtain from analytical considerations, but we can give an upper bound: clearly we can assume $P_{\max} = \min\{n, m\} = m$ since no diagonal will be longer than m.
In this case *every* cell of *each* diagonal is assigned to a *different* processor. By the remark, it's easy to prove that $P(i, j) = P(i, j-1) \neq P(i-1, j)$ if $i + j \leq n - 1$ and $P(i, j) = P(i-1, j) \neq P(i, j-1)$ if $i + j \geq n$. Hence:

$$\text{messages exchanged} = \sum_{i=1}^{m-1} \sum_{j=1}^{n-1} 1 + \underbrace{m-1}_{\text{first column}} = n(m-1) = \theta(nm)$$

Computing the list of entries assigned to a given processor

Each processor needs to know which entries to compute. We make use of the observations above, i.e. the procedure `diag_start_end`. Notice that processor i will never appear on principal diagonals d s.t. $d < i$ or $d \geq n + m - 1 - i$, since in both cases the length of the diagonal is $\leq i$. Also since no diagonal has length $\geq \min\{m, n\} = m$, we need $i < m$.
The algorithm to determine the list of elements of the whole matrix then is:

```python
def matrix_elements(i:int) -> List[Tuple[int, int]]:
    """
    Parameters:
        - i: int Processor index.
    Returns:
        A list of cells of the LCS matrix assigned to processor i.
    """
    # If there are too many processor this one doesn't do anything
    if i >= M:
        return []
    elements = []
    # Eg. Processor 1 (starting from 0) will never be in the
    # first and last diagonal
    for d in range(i, N+M-1-i):
        start, end = diag_start_end(d, i)
        for e in range(start, end):
            # e if d < N and the diagonal starts from
            # the top, d-N+1+e otherwise
            x = max(0, d-N+1) + e
            # d-e if d < N and the diagonal starts from
            # the top, N-1-e otherwise
            y = min(d, N-1) - e
            elements.append((x,y))
    return elements
```

Some useful functions

Given entry (i, j), it will be useful for the following to know which diagonal index d it corresponds, as well as its position relative to $D(d)$ from top to bottom, which we call $\mathrm{pos}(i, j)$.
Given the coordinates (i, j) the diagonal is clearly $i + j$, while $\mathrm{pos(i, j)}$ is given by row i if $d < n$ and $N - j - 1$ otherwise: this can be condensed into $\min\{i, n - j - 1\}$.

```
def cell_diag(i: int, j: int):
    """
    Parameters:
        - i, j: int Coordinates of a matrix cell.
    Returns:
        The index of the diagonal where the cell belongs.
    """
    return i+j

def cell_diag_index(i: int, j: int):
    """
    Parameters:
        - i, j: int Coordinates of a matrix cell.
    Returns:
        The index of the cell w.r.t. the diagonal it belongs to.
    """
    return min(i, N-j-1)
```

Another useful function is to compute $P(i, j)$.
It's easy to derive a formula from the previous assignment of processors to each diagonal;
set $L = L(i + j)$ and $p = \min\{L, P_{\max}\}$, $q = \lfloor L/p \rfloor$, $r = (L \mod p)$.
We have:

$$P(i,j) = \begin{cases} \left\lfloor \frac{\mathrm{pos}(i,j)}{q+1} \right\rfloor & \text{if } \mathrm{pos(i, j)} < (q+1)r \\ \left\lfloor \frac{\mathrm{pos}(i,j)-r}{q} \right\rfloor & \text{otherwise} \end{cases}$$

Hence the following algorithm:

```
def cell_proc(i: int, j: int):
    """
    Parameters:
        - i, j: int Coordinates of a matrix cell.
    Returns:
        The processor assigned to the given cell.
    """
    d = cell_diag(i, j)
    L_d = diag_length(d)
    pos = cell_diag_index(i, j)
    p = min(L_d, P)
    ceil_size = math.ceil(L_d/p)
    floor_size = math.floor(L_d/p)
    rem = L_d % p
    if pos < ceil_size * rem:
        return math.floor(pos / ceil_size)
    else:
        return math.floor((pos - rem)/ floor_size)
```

Sending the computed values to the proper processes

Each processor doesn't need to keep a copy of the whole matrix $P(i, j)$: the previous formula can be used to find where to send the computed values. As we already mentioned in a remark, each processor needs to send at most one value to a different neighbor: we can use the previous algorithm `cell_proc(i, j)` to check whether the cells $(i + 1, j)$ and $(i, j + 1)$ belong to the current processor and in case of a negative answer we send their values to the proper processes.

We know for sure that the cell on the right can belong either to the current processor or the previous one, while, similarly, the one below either to the current or to the next one, but we couldn't find an usage of this information to improve the algorithm speed. With a litte abuse of notation for MPI signature of `send`, we can write the following pseudocode:

```
def send(value: int, i: int, j: int, p: int):
    """
    Parameters:
        -value: int
            Value to be sent
        - i, j: int
            Coordinates of a matrix cell.
        - p: int
            Process that makes a send
    """
    # Send the value right if needed
    if p != 0: # process 0 never sends right
        # No need to check whether j+1 < N because only processor 0 woul
        # Can either be process p or p-1
        p_right = cell_proc(i, j+1)
        if p_right != p:
            MPI_SEND(p_right, value)
            return # No need to send it below too
    # Send the value below if needed
    if p != P-1: # process p-1 never sends below
        if (i + 1 < N): # Avoid out of bounds
            p_below = cell_proc(i+1, j)
            if p_below != p:
                MPI_SEND(p_below, value)
```

Storing the local portion of matrix M

For big problem sizes we aim to reduce the amount of memory each processor uses. Storing the whole $m \times n$ matrix could be too costly, so each processor should only store the cells it computed and the ones from other processors that it used for its computations. This turns out to be useful also for the reconstruction of an LCS, outlined in the next paragraph. To simplify storing and retrieving data we decided to use an hash table, which guarantees $O(1)$ access time on average.

Reconstruction of an LCS from the M matrix

Once the M matrix has been computed by the parallel algorithm, process $P(m-1, n-1) = 0$ knows entry $M[m-1, n-1]$, i.e. the length of an LCS of X_m and Y_n. We show how to compute an LCS of X_{i+1} and Y_{j+1} starting at entry (i, j): if $x_{i+1} = y_{j+1}$ then process $p = P(i, j)$ checks whether :

1. $M[i, j] = M[i-1, j-1] + 1$
2. $M[i, j] = M[i, j-1]$
3. $M[i, j] = M[i-1, j]$

If $1.$ is true, then p sends x_{i+1} to $p' = P(i-1, j-1)$. If $2.$ or $3.$ is true, then p sends e to $p' = P(i, j-1)$ or $p' = P(i-1, j)$ respectively, where e is the null string. The same procedure applies p', which will prepend its message to the one it just received from p. Once a processor assigned to a cell $(0, j)$ or $(i, 0)$ is reached, the resulting message is the required LCS. Here the number of messages exchanged is at most $m + n$. Here's the pseudocode, with a little abuse of notation as before:

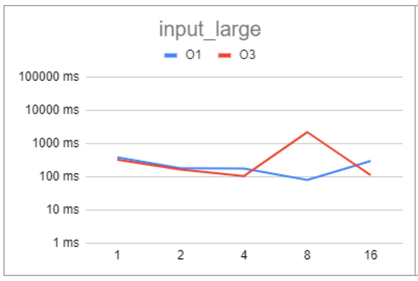
```python
def compute_LCS(i: int, j: int, m: str):
    """
    Parameters:
        - i, j: int Coordinates of a matrix cell.
        - m: Message received from a previous process, which
             signals this process to call this function
    It's assumed P(i, j) is calling this function.
    """
    # M[i, j], M[i-1, j], M[i, j-1], M[i-1, j-1] are all stored in the lo
    # memory of the calling process, as well as x_i, y_j
    # let p_curr be global variable s.t. p_curr = P(i, j)

    if i == 0 or j == 0:
        if x[i] == y[j]:
            print("LCS: " + x[i] + m)
        else:
            print("LCS: " + m)
        return

    if x[i] == y[j]:
        # the receving process has to call this function upon receiving t
        # this can be accomplished using MPI tags
        MPI_SEND(cell_proc(i-1, j-1), x_i + m)
    elif M[i, j] == M[i-1,j]:
        MPI_SEND(cell_proc(i-1, j), m)
    else:
        MPI_SEND(cell_proc(i, j-1), m)
```

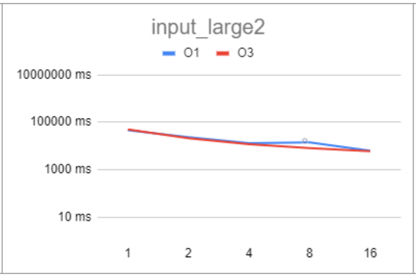
Results and conclusions

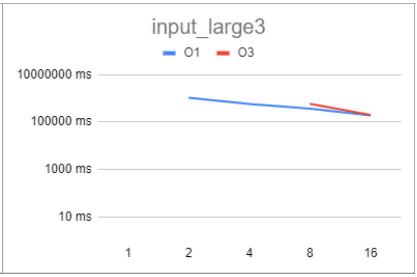
We ran our parallel program on the CAPRI cluster, after compiling with two different optimization flags, O1 and O3.
We tested 5 different kind of input files named `small`, `medium`, `large`, `large_2` and `large_3`.

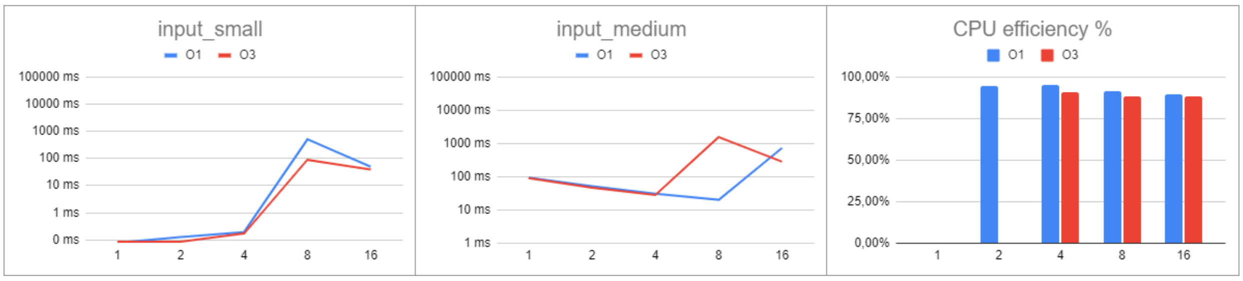
input	size	approx. X, Y size	sequential O3	parallel p = 2 O3	parallel p = 4 O3	parallel p = 8 O3
small	26B	13 char	1 μs	0.1 ms	0.2 ms	88 ms
medium	1.08 KB	550 char	1 ms	45 ms	28 ms	1.5 s
large	1.96 KB	1000 char	3 ms	162 ms	103 ms	2 s
large_2	19.5 KB	10.000 char	418 ms	21 s	12 s	8 s
large_3	97.7 KB	50.000 char	39 s	n.a.	n.a.	587 s

This table does not show all tests, but *only* a few. For the complete numerical details, we refer the reader to the shared spreadsheet document [3]. Instead we proceed to show the graphical details.
In almost every test we performed at least 3 measures of the total time (processing time + communication time), so the graphs below actually show the average of these measures. As we expect, the time decreases as the number of processors increase, at least for the large inputs: for example with input_large2, time decreases by almost a factor of 100 going from 1 to 16 processors. The things are a bit different for small and medium input sizes, where using at least 8 processors degraded the performance in a surprising way. This is probably due to more communications between processors which have a relatively high impact on the performace.









What really striked us is the huge difference of the total time between the sequential version and the parallel version: probably our input files were too tiny to justify going parallel; we have to mention however that quite a good amount of time is spent on research for the best hash table: a few simulations on the large files using 4 processors showed that about 33% of computation time is spent on looking up values; originally we used the STL implementation of the hash map, which yielded a fourfold total time, which led us to use a faster implementation [4]. We are aware that the hash table could be replaced with a faster data structure: for each processor, we could use an array of vectors, each holding elements received by another process or computed by the current process; furthermore this array is indexed using the diagonal number. However the detailed implementation required a careful study of specific cases, so we ended up with an existing data structure, in order to not slow down the flow of the project.
The CPU efficiency values are obtained from the `seff` command on CAPRI and have meaningful values only for some of the jobs. It is worth mentioning that all of the jobs running the parallel algorithm resulted in $> 85\%$ CPU efficiency while the job running the sequential algorithm only reached around 34% CPU efficiency.
We notice also that the sequential algorithm is a lot more cache friendly, since the whole LCS matrix is actually a linear array, whereas the parallel algorithm fails to take advantage of this. We were not able to test the large_3 input on the cluster using 2 or 4 processors, since the required memory for the execution was bigger than 64GB; of course the parallel algorithm requires more RAM since it's written in C++ (objects take more space).

References

[1] *Longest common subsequence problem:*
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
[2] *Thomas H. Cormen, Introduction to algorithms*
[3] *Spreadsheet with numerical results in microseconds:*
https://docs.google.com/spreadsheets/d/1rcYe3zi5sDbGkvDs1t6joy9k7QzM-bF1frSsXAbS6wY
[4] *Github reference to Robin Hood Hash Map:* https://github.com/martinus/robin-hood-hashing