Distribuited Levenshtein distance with MPI

Giuseppe Palazzolo
giuseppe.palazzolo@stud.unifi.it

Federico Vaccaro
federico.vaccaro@stud.unifi.it

Abstract

The Levenshtein distance between two string is a common distance metric. A simple way to compute the edit distance between two given strings, is the Wagner-Fischer algorithm, which translate a recursive procedure to an iterative one through the dynamic programming. Here it is used an algorithm that uses MPI to distribute the work on more processes and allows to calculate the distance between longer characters sequence.

1. Introduction

1.1. Levenshtein distance

Given two string x and y with length respectively $|x|$ and $|y|$, the Levenshtein distance between them is the minimum number of *elementary edits* necessary for transform the string y to the string x [4]. The elementary edits are:

- **Substitutions** of a character of y with one of x;
- **Deletions** of a character;
- **Insertions** of a character;

This cost, is summarized by this recursive definition:

$$lev_{x,y}(i,j) \begin{cases} max(i,j) \text{ if } min(i,j) = 0 \\ min \begin{cases} lev_{x,y}(i-1,j) + 1 \\ lev_{x,y}(i,j-1) + 1 \\ lev_{x,yb}(i-1,j-1) + \mathbf{1}_{(x_i \neq y_i)} \end{cases} \end{cases} \quad (1)$$

where $\mathbf{1}_{(x_i \neq y_i)}$ is the indicator function, equal to 1 if character $x_i \neq y_i$. The distance between x and y is $lev_{x,y}(|x|,|y|)$. Implementing this algorithm through the definition is inefficient, Wagner and Fischer proposed in 1974 [3] an algorithm (**Algorithm 1**) which make use of the dynamic programming, in order to avoid an exaggerated number of function call.

The Wagner-Fischer algorithm computes the edit distance in a bottom-up method, based on the observation that if we reserve a matrix to hold the edit distances between all prefixes of the first string and all prefixes of the second, then we can compute the new values in the matrix reusing the old values through definition **(1)**, and thus find the distance between the two full strings as the last value computed.
The **Algorithm 1**, assuming that $n = |x| \cong |y|$, has $O(n^2)$ time complexity and $O(n^2)$ space complexity.

Algorithm 1 Wegner-Fischer algorithm

```
procedure LEVENSHTEINDISTANCE(x, y)
    D ← Matrix(|x| + 1, |y| + 1)
    for i = 0...|x| do
        D[i][0] ← i
    for j = 0...|y| do
        D[0][j] ← j

    for i = 1...|x| do
        for j = 1...|y| do
            if x_{i-1} ≠ y_{j-1} then
                D[i][j] = min{
                    D[i − 1][j],
                    D[i][j − 1],
                    D[i − 1][j − 1]} + 1}
            else
                D[i][j] = D[i − 1][j − 1]
    return D[|x|, |y|]
```

1.2. Message Passing Interface

MPI (Message-Passing Interface) is a message-passing library interface specification. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. MPI is a specification, not an implementation; there are multiple implementations of MPI and all of these are expressed as functions, subroutines, or methods, according to the appropriate language. [2] Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular in '80 -'90. As architecture trends changed, shared memory processes were combined over networks creating hybrid distributed memory / shared

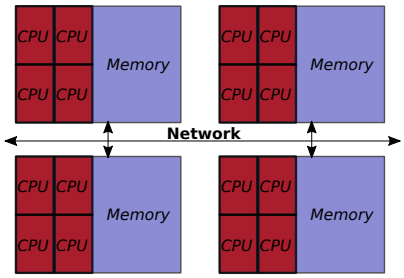

Figure 1. Hybrid system: shared and distributed memory

memory systems. Today it is used both distributed, shared and hybrid(**Figure 1**). [1]

2. Algorithm idea

The idea is to divide the two query , *string A* and *string B*, over all processes so that every process has a different strings couple (**Figure 2**). The processes are arranged in a virtual mesh of size rows $\times$ columns, every process compute edit distance of its sub-matrix and send respectively the column on the right side and the bottom row to the process on the right and bottom, as shown in **Figure 4**. Last number of last process sub-matrix will be Levenshtein distance between the two initial strings.

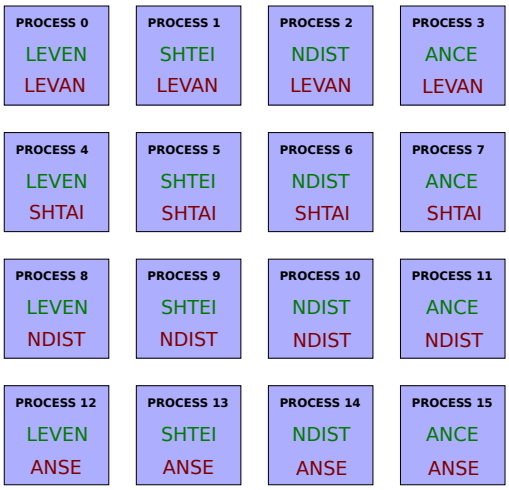

Figure 2. Processes with their unique couple of sub-strings

3. Implementation

In this implementation it is used *openmpi-4.0.0* library for *C++* language and algorithm runs in shared memory. There are three phases:

1. Initialization:
   - Initialize MPI;
   - Create sub-communicators;
   - Scatter;
2. Instantiation of workers;
   - Set their context;
3. Execution;
   - Initialize matrix;
   - Receive;
   - Calculate;
   - Send;

In next paragraphs will be explained every step.

3.1. Inizialization

First of all need to *initialize MPI* session and set `size` i.e. total number of processes and `rank` i.e. unique id of every process; every process belong to one or more *communicator*: an object describing a group of processes (all process belong to `MPI_COMM_WORLD` communicator):

```
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

In order to send sub-strings there are been *created two communicators* to divide process in columns and rows. For examle to divide in rows:

```
int row_comm_rule = rank % process_y;
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD,
  row_comm_rule, &row_comm);
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);
```

Similary it is done for the columns changing the rule. The result is shown in **Figure 3**. Now all new communicator's master processes (`row_rank == 0` and `col_rank == 0`) load the strings and with `MPI_Scatter` send to all processes of their communicator sub-strings.

3.2. Instantiation of workers

Any worker process can receive and send both row and column but according to their position matrix size can change. Worker constructor set two boolean flag to send and receive rows and another two to send and receive columns depending on process position, for example left side processes have to receive only row and send both row and column. only for master process `worker_matrix` will instantiated here.

Furthermore any process define two datatype based on their row and column size setting: dimension, number of element and stride. For example:

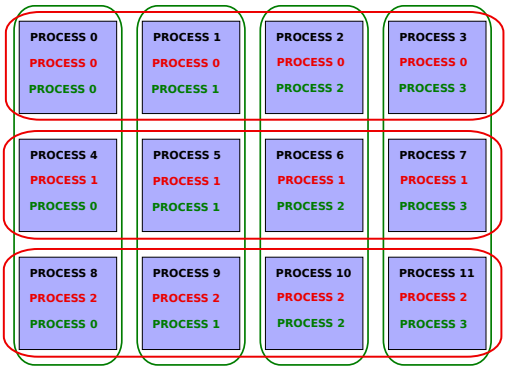

Figure 3. Scatter: row and column communicator

```
MPI_Datatype SUB_VECTOR_COL;
MPI_Type_vector(tile_A, 1, tile_B,
    MPI_INT, &SUB_VECTOR_COL);
MPI_Type_commit(&SUB_VECTOR_COL);
```

3.3. Execution

Execution start with master that calculate edit distance of its matrix and sends row and column result to appropriate processes. Any receiving process is notified by `MPI_Probe` cause of process's status changed. So now matrix is instantiated and row and/or column received are saved respectively in first row and first column of the worker matrix. To receive it is used non blocking receive mpi function due to start both row and column receiving process:

```
MPI_Irecv(&worker_matrix[0], 1,
SUB_VECTOR_ROW, ..., &row_request);
MPI_Wait(&row_request, &status);
```

Now processes can calculate Levenshtein distance, it is used a single thread algorithm like **Algorithm 1** and send row and column to right and bottom processes as it is shown in **Figure 4**. Also here it is used a non blocking function:

```
MPI_Isend(&worker_matrix[tile_B *
(tile_A-1)], 1, SUB_VECTOR_ROW, ...,
            &row_request);
MPI_Request_free(&row_request, &status);
```

Finally the process can delete its sub-strings and worker matrix. Levenshtein distance will be in:
`worker_matrix [ tile_A * tile_B - 1 ]`
of last process.

4. Conclusion: Parallelism and distribution

Thanks to the use of MPI this algorithm has been rendered parallel and distributed. The distributed version was not tested here. The processes started in each machine will

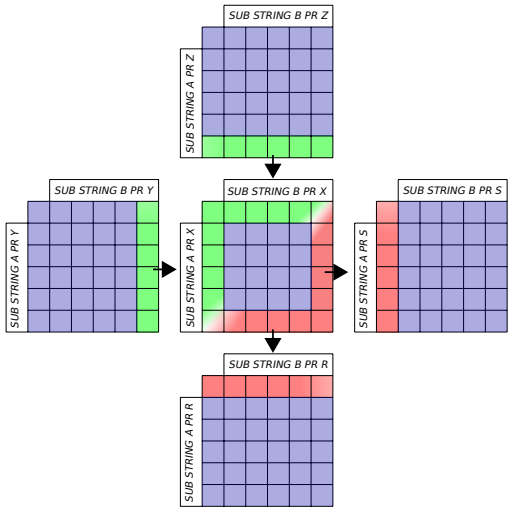

Figure 4. Send and receive process

calculate their sub-matrices and after that send the results (row and/or column). This allows you to compute strings that are much longer because you do not have to load all the sub-matrices but only those that are used in parallel, **Figure 5** both sending and receiving. The parallel computing takes place only along the diagonals, red line **Figure 5**, as each process has to wait the data it need. Therefore the maximum number of parallel running processes will be equal to the number of sub-matrices in the diagonal.

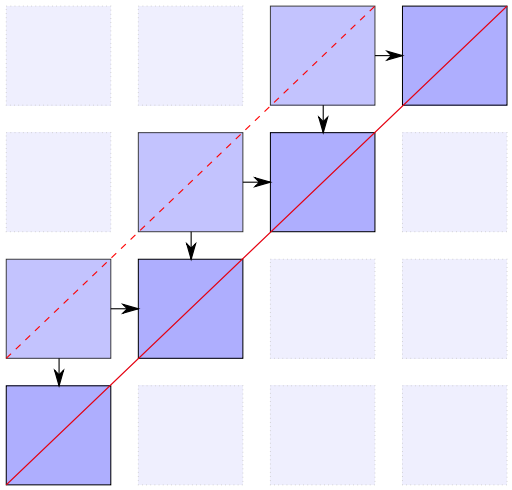

Figure 5. In bold sub-matrices that are actually in memory; Red line links processes that compute in parallel: dashed line sender processes, continuous line receiver processes

References

[1] U. D. of Energy by Lawrence Livermore National Laboratory. Message passing interface (mpi). `https://computing.`

llnl.gov/tutorials/mpi/.

[2] K. University of Tennessee. Message passing interface (mpi). https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[3] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[4] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2019. [Online; accessed 04-02-2019].