University Politehnica of Bucharest
Faculty of Automatic Control and Computer Science
Computer Science and Engineering Department

DIPLOMA PROJECT

Parallelizing the LCA problem

Pop Ioan-Cristian
ionut.pop118@gmail.com

Thesis advisor:

Prof Dr. Ing. Nicolae Țapuș

Bucharest
2023

Universitatea Politehnica din București
Facultatea de Automatică și Calculatoare
Departamentul de Calculatoare

PROIECT DE DIPLOMĂ

Paralelizarea problemei LCA

Pop Ioan-Cristian
ionut.pop118@gmail.com

Coordonator științific:

Prof Dr. Ing. Nicolae Țapuș

București
2023

Contents

Abstract

In 1993, Omer Berkman and Uzi Vishkin introduced the concept of using the RMQ (Range Minimum Query) data structure to efficiently compute the LCA (Lowest Common Ancestor) of nodes in a tree. This had a significant impact on the field of algorithm design and tree-related computations, as it became one of the fundamental method to solve LCA-related problems. Based on that solution, we propose two implementations in parallel and distributed environments (cluster), obtaining even faster execution times.

The implementation is made in `C++`, using `OpenMP` and `MPI`, and can acessed on GitHub: @ionutpop118/ParallelAndDistributedLCA. Speedups above 2x were achieved compared to the serial implementation (and above 4.5x if ignoring I/O).

Sinopsis

În 1993, Omer Berkman și Uzi Vishkin au introdus conceptul de utilizare a structurii de date RMQ (Range Minimum Query) pentru a calcula eficient LCA-ul (Lowest Common Ancestor) a doua noduri dintr-un arbore. Acest lucru a avut un impact semnificativ asupra proiectării algoritmilor și teoriei grafurilor, deoarece a devenit una dintre metodele fundamentale de rezolvare a problemelor legate de LCA. Pe baza acelei soluții, propunem doua implementări în medii paralele și distribuite (cluster), obținând durate de execuție și mai mici.

Soluțiile sunt implementate în `C++`, folosind `OpenMP` și `MPI`, și pot fi accesate pe GitHub: @ionutpop118/ParallelAndDistributedLCA. Soluțiile au timpi de execuție de cel puțin de 2 ori mai mici decât soluția serială (iar 4.5 ori dacă ignorăm I/O).

Acknowledgments

I want to thank may family for their continuous support. Without their support, nothing I ever achieved would have been possible.

I also want to thank everyone from the Romanian community of competitive programming. With their help I learned various algorithms and programming techniques, and achieved many great results at national and international contests.

Finally, want to thank my thesis advisor, Prof. Dr. Ing. Nicolae Țăpuș, for his suppport and trust in me.

1 Introduction

1.1 Context

Algorithms, or in general, solving problems has been one of my main passions since I've started learing `C++` in 2011. While in high-school I've learned various algorithms and programming techniques, I've always challenged myself to optimize my solutions as much as possible, in terms of both time and memory complexity. One of my favorite problems is known as **LCA** – Lowest Common Ancestor of two nodes in a tree.

In university, I've learned more about cache memory, as well as parallel and distributed computing. With the new knowledge acquired, I've decided to go back to this problem and write an optimized implementation.

Notations

Let's consider N = number of nodes in a graph and Q = number of queries to answer.

1.2 Motivation

LCA has lots of practical usages – graph theory, bio-informatics, networking, and so on. However, different variations of the LCA problem exist – the graph may static or dynamic, the number of queries may be equal to N^2 or may vary, and so on.

In this thesis we will approach the case when the graph is static (so no updates are made to it), and the number of queries may vary. For when the number of queries may vary, one solution (further discussed in this thesis) has the time and memory complexity $\mathcal{O}(N \log_2 N + Q)$. This solution scales well with the number of queries, being more flexible.

If the number of queries is equal to N^2, then essentially we have to find the LCA for every pair of nodes. With a very large number of nodes (e.g. $N = 10^7$), the number of queries is unreasonably high. For this particular problem, a solution exists in time and memory complexity $\mathcal{O}(N^2)$.

This thesis aims to provide optimized implementations for the LCA problem, starting form a serial solution, in parallel and distributed systems, which scales with any numbers of processors/devices provided

1.3 Problem

To find an efficient way to solve the LCA problem, in parallel and distributed systems, given a static graph and a list of queries to answer.

1.4 Objectives

This thesis aims to present a solution to the LCA problem in a parallel and distributed environment and provide various `C++` implementations for this problem.

1.5 Solution

The proposed solution is to first implement the algorithm presented by Omer Berkman and Uzi Vishkin, and then adapt it to parallel and distributed computing.

1.6 Results

Including Input and Output times, a speedup of up to 2.2x was obtained from the parallel solution compared to the serial solution, while running with less than 16 threads. Excluding I/O times, the speedup goes above 4x.

1.7 Outline of the Thesis

The remaining of this document is structured in the following way. In chapter 2 we will discuss the solution presented by Harel and Tarjan. In chapter 3, we describe how we adapt that solution to work in a parallel and distributed environemnt. We discuss implementation details in chapter 4. In chapter 5, we apply the algorithm on various datasets and discuss the results. Finally, in chapter 6, we give some final remarks and discuss further improvements.

2 LCA problem and solution

In this section, the LCA problem will be presented, as well as the solution proposed by Omer Berkman and Uzi Vishkin in more detail. It is necessary to understand how this solution works when discussing the parallel and distributed approach.

2.1 LCA problem

2.1.1 Clarifications

For the rest of the thesis, we will consider the following:

- LCA is short for *lowest common ancestor*
- N: the number of nodes
- Q: the number of queries
- $LCA(x, y)$: the lowest common ancestor of nodes x and y.

2.1.2 Definitions

The Lowest Common Ancestor (LCA) problem involves finding the lowest (deepest) common ancestor of two nodes in a tree or directed acyclic graph (DAG). The LCA is defined as the shared ancestor that is located farthest from the root and has both target nodes as descendants. It represents the common point of origin for the two nodes, reflecting their closest common ancestor in terms of tree or graph hierarchy.

Properties of the LCA:

- The LCA of a node with itself is the node itself.
- The LCA is unique for any pair of distinct nodes.
- The LCA can be one of the input nodes if one is an ancestor of the other.

Specific problem discussed

While the same solution works both for DAGs and trees, the problem presented in the thesis will have the input a tree.

The problem statement is: given a tree and N nodes, and Q pair of nodes, find the lowest common ancestor of each pair. Note that nodes will be indexed from 1.

2.1.3 Input and output

The input consists of

- two integers: N – the number of nodes, and Q – the number of queries.
- N integers, where the i^{th} integer describes the parent of the i^{th} node in the tree.
- Q pairs of two integers, defining a query.

The output consists of

- Q integers, where the i^{th} integer represents the answer to the i-th query.

2.1.4 Example

We will use this as the base tree for this chapter of the thesis.

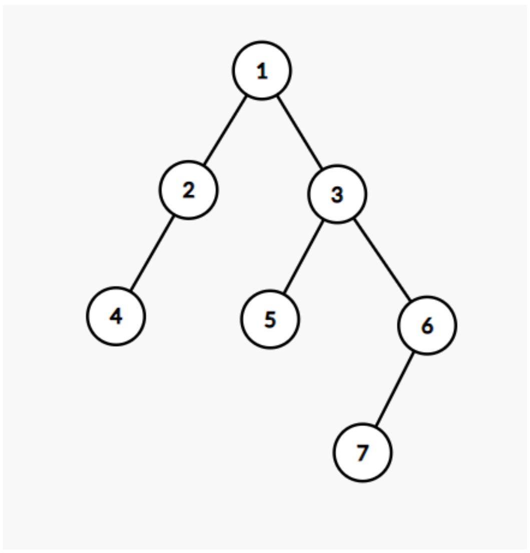

Figure 1: Example graph, generated using CS Academy's graph editor tool

In this graph, we could observe the following:

- LCA(2, 5) = 1
- LCA(5, 7) = 3

Example input/output

```
Input:
7 2 (N and Q)
0 1 1 2 3 3 6 (Parent of each node)
2 5 (First query)
5 7 (Second query)

Output:
1 (Answer to first query)
3 (Answer to second query)
```

2.2 Naive solution

A simple, but inefficient way to answer a query is to take the path to the root for each of the nodes, and then find the last value (starting from root) that appears in both paths.

The time complexity for answering each query is $\mathcal{O}(N)$, therefore the overall time complexity is $\mathcal{O}(NQ)$.

The memory complexity is $\mathcal{O}(N + Q)$, as only two auxiliary arrays of size N are needed for storing the paths.

Example

When calculating LCA(5, 7), we build the paths from the nodes to the root

path from node 5: 5 3 1
path from node 7: 7 6 3 1

We reverse them, and iterate from the root until the values in both paths differ. Note that after 3, the values differ. Therefore, LCA(5, 7) = 3.

2.3 Solution with RMQ

This solution uses the Euler tour of the tree to calculate the LCA of two nodes, and then uses RMQ to improve the time complexity of answering queries.

2.3.1 Euler tour

The first step is to get the Euler tour of a tree. It is a traversal technique similar to the inorder traversal, but when we return to the parent node, we insert the parent node in the tour. The size of the Euler tour is $2N - 1$.

During the Euler tour, we will create 3 arrays:

- *Euler* – containing the Euler tour of the tree
- *Level* – containing the distance of height of each node
- *First* – containing the index of the first occurences of nodes in the Euler tour

Example

For the graph above, after the Euler tour traversals, the arrays look like:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Euler	1	2	4	2	1	3	5	3	6	7	6	3	1
Level	1	2	3	2	1	2	3	2	3	4	3	2	1

Index	1	2	3	4	5	6	7
First	1	2	6	3	7	9	10

2.3.2 Answering a query

Using these 3 arrays, we can now find the LCA of two nodes x and y. The steps are as follow:

1. Extract i = First[x] and j = First[y]. If $i > j$, swap them.
2. From the Level array, calculate pos = the position containing the minimum value from the positions interval [i, j]
3. Extract ans = Euler[pos]. It should be the answer to the query.

This can be done initially in $\mathcal{O}(N)$ time complexity by iterating over the *Level* array.

Example

We would like to calculate LCA(5, 7). We know that First[5] = 7, and First[7] = 10.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Euler	1	2	4	2	1	3	5	3	6	7	6	3	1
Level	1	2	3	2	1	2	3	2	3	4	3	2	1

The minimum value in the Level array, from the 7^{th} to the 10^{th} position, is equal to 2, found on the 8^{th} position. The node in the Euler path from the 8^{th} position is 3, therefore LCA(5, 7) = 3.

2.3.3 Optimizing with RMQ

Since the graph is static, no changes are made to it, therefore none of the 3 arrays will modify. Therefore, we can use the RMQ (Range Minimum Query) data structure to answer the queries in $\mathcal{O}(1)$ time complexity, while building the structure in $\mathcal{O}(N \log_2 N)$

RMQ is a two-dimensional array, with $\log_2 N$ rows and N columns, where RMQ[i][j] represents the minimum value from the positions interval $[j, j + 2^i - 1]$ from an array. In our case, RMQ[i][j] represents the position containing the minimum value from the positions interval $[j, j + 2^i - 1]$ from the Level array.

RMQ[i][j] is calculating using the minimum value from the first half ($RMQ[i-1][j]$) and the second half($RMQ[i-1][j + 2^{i-1}]$): of the interval

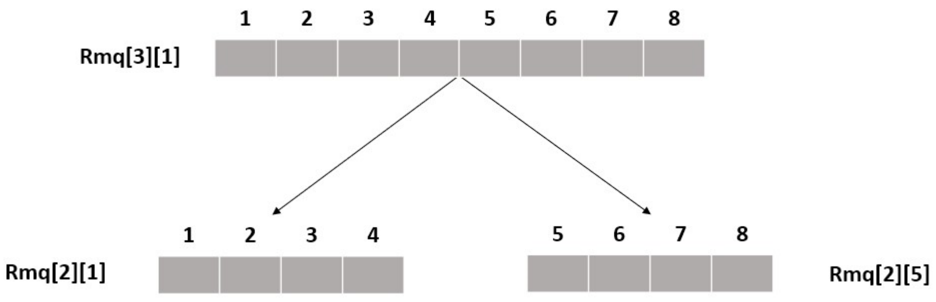

Figure 2: Building the RMQ

Example – finding the minimum from an interval using RMQ

For the graph above, we would like to calculate LCA(2, 5). We know that First[2] = 2, and First[5] = 7, so we would like to know the minimum value the interval [2, 7].

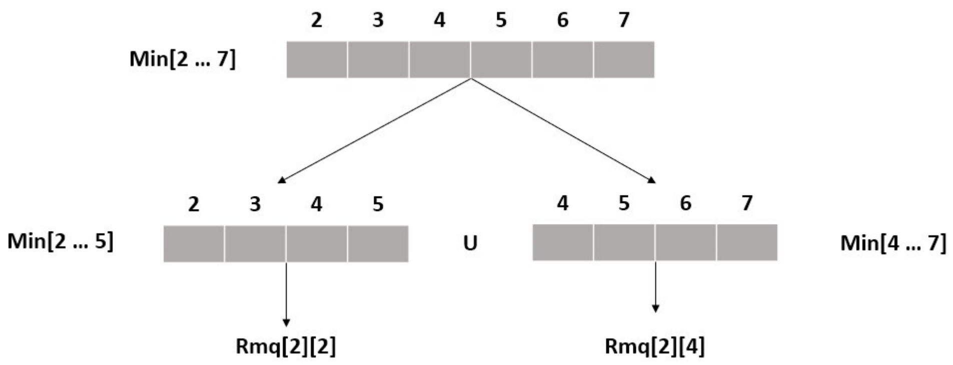

Figure 3: Answering LCA(2, 5)

This can also be written as the minimum value of these intervals: [2, 5] and [4, 7]. The minimum value from [2, 5] can be accessed from RMQ[2][2], while the minimum value form [4, 7] can be accessed from RMQ[2][4]. The time complexity of calculating the LCA becomes $\mathcal{O}(1)$.

2.3.4 Summary of the solution

In short, the algorithms steps are (excluding input/output)

1. Make the Euler tour traversal of the graph
2. Build the RMQ, line by line
3. Answer the queries, using the RMQ

The total time and memory complexity of this solution is $\mathcal{O}(N \log_2 N + Q)$. Compared to other solutions, this one works the best in practice, especially when the number of queries is much larger than the number of nodes.

3 Parallel and Distributed solutions

In this chapter we will discuss how we adapted the solution presented before to work in a parallel and distributed environment. Note that, for all of the solutions, the input/output parts are serial. Only the 3 main steps of the algorithm are affected.

We will consider the following values:

- T = number of threads available
- M = number of machines available

3.1 Parallel environment

The main advantage the parallel environment offers is that all the threads share the exact same memory. Therefore, we don't have to worry about having to copy and share data in between threads.

3.1.1 Euler Tour

While this is a DFS-like algorithm, the fact that we have to insert the node in the list each time we return from a child node makes it rather difficult to parallelize. Using a dynamic programming approach, to calculate the exact positions were we would have to insert the values makes this possible, but the gains are minimal. For now, the serial implementation is being used.

3.1.2 Building the RMQ

We know that calculating RMQ[i] is done strictly using the values from RMQ[i - 1]. Therefore, each row of RMQ can be calculated in parallel. The time complexity becomes $\mathcal{O}\left(\frac{N\log_2 N}{T}\right)$ for this section.

3.1.3 Answering the Queries

Queries can be answered in parallel. The time complexity becomes $\mathcal{O}\left(\frac{Q}{T}\right)$ for this section.

3.1.4 Summary

By building the RMQ and answering the queries in parallel, the time complexity now becomes $\mathcal{O}\left(N + \frac{N\log_2 N + Q}{T}\right)$. If T $\approx \log_2 N$, the time complexity is $\mathcal{O}\left(\frac{N\log_2 N + Q}{T}\right)$. Note that the memory complexity doesn't change at all, it's still equal to $\mathcal{O}(N\log_2 N+Q)$.

3.2 Distributed environment

The main disadvantage of this solution, when in a distributed environment, is that for answering a set of queries, you need access to (likely) the entire RMQ data structure. Calculating the RMQ in parallel requires each machine to send their calculated fragment to the main machine, and then receive the entire line in order to be able to calculate the next RMQ line. These operations are potentially making the solution slower than the serial one, therefore a slightly different solution is presented.

Note that each individual machine has access to their own threads, therefore, it can also calculate in parallel. Additionally, this solution is mainly suited for **cluster computing**.

3.2.1 Proposed solution

Considering the above, we decided to separate the machines into two groups: one group, consisting of a single machine (also the main one, we will name it MAIN), will build the RMQ. The second group, consisting of all other machines (we will name it OTHERS), will answer the queries. Each machine from the second group will have a fragment of the queries assigned to answer.

Note that in the serial implementation, we first build the RMQ, then we answer the queries. Note that it's possible to answer some of the queries while building the RMQ – each query may require a different line from RMQ in order to answer the query. Therefore, the solution is as follows:

1. MAIN receives the input data and sends the queries to OTHERS
2. MAIN builds the Euler tour
3. MAIN builds the current line of RMQ (in parallel)
4. MAIN sends the current line to OTHERS
5. OTHERS now answer the queries that require the current line (in parallel)
6. Repeat the 3 steps above until all the queries are answered
7. MAIN collects the answers to the queries from OTHERS
8. MAIN returns the output data.

The main advantage of this solution is that, in theory, the main machine, after finishing calculating the last line of RMQ, only has to collect the answer to the queries. While answering a query has $\mathcal{O}(1)$ time complexity, it still takes some time due to accessing different data structures and portions of it, inconsistently. Multiple cache misses occur, and just receiving the answers (since some of them are calculated in advance) can be faster.

Each machine from the second group has to answer $\dfrac{Q}{M-1}$ queries. Since there are $\log_2 N$ rows, let's assume that they have to answer $\dfrac{Q}{(M-1)\log_2 N}$ queries each iteration.

Therefore, after receiving the RMQ line, the time complexity becomes $\dfrac{Q}{(M-1)\log_2 N \cdot T}$ to answer the assigned queries.

The time complexity of the transfers is equal to [REDACTED]

3.2.2 Summary

The time complexity of this solution is theoretically $\mathcal{O}\left(\dfrac{N\log_2 N}{T} + \dfrac{Q}{(M-1)\log_2 N \cdot T}\right)$. The main disadvantage of this solution is that it can be slower compared to the previous one because of the additional execution time generated by sharing data structures between the machines.

However, the execution time influenced by the number of queries is much much smaller compared to the one of the previous solution, from $\mathcal{O}\left(\dfrac{Q}{T}\right)$ to $\mathcal{O}\left(\dfrac{Q}{(M-1)\log_2 N \cdot T}\right)$, making this solution a great alternative to the parallel one.

4 Implementation

The implementations are all made in `C++11`. They can also be found on GitHub: @ionut-pop118/ParallelAndDistributedLCA.

4.1 Serial environment

Here is an example implementation of the RMQ build (line by line):

Algorithm 1: Building the RMQ

```
function BuildRmq():
    for $i = 1; i \leq N; i = i + 1$ do
        $rmq[0][i] = i$;
    Let $k = lg[tour\_length]$ ;   /* lg[] contains logarithms in base 2 */
    for $i = 1; i \leq k; i = i + 1$ do
        for $j = 1; j <= tour\_length - 2^i + 1; j = j + 1$ do
            $rmq[i][j] = rmq[i-1][j]$ ;   /* Set minimum as the left half */
            if $level[rmq[i][j]] > level[rmq[i-1][j + 2^{i-1}]$ then
                $rmq[i][j] = rmq[i-1][j + 2^{i-1}]$;
```

The following method is used for answering a query:

Algorithm 2: Answering a query

```
Input   : $x, y$ = two nodes from the graph.
Output: $ans$ = the lowest common ancestor of $x$ and $y$.
function LCA(x, y):
    Let $x_f = first[x]$;
    Let $y_f = first[y]$;
    if $x_f > y_f$ then
        $swap(x_f, y_f)$;
    Let $mid = lg[y_f - x_f + 1]$;
    Let $ans = rmq[mid][x_f]$;
    if $level[ans] > level[rmq[mid][y_f - 2^{mid} - 1]$ then
        $ans = rmq[mid][y_f - 2^{mid} - 1]$
    return $ans$;
```

4.2 Parallel environment

For this solution, the parallelization was achieved by building the RMQ lines and answering the queries in Parallel. We used OpenMP for the parallelization, using the auto scheduler.

The affected sections of the code will look like this:

Algorithm 3: Building the RMQ – in parallel

```
function BuildRmq():
    ...;
    for $i = 1; i \leq k; i = i + 1$ do
        #pragma omp parallel for private($j$)
        for $j = 1; j <= tour\_length - 2^i + 1; j = j + 1$ do
            ...;
```

Algorithm 4: Answering the queries – in parallel

```
#pragma omp parallel for private($i$)
for $i = 1; i \leq q; i = i + 1$ do
    $ans[i] = LCA(qrr[i].first, qrr[i].second)$;
```

4.3 Distributed environment (cluster computing)

We used `MPI` for the distributing the tasks to multiple machines, in combination with `OpenMP` to further optimize the solution. MAIN uses `MPI_Bcast` to share various data to the other machines (Euler tour, RMQ lines, etc.), while OTHERS use once `MPI_Send` to share their query results.

The rank of each machine is numbered from 0 to $proc\_cnt - 1$. MAIN will have the rank 0. The following algorithm is used to determine the interval of queries each machine has to answer (excluding MAIN):

Algorithm 5: Interval of queries to answer, based on rank

```
Int $start\_idx = 1 + (rank - 1) * (double)q/(proc\_cnt - 1)$;
Int $end\_idx = 1 + (rank) * (double)q/(proc\_cnt - 1)$;
if $end\_idx > q + 1$ then
    $end\_idx = q + 1$;
for $i = start\_idx; i < end\_idx; i = i + 1$ do
    ...;
```

The following section describes how the RMQ is build my MAIN, while OTHERS answer the queries.

Algorithm 6: Building the RMQ and answering queries with MPI

```
for $i = 1; i \leq lg[tour_length]; i = i + 1$ do
    if $rank == MAIN$ then
        $BuildRmqLine(i)$;
    $MPI\_Bcast(\&rmq[i][0], tour\_length + 1, MPI\_INT, MAIN, MPI\_COMM\_WORLD)$;
    if $rank! = MAIN$ then
        $AnwerQueriesWithLine(i)$;
```

5 Results

In this Chapter we will analyze the results obtained after testing the solutions in different environments, and then discuss which solution has the best potential depending on the dataset

5.1 Unit Testing

In order to validate all the solutions, we generated a set of random tests and made sure the outputs of all the implementations (Serial, Parallel and Distributed) would match. The test generator used for this can be found on the GitHub page.

The tests were generated in a way that queries will use nodes far from the root node. Therefore, the time complexity of answering these tests using the naive solution is high.

5.2 Results and analysis

Disclaimer

We were unable to get accurate results for the distributed solution. For the `MPI` solution, the testing environment distributes all the processes on a single machine, therefore the combination of `MPI & OpenMP` could not be used. While the solution does work (each process has their unique threads to work with), there are no time gains from `OpenMP`.

The execution times shown for the distributed solution will not use the parallel optimizations for each individual machines. We will then show the expected execution times if the testing environment would support the combination of `MPI & OpenMP`.

5.2.1 Testing environment

The solutions were tested using the university's clusters. More information can be found at https://infrastructure.pages.upb.ro/wiki/docs/grid.

Note: on a modern computer (e.g. Intel® Core™ i7-11370H processor, 16GB of RAM), the algorithms may run up to 15 times faster than on this environment.

5.2.2 Execution times

The following values were used for testing these solutions:

- $N = 10^6$
- $Q = 2 \cdot 10^7$

For each solution, multiple runs were executed with a different number of processors available. Only the ones with 4, 8 and 16 are displayed.

Additionally, two types of result will be shown: including and excluding I/O times. In order to have precise results, the times were calculated as the average execution time of multiple runs. The execution times are rounded to the first decimal.

Execution times – including I/O

Solution	Serial	Parallel	Parallel	Parallel	Distrib	Distrib	Distrib
Variant	1 proc	4 procs	8 procs	16 procs	4 procs	8 procs	16 procs
Duration (s)	14.2	8.7	7.5	7	9.8	9.6	9.4
Speedup ($\times$)	1	1.63	1.89	2.02	1.45	1.47	1.51

Execution times – excluding I/O

Solution	Serial	Parallel	Parallel	Parallel	Distrib	Distrib	Distrib
Variant	1 proc	4 procs	8 procs	16 procs	4 procs	8 procs	16 procs
Duration (s)	9.1	3.6	2.4	1.9	4.7	4.5	4.3
Speedup ($\times$)	1	2.53	3.8	4.78	1.94	2.02	2.12

Analysis

We can observe that I/O operations take approximately 5.1 seconds, largely affecting the duration time of the parallel and distributed solution. For the serial solution, I/O operations take 36% of the total execution time, while for the parallel solutions, I/O operations take up to 73% of the total execution time.

The speedup obtained by the parallel solution is 2.02 when using 16 processes. However, if the solution would be used as a library function (take the input as parameters and return the output), a speedup of 4.78 is obtained.

The distributed solution does not scale that well, in this environment – because the RMQ construction is done by a single machine. Minor time improvements are gained by dividing the queries to more machines.

5.2.3 Estimated execution times of MPI + OpenMP

By calculating the exact duration it takes to build the RMQ and as answer the Queries (for Serial and Parallel solutions), as well as analysing the theoretical time complexity of each solution, we were able to estimate the execution times of the distributed solutions.

The distributed solution will use a number of processes per machine equal to the number of machines. For example, the solution which used to have 4 processes, now will have 4 machines were each machine has 4 processes available.

We will consider mch/prc as a short way to say machines and processes available

Estimated execution times – including I/O

Solution	Serial	Parallel	Parallel	Parallel	Distrib	Distrib	Distrib
Variant	1 proc	4 procs	8 procs	16 procs	4 mch/prc	8 mch/prc	16 mch/prc
Duration (s)	14.2	8.7	7.5	7	7.9	7.5	7.3
Speedup (×)	1	1.63	1.89	2.02	1.79	1.89	1.95

Estimated execution times – excluding I/O

Solution	Serial	Parallel	Parallel	Parallel	Distrib	Distrib	Distrib
Variant	1 proc	4 procs	8 procs	16 procs	4 mch/prc	8 mch/prc	16 mch/prc
Duration (s)	9.1	3.6	2.4	1.9	2.8	2.4	2.2
Speedup (×)	1	2.53	3.8	4.78	3.25	3.8	4.13

Analysis

We observe that, in this specific test case, the distributed solution is faster than the parallel solution when there are less resources available, but does not scale as well as the available resources increase, even though the theoretical time complexity is better. That is because the solution has increased execution times generated by MPI_Bcast and MPI_Send calls.

However, the distributed solution scales much better as the number of queries is much larger than the number of nodes. While building the RMQ is maybe the most expensive operation in terms of time complexity of the distributed solution, a data set with a small number of nodes and a large number of queries will generate much better results than the parallel solution.

5.2.4 Conclusions

The Parallel solution is very efficient and has great results when $Q \leq N \log_2 N$, considering the resources available. The executions times are similar to the distributed solution, however, the later one scales batter as the number of queries increases, the less resources are available.

6 Closing Remarks

In this chapter we will present a summary of the solutions, and our recommendation on which solution to use, depending on the input data. After, we will present of list of further improvements that will be made, providing even more accurate data.

6.1 Solutions

We proposed two solutions – one in a parallel environment, and one in a distributed environment. As stated in the previous chapters, we observe that the two solutions are not necessarily one faster than another – they have different execution times depending on the dataset.

Our recommendation is to use the parallel solution if $Q \simeq N \log_2 N$, because the execution times are very close to the ones of the distributed solution. While they could be slightly slower, the amount of resources used is significantly slower.

However if $Q \gg N \log_2 N$, the distributed solution is recommended. While the parallel solution has a single process answering the queries in parallel, the distributed solution has multiple machines answering those queries in parallel.

6.2 Further improvements

There are various aspects that we will go trough in the near future and update the solutions based on the results. Some of them are:

6.2.1 Parallelizing the Euler tour

As stated before, it is possible, but with the initial implementation the results are underwhelming. We will try to fix it or look for a new approach, and update the solution once satisfying results are achieved.

6.2.2 Provide and analyse more testing data

Currently this paper covers a single dataset. We plan to update this as more datasets are properly analysed. Additionally, in order to further optimize answering the queries, we will look at how many queries require each individual line of the RMQ.

6.2.3 Modifying the number of machines from each group

Currently there's a single machine working on RMQ and all others answering queries – we could assign more machines to build the RMQ. Based on that, we can make the distributed solution to be more efficient when the number of queries is much smaller than $N \log_2 N$.

6.2.4 Improve I/O execution times

`C++20` provides new alternatives for I/O compared to `C++11`. Updating the solutions to this standard may improve the I/O execution times. Note that the solutions can still be used by providing the input/output data as parameters, as a library function.

References

[Aca] CS Academy. Graph editor. `https://csacademy.com/app/graph_editor/`.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[CDK+01] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[oB] University Politehnica of Bucharest. Ghid folosirea gridului instituțional. `https://infrastructure.pages.upb.ro/wiki/docs/grid/`.