PARALLEL & DISTRIBUTED COMPUTING

PROJECT REPORT

Made By:

Farhan Ahmed (21K-4600)

Hassaan Shahid (21K-3177)

Aimal Amir (21K-3339)

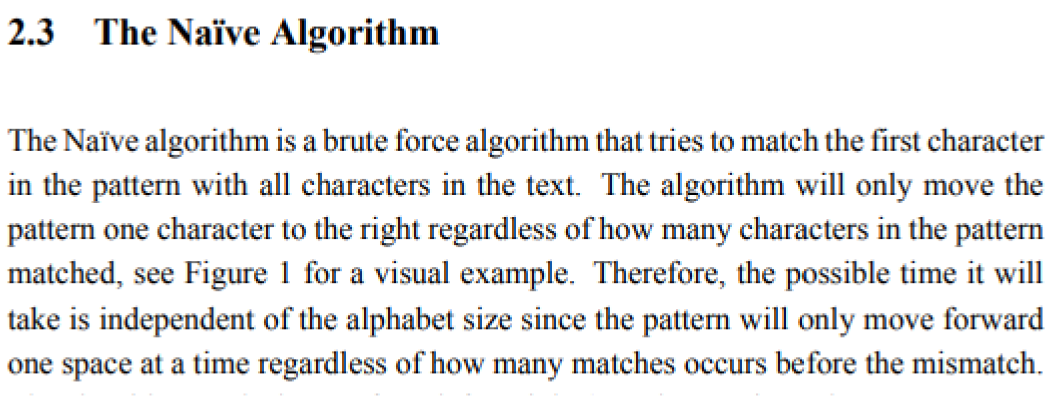
Course Instructor:

Dr Nausheen Shoaib

Introduction:

This project focuses on parallelizing two vital algorithms—Brute Force Pattern Matching and Breadth-First Search Traversal—using OpenMP and MPI for enhanced computational efficiency. We will examine serial and parallel pseudocode for both algorithms, followed by a performance comparison in OpenMP and MPI. The evaluation will encompass computational time, communication overhead, and scalability across varying input sizes. The report concludes with an overall performance comparison of OpenMP and MPI based on total execution time, providing insights into the strengths and trade-offs of each parallelization approach.

I. Brute Force Patter Matching:



2.3 The Naïve Algorithm

The Naïve algorithm is a brute force algorithm that tries to match the first character in the pattern with all characters in the text. The algorithm will only move the pattern one character to the right regardless of how many characters in the pattern matched, see Figure 1 for a visual example. Therefore, the possible time it will take is independent of the alphabet size since the pattern will only move forward one space at a time regardless of how many matches occurs before the mismatch.

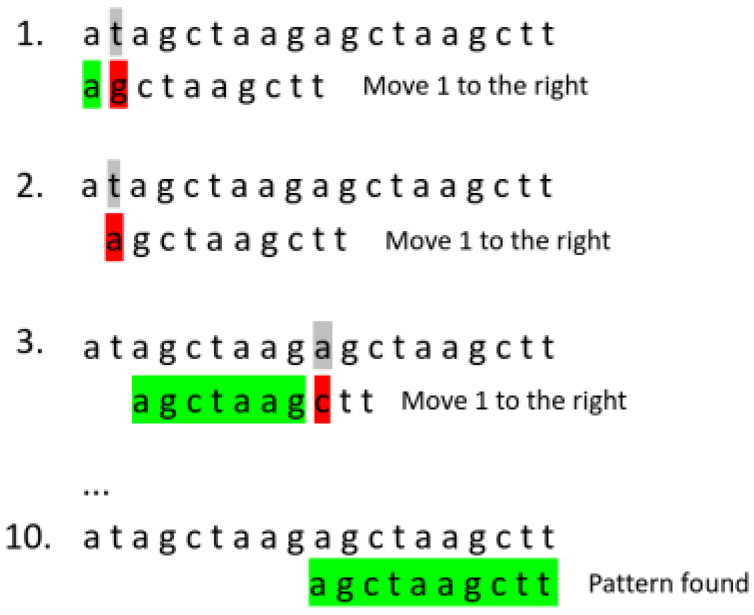
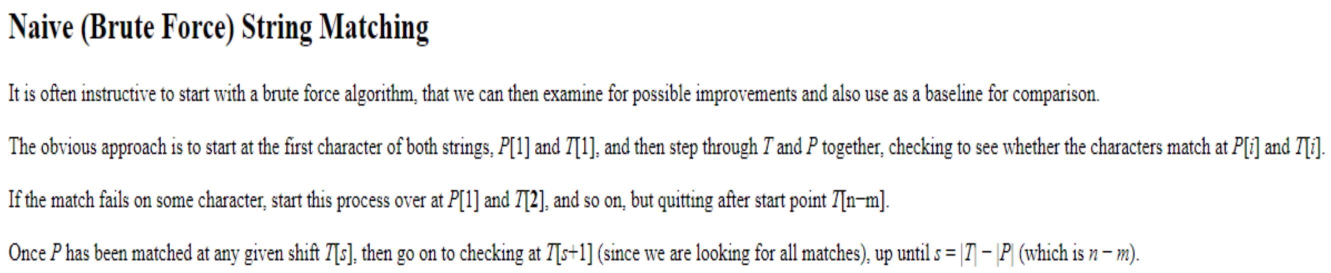

1. a t a g c t a a g a g c t a a g c t t
a g c t a a g c t t Move 1 to the right
2. a t a g c t a a g a g c t a a g c t t
a g c t a a g c t t Move 1 to the right
3. a t a g c t a a g a g c t a a g c t t
a g c t a a g c t t Move 1 to the right
...
10. a t a g c t a a g a g c t a a g c t t
a g c t a a g c t t Pattern found

Figure 1: The figure shows an example of the serial Naïve algorithm.

- **Serial Pseudocode:**



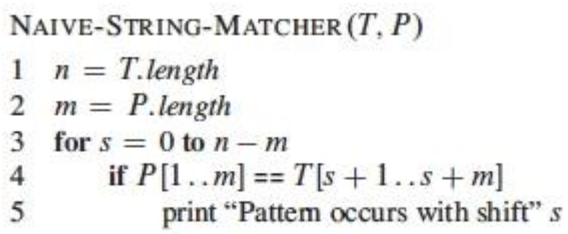
Naive (Brute Force) String Matching

It is often instructive to start with a brute force algorithm, that we can then examine for possible improvements and also use as a baseline for comparison.

The obvious approach is to start at the first character of both strings, *P*[1] and *T*[1], and then step through *T* and *P* together, checking to see whether the characters match at *P*[*i*] and *T*[*i*].

If the match fails on some character, start this process over at *P*[1] and *T*[**2**], and so on, but quitting after start point *T*[n−m].

Once *P* has been matched at any given shift *T*[*s*], then go on to checking at *T*[*s*+1] (since we are looking for all matches), up until *s* = |*T*| − |*P*| (which is *n* − *m*).

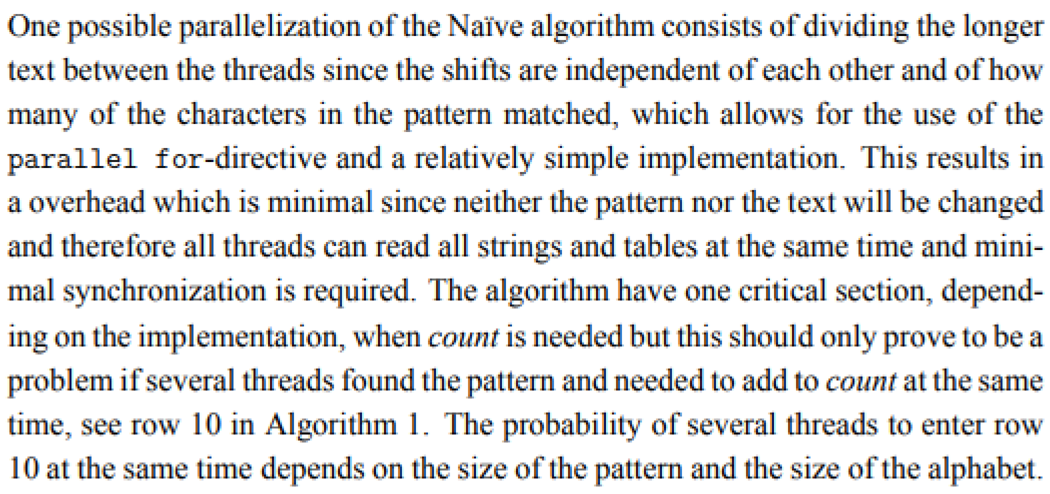


```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```

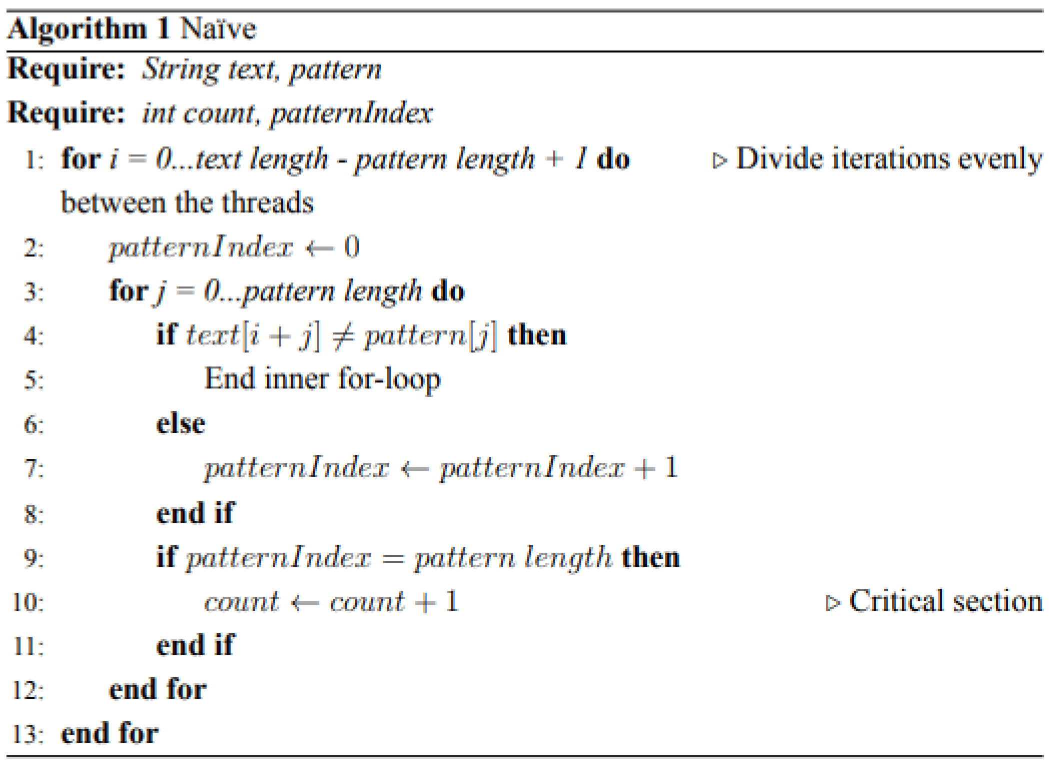
Reference:

https://pd.daffodilvarsity.edu.bd/course/material/book-430/pdf_content (Book: Introduction to Algorithms, Third Edition)

• Parallel Pseudocode:



One possible parallelization of the Naïve algorithm consists of dividing the longer text between the threads since the shifts are independent of each other and of how many of the characters in the pattern matched, which allows for the use of the `parallel for`-directive and a relatively simple implementation. This results in a overhead which is minimal since neither the pattern nor the text will be changed and therefore all threads can read all strings and tables at the same time and minimal synchronization is required. The algorithm have one critical section, depending on the implementation, when *count* is needed but this should only prove to be a problem if several threads found the pattern and needed to add to *count* at the same time, see row 10 in Algorithm 1. The probability of several threads to enter row 10 at the same time depends on the size of the pattern and the size of the alphabet.



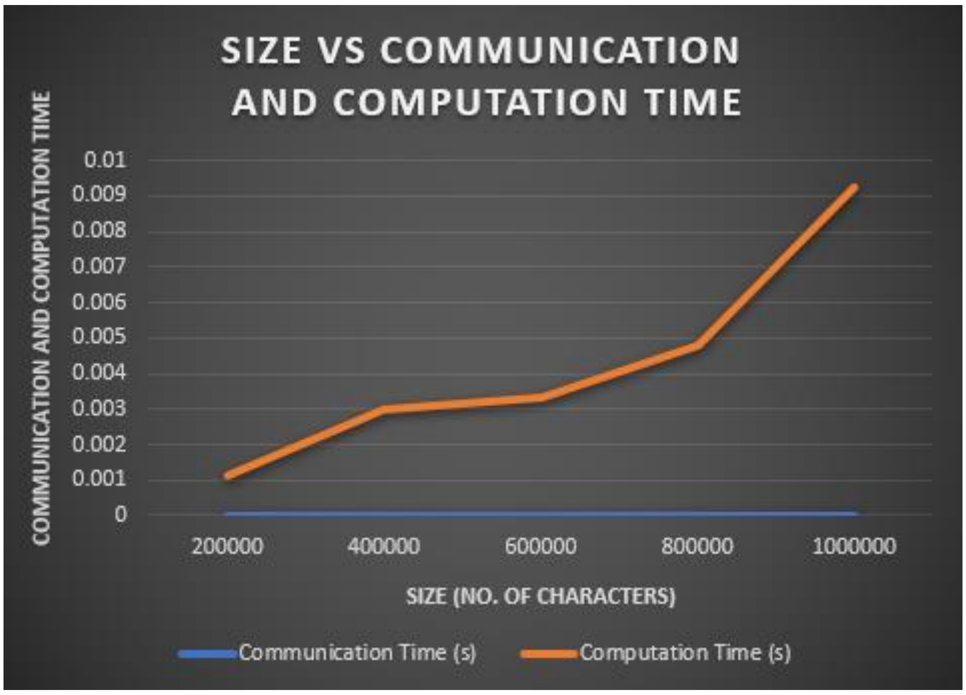
Algorithm 1 Naïve

```
Require: String text, pattern
Require: int count, patternIndex
 1: for i = 0...text length - pattern length + 1 do      ▷ Divide iterations evenly
    between the threads
 2:     $patternIndex \leftarrow 0$
 3:     for j = 0...pattern length do
 4:         if $text[i + j] \neq pattern[j]$ then
 5:             End inner for-loop
 6:         else
 7:             $patternIndex \leftarrow patternIndex + 1$
 8:         end if
 9:         if $patternIndex = pattern\ length$ then
10:             $count \leftarrow count + 1$                ▷ Critical section
11:         end if
12:     end for
13: end for
```

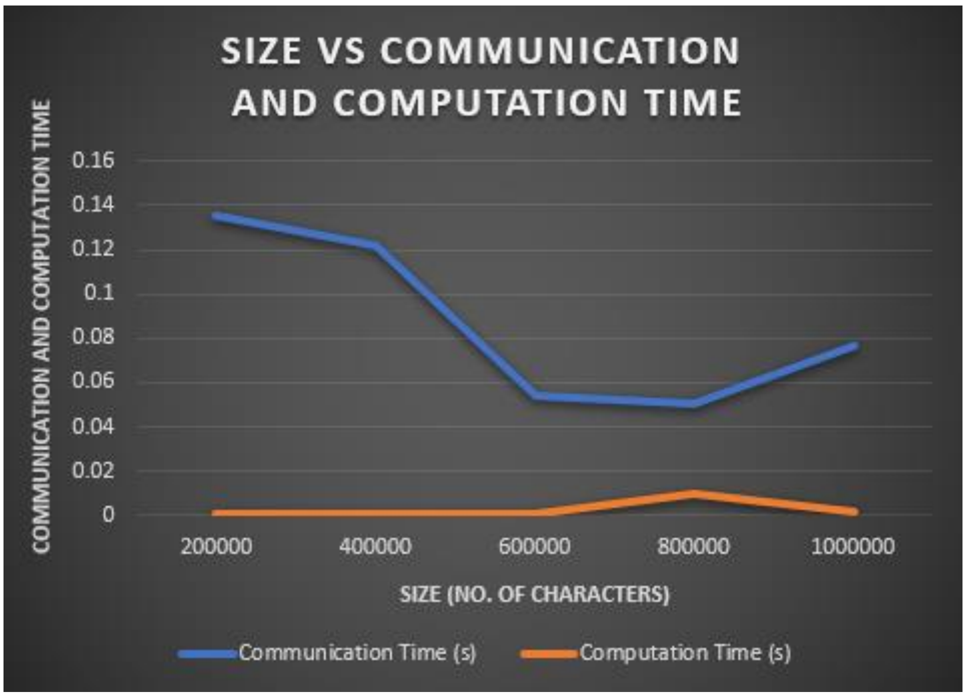
Reference: https://www.diva-portal.org/smash/get/diva2:1676516/FULLTEXT01.pdf (Research Paper: A Parallelized Naïve Algorithm for Pattern Matching)

Performance Comparison:

- **OpenMp:**



- **MPI:**



- **Overall Performance:**

1. Communication Time:

- **OpenMP:** Across all tested sizes (200,000 to 1,000,000 characters), OpenMP consistently demonstrated zero communication time. This indicates that communication overhead in OpenMP for the given Brute Force Pattern Matching algorithm is negligible.

- **MPI:** MPI, on the other hand, displayed varying communication times across different sizes. However, the values remain relatively low, suggesting efficient communication handling. The highest observed communication time was 0.135494 seconds for a size of 200,000 characters.

2. Computation Time:

- **OpenMP:** Computation time in OpenMP increased gradually with the size of the input, ranging from 0.001152 seconds for 200,000 characters to 0.009278 seconds for 1,000,000 characters. The trend indicates a reasonable scalability of computation in OpenMP.

- **MPI:** Computation time in MPI also increased with input size, with varying values. The highest computation time observed was 0.009699 seconds for 800,000 characters. MPI demonstrates competitive computation times, especially as the input size grows.

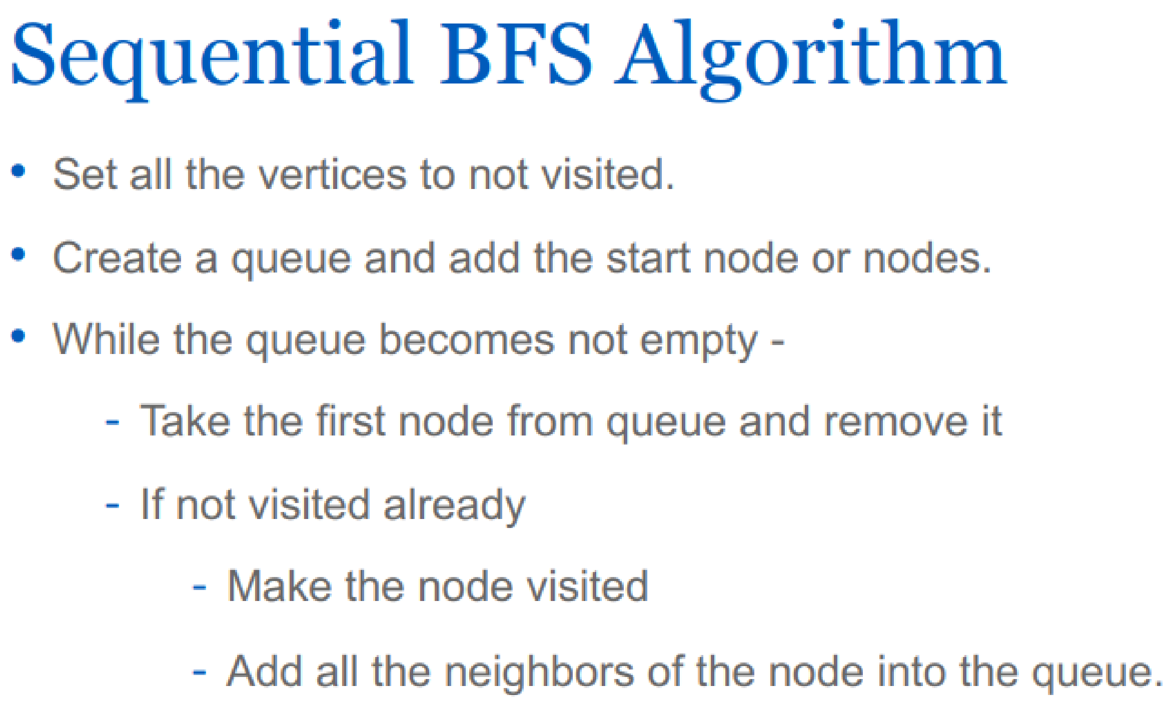
Overall Analysis:

OpenMP: OpenMP excelled in communication efficiency, maintaining consistently low or zero communication times. The computation times, while increasing with input size, remained reasonable.

MPI: MPI showcased efficient communication handling, with computation times comparable to OpenMP. However, some variability in communication times was observed.

II. Breadth First Search Traversal:

- **Serial Pseudocode:**



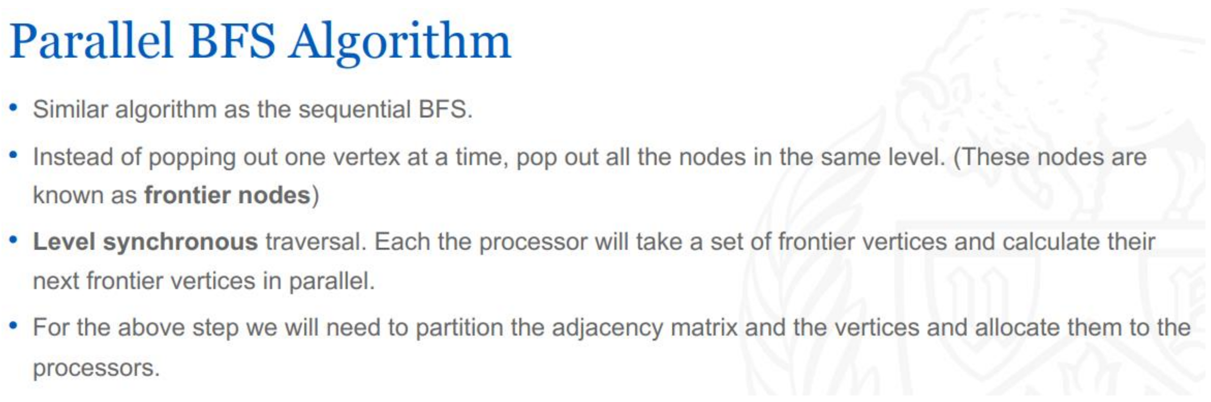
Sequential BFS Algorithm

- Set all the vertices to not visited.
- Create a queue and add the start node or nodes.
- While the queue becomes not empty -
    - Take the first node from queue and remove it
    - If not visited already
        - Make the node visited
        - Add all the neighbors of the node into the queue.

Reference:

https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Aditya-Nongmeikapam-Spring-2022.pdf (University at Buffalo: The State University of New York)

- **Parallel Pseudocode:**



Parallel BFS Algorithm

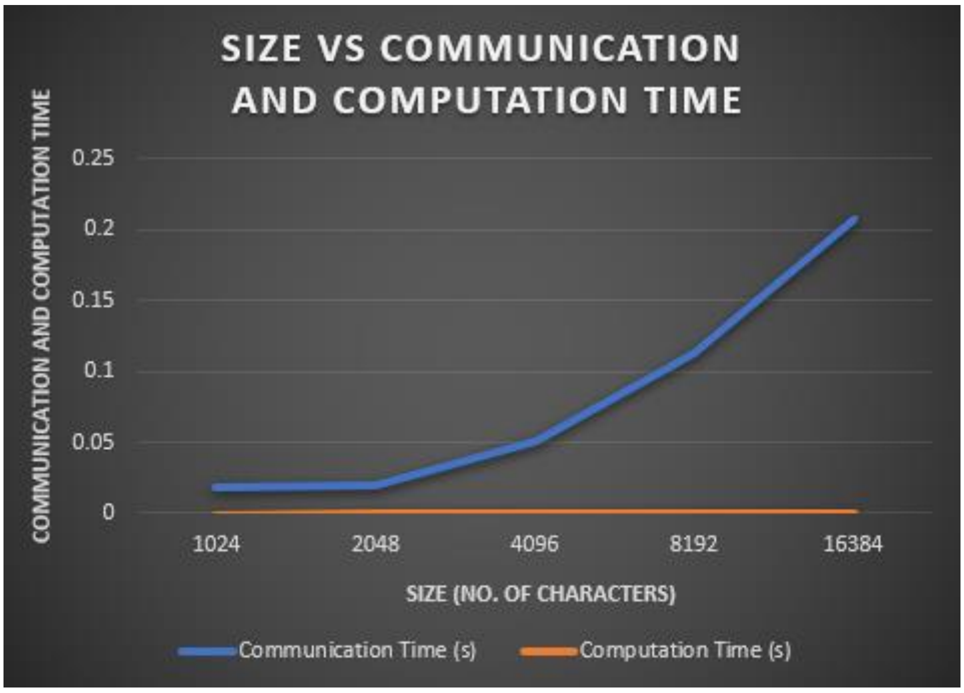
- Similar algorithm as the sequential BFS.
- Instead of popping out one vertex at a time, pop out all the nodes in the same level. (These nodes are known as **frontier nodes**)
- **Level synchronous** traversal. Each the processor will take a set of frontier vertices and calculate their next frontier vertices in parallel.
- For the above step we will need to partition the adjacency matrix and the vertices and allocate them to the processors.

Reference:

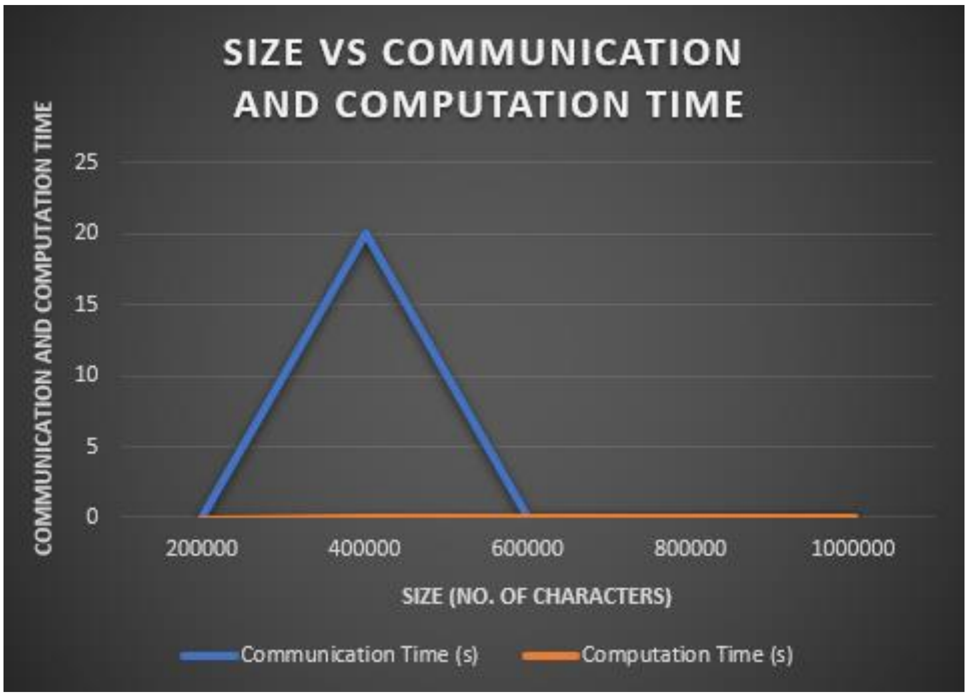
https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Aditya-Nongmeikapam-Spring-2022.pdf (University at Buffalo: The State University of New York)

Performance Comparison:

- **OpenMp:**



- **MPI:**



- **Overall Performance:**

1. **Communication Time:**

OpenMP: Gradual increase in communication time as the size grows, reaching 0.207686 seconds for a size of 16384 characters.

MPI: Varied communication times, with a notable spike to 20.00042487 seconds for a size of 400000 characters. Smaller sizes demonstrate significantly lower communication times.

2. Computation Time:

OpenMP: Consistently low computation times, with a slight fluctuation. Ranges from 0.00001804 seconds for 1024 characters to 0.00007131 seconds for 16384 characters.

MPI: Varied computation times, with the highest observed for a size of 1000000 characters (0.0126043 seconds). Smaller sizes show relatively lower computation times.

- ## Overall Analysis:

OpenMP: Demonstrates consistent and reasonable communication and computation times, maintaining efficiency across different input sizes.

MPI: Shows variability in communication times, with occasional spikes. Computation times are competitive but can escalate for larger inputs.

Conclusion:

In summary, the parallelization of Brute Force Pattern Matching and Breadth-First Search Traversal algorithms using OpenMP and MPI showcased commendable performance. OpenMP excelled in communication efficiency, maintaining zero communication overhead, while MPI demonstrated efficient communication with slight variability. Both exhibited reasonable scalability in computation time, with OpenMP showing a gradual increase and MPI presenting competitive times, especially for larger inputs.