Parallelization of Pattern Matching Z Algorithm

Divya Jyothsna Pulivarthi
Computational and Data Science
Indian Institute of Science, Bangalore, India
divyajp@iisc.ac.in

***Abstract*—Z-algorithm is an exact string matching problem that find all the occurrences of a pattern in the given text in a linear time. It constructs an array called Z-array, which stores the count of the matching prefix for each character in the text with respective to the pattern. And maintaining this array makes this algorithm simpler and efficient compared with other algorithms like KMP which have similar time and space complexities.**

In this research, we adopted two parallel methods to implement Z-algorithm to reduce the over execution of the program. And the experimental results show good speedups for the proposed algorithms which talks about the suitability of parallelizing Z-algorithm.

Keywords: Z algorithm, String-pattern matching, Pattern searching, Exact String Matching Algorithm, parallelism.

I. Introduction

Pattern matching is to find a text within a text ie. given a pattern P of length m and text T of length n, we have to find all the occurrence of P in T such that $m <= n$. A text is a large corpus of data consists of either alphabets, digits or special characters. And all alphabets of patterns must be matched to corresponding matched subsequence. Most commonly used datasets are English words, DNA, RNA or protein sequences.

The most oldest and initial algorithm for string searching is the Naive Brute-force algorithm. But it is the least efficient to check a specific pattern in a given string.

This algorithm works by matching with given text that is T = $t_0.t_1.t_2. ..t_{n-2}.t_{n-1}$ with given pattern that is P = $p_0.p_1.p_2...p_{m-1}$ in character by character. That means when t[0] is not equal to p[0] then it starts it matching with t[1] and p[0]. And this process is continued until any mismatch is found. After finding the complete match of the pattern it gives back the position of the pattern and continues its process of next character.

Several pattern matching algorithms were proposed later, such as KMP, Bayer-Moore, Rabin-Karp, and the Z-algorithm. The Z-algorithm stands out for its ease of implementation, as it is an alphabet-independent linear-time method. That is, we never had to assume that the alphabet size was finite or that we knew the alphabet ahead of time, a character comparison only determines whether the two characters match or mismatch. This makes this Z approach parallelizable easily in CPUs, GPUs and in hybrid as well.

The Z-algorithm efficiently computes the Z-array, which represents the longest common prefix between a pattern and all its suffixes, aiding in pattern matching tasks with linear time complexity. The Z-function for this string is an array of length n where the i-th element is equal to the greatest number of characters starting from the position i that coincide with the first characters of s. In other words, $z[i]$ is the length of the longest string that is, at the same time, a prefix of s and a prefix of the suffix of s starting at i.

Algorithm 1 Z-function

```
Inputs: z[n]
n ← s.size()
l ← 0, r ← 0
z[0] ← 0
i ← 1
for i < n do
  if i < r then
    z[i] = min(r - i, z[i - l])
  end if
  while i + z[i] < n and s[z[i]] == s[i + z[i]] do
    z[i] ← z[i] + 1
  end while
  if i + z[i] > r then
    l ← i
    r ← i + z[i]
  end if
end for
```

The concept involves merging the pattern P and the text T into a single string, denoted as "$P\$T$" where '$' serves as a unique delimiter that ensures it doesn't appear in either the pattern or the text. By constructing this concatenated string, we then proceed to generate the Z-array for it.

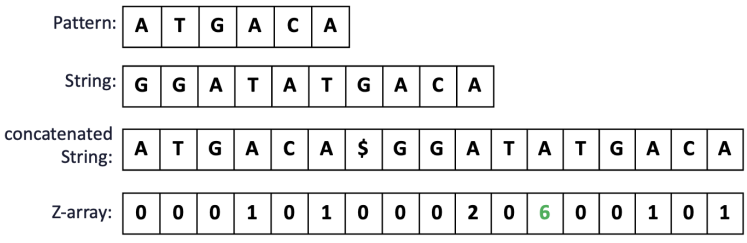

Fig. 1. Z-array calculation

The algorithm mentioned above operates in $O(n+m)$ time complexity for constructing the Z-array and for identifying the $z[i]$ that corresponds to the pattern length m, which is also $O(n+m)$. Consequently, regardless of the scenario—be it best, worst, or average— the Z-algorithm consistently runs in linear time $O(m+n)$.

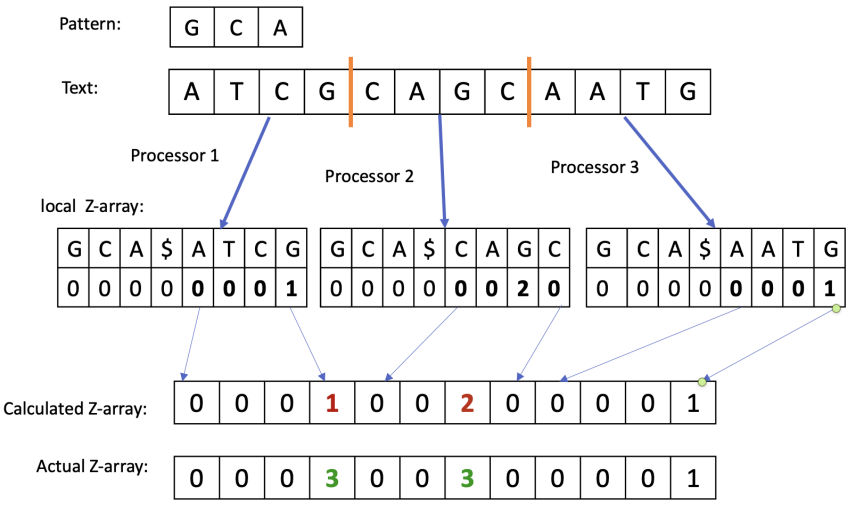

Fig. 2. Drawbacks when dividing the text T among the 3 processors

Parallelization often leverages the divide-and-conquer paradigm. This strategy decomposes the problem into smaller, independent sub problems. Each sub problem can be solved concurrently on separate processors, leading to faster computation. Finally, the individual solutions are combined together to obtain the final outcome.

But the main bottleneck of parallelizing the String-pattern matching is when dividing the text T among multiple processes, a challenge arises due to the inherent dependency between characters, particularly at the end of each divided text in each processor. This dependency often results in incorrect values being produced, undermining the accuracy and reliability of the parallelized algorithm, as in fig. 2. And we need to address these dependency issues and ensure the integrity of parallelized string matching processes.

In this research, we focus on the parallelization of the Z-algorithm addressing the bottlenecks using the MPI, a widely used standard for distributed memory parallel computing. Our goal is to minimize the algorithm's overall execution time and the space requirements while effectively employing the computational resources at hand by dividing the workload among numerous processing units to achieve faster pattern matching performance on distributed memory systems.

II. Related Work

A study conducted by F. Retkoceri et al. [1] compares the speed of various pattern searching algorithms (Naive, KMP, Rabin-Karp, Z-algorithm etc.) in C#, Java, and Python on different CPUs to analyze their time complexity. This paper shows that each algorithm demonstrates distinct effectiveness and uniqueness based on the specific configuration and increasing text size. Therefore, there isn't a one-size-fits-all optimal algorithm, and it's necessary to utilize different algorithms and strategies for different scenarios.

The paper by U. S. Alzoabi et al. [2] is based on parallelizing KMP algorithm, another Pattern matching algorithm, proposes a way to split the text up into chunks. To avoid missing matches at the end of chunks, each chunk borrows an extra characters of $(m-1)$ from the beginning of the next chunk. Thus, different processors work on these chunks at the same time.

Given that both the KMP and Z-algorithms share a common foundation in finding all instances of patterns within the text, we can adopt a similar strategy to compute the Z-array efficiently in Method 1.

III. Methodology

We introduced two different parallel algorithms that can help to levarge the Z-algorithm in parallel computing. Both methods achieve significant speedups compared to the traditional sequential approach. We will explain the details of each method in the following sections.

A. Method I:

The idea for the first algorithm is taken from the paper [2]. This approach investigates dividing the text into local chunks based on the processor size. To ensure pattern matches aren't missed at chunk boundaries, an additional prefix of the pattern length $(m-1)$ is appended from the subsequent processor's chunk. These chunks are then distributed amongst various processors for concurrent processing. After calculating the Z-array within each processor, we remove the additional $(m-1)$ values from the end of the Z-array to disregard the appended characters.

For example, Pattern = GCA with length (m) = 3, Text = ATCGCAGCAATG with length (n) = 12 and no. of processes (p) = 3.

In this case, each processor will receive chunks of characters with a length of $\frac{12}{3} = 4$, plus an additional (3 - 1) characters from the following processor. The last processor will receive $\frac{n}{p} + n\%p$ which equals 4 + 0 characters, without any additional characters appended, as it is the final processor

and doesn't have any dependencies.

Here, in Processor 0 and 1, the first 4 values are ignored, since it is pattern + $ special character and also last 2 values are ignored since they are added additionally.

On processors 0 and 1, we skip the first 4 values because they represent the pattern and a special character. We also ignore the last 2 values because they were added extra for dependency purposes.

1) Processor 0

:

G	C	A	$	A	T	C	G	C	A
0	0	0	0	0	0	0	3	0	0

The local z-array will be,

A	T	C	G
0	0	0	3

2) Processor 1

:

G	C	A	$	C	A	G	C	A	A
0	0	0	0	0	0	3	0	0	0

The local z-array will be,

C	A	G	C
0	0	3	0

3) Processor 2

:

G	C	A	$	A	A	T	G
0	0	0	0	0	0	0	1

This is the last processor, so the there is no additional characters added at the end, so only the initial pattern and special characters are skipped.

A	A	T	G
0	0	0	1

Finally gathering all the individual local Z-arrays from each processor to construct the ultimate global Z-array.

A	T	C	G	C	A	G	C	A	A	T	G
0	0	0	3	0	0	3	0	0	0	0	1

which matches the Actual Z-array result from Fig. 2.

The time and space complexity for this method, will be $O(m + \frac{n}{p} + (m-1))$. As the pattern length increases, this method won't be able to perform as expected due to the additional space required for each chunk to store extra characters of length $(m-1)$.

B. Method 2

In contrast to Method 1, this approach aims to reduce the additional space required by considering subsequent $(m-1)$ characters. In this method, we propose an alternative strategy to identify whether any suffix of the pattern matches the prefix of the chunk. It computes the total number of matches and determines the longest possible substring from the suffix of the pattern and the prefix of the chunk up to $(m-1)$ characters. These counts and the maximum suffix-prefix length are then stored for further analysis and optimization. We implement this method from Processor 1 till Processor p.

Parallelizing the Z-algorithm introduces a challenge when splitting the text into chunks for processing on different processors. There are three possibilities for pattern occurrence:

1) The pattern is entirely contained within a single chunk.
2) The pattern doesn't exist in any chunk
3) The pattern is divided between two adjacent chunks.

Method2 specifically addresses the scenario of a split pattern, i.e. the pattern is divided between two adjacent chunks. It allows us to identify this situation and take corrective action. If the pattern appears to be divided, we update the local Z-array of the previous processor and update the relevant $z[i]$ values based on specific conditions.

Initially, we create and initialize two variables: "maxSuffixLength" with an initial value of -1 and "repeatedCount" with an initial value of 0. These variables serve to indicate whether the pattern is distributed across processors. If the pattern is indeed divided, "maxSuffixLength" will store the longest common suffix of the pattern concerning the prefix of that chunk, while "repeatedCount" is used to track all possible instances of such matches.

If "maxSuffixLength" equals -1, it indicates that the pattern is not divided between the current and preceding processors. Consequently, there is no need to update any values.

Alternatively, if "maxSuffixLength" takes on a different value, we update the preceding processor's values as follows:

Algorithm 2 Condition to update preceding Z-array values if array is divided

Inputs: z-array of the preceding processor, m is the pattern length, chunk size is $\frac{n}{p}$
while maxSuffixLength > 0 && repeatedCount > 0 **do**
  currPos := chunkSize - (m - maxSuffixLength)
  if z[currPos] == m - maxSuffixLength **then**
    update z[currPos] := z[currPos] + maxSuffixLength
    repeatedCount := repeatedCount - 1
  end if
  maxSuffixLength := maxSuffixLength - 1
end while

To understand this method, We will illustrate two examples for more clarity.

1) Example 1: Consider,

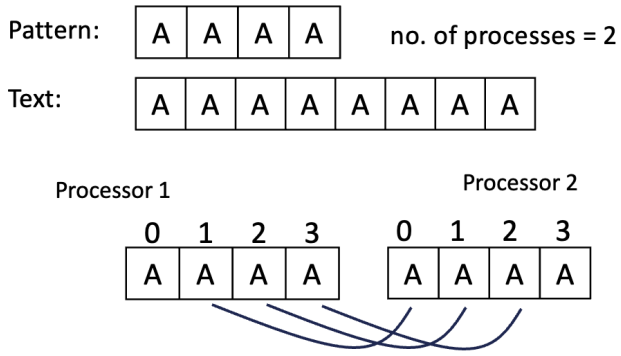

Fig. 3. Scenario 1: All the characters are equal in Pattern

In this case, we need to find a pattern that is repeated and the lines represent the indexes that are intertwined together incase of the update. First, we need to find local z-arrays of both.

Processor 1,

A	A	A	A	$	A	A	A	A
0	3	2	1	0	4	3	2	1

(Note: The initial condition is z[0] = 0, thus the first array value will be 0, and we start calculating Z-array from i = 1.)

Processor 2 will calculate its own local z-array in parallel.

Next, we'll implement the method to determine the maximum length of the prefix in the chunk that matches the suffix of the pattern from position 0 to (m-1) in Processor 2

For pos = 0,

Suffix of Pattern	Prefix of chunk
A	A

Both are matched, we update maxSuffixLength = 1 and repeatedCount = 1.

For pos = 1,

Suffix of Pattern	Prefix of chunk
AA	AA

Both are matched, we update maxSuffixLength = 2 and repeatedCount = 2.

For pos = 2,

Suffix of Pattern	Prefix of chunk
AAA	AAA

Both are matched, we update maxSuffixLength = 3 and repeatedCount = 3.

We reached the condition where pos $= m - 1$. So we stop the process here.

Now we have to update the Processor 1 based on these obtained values.

Initial local z-array is,

0	1	2	3
A	A	A	A
4	3	2	1

Now maxSuffixLength (3) > 0 and repeatedCount (3) > 0, so we enter the loop.

z[(4-(4-3) =3] == 4-3 (1==1), so we update z[3]=1+3, repeatedCount = 2 and maxSuffixLength = 2.

z[(4-(4-2) =2] == 4-2 (2==2), so we update z[2]=2+2=4, repeatedCount = 1 and maxSuffixLength = 1.

z[(4-(4-1)= 1] == 4-1 (3==3), so we update z[1]=3+1=4, repeatedCount = 0 and maxSuffixLength = 0.

Loop breaks here and the final Z-array in Processsor 1 will be updated as,

0	1	2	3
A	A	A	A
4	**4**	**4**	**4**

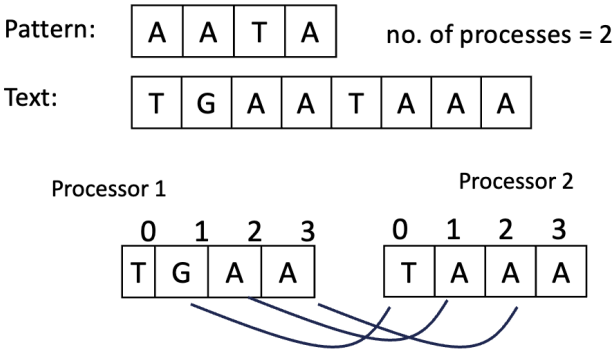

Fig. 4. Scenario 2: Non-repetitive pattern

2) Example 2: In this case, we have two consecutive A's in the pattern plus a non-repetitive pattern TA. First, we find local z-arrays of both.

Processor 1,

A	A	T	A	$	T	G	A	A
0	1	0	1	0	0	0	2	1

Processor 2 will calculate its own local z-array in parallel.

Next, we'll implement the method to determine the maximum length of the prefix in the chunk that matches the suffix of the pattern from position 0 to (m-1) in Processor 2.

For pos = 0,

Suffix of Pattern	Prefix of chunk
A	T

Not matched, we continue the process.

For pos = 1,

Suffix of Pattern	Prefix of chunk
TA	TA

Both are matched, we update maxSuffixLength = 2 and repeatedCount = 1.

For pos = 2,

Suffix of Pattern	Prefix of chunk
ATA	TAA

Not matched and since we reached the condition where pos $= m - 1$. So we stop the process here.

Now we have to update the Processor 1 based on these obtained values.

Initial local z-array is,

0	1	2	3
T	G	A	A
0	0	2	1

Now maxSuffixLength (2) $>$ 0 and repeatedCount (1) $>$ 0, so we enter the loop.
z[(4-(4-2)=2] == 4-2 (2==2), so we update z[2]=2+2, repeatedCount = 0 and maxSuffixLength = 1.

Loop breaks here and the final Z-array in Processor 1 will be updated as,

0	1	2	3
T	G	A	A
0	0	**4**	1

This method worked pretty well for different pattern sizes and text files. But this method fails when the length of the pattern exceeds the size of the chunk because it relies on comparing the pattern with the chunk. If the chunk size is smaller than the pattern size, it cannot accommodate the entire pattern for comparison, leading to failure.

IV. Experiments and Results

A. Experiment Setup

The algorithms are executed on the IOE Extreme Scale Cluster, which is equipped with an AMD EPYC 7302 16-Core Processor running on Linux 4.18.0 version. Open MPI version is 4.1.5 was used for the implementation. The DNA dataset is taken from Pizza & Chili Corpus [3] and the data used for testing the proposed algorithms includes repetitive and standard corpus where in one or more patterns are identifiable within the text.

The text files utilized for testing vary in sizes, 50, 100, and 200 MB.

B. Results

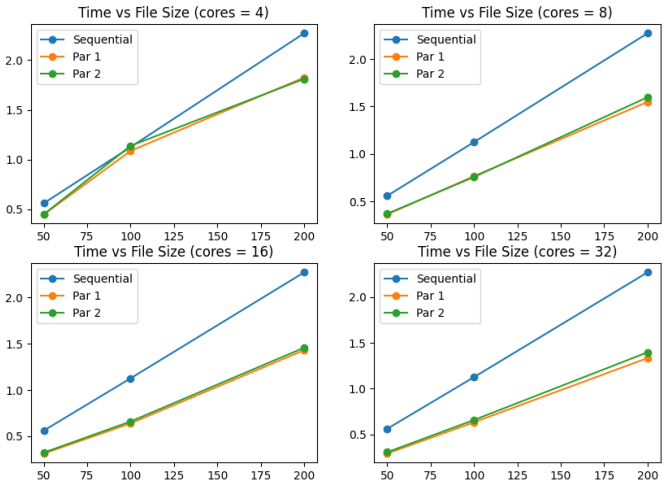

Fig. 5. File size vs Execution time for a constant pattern size 32

In the above graph, x-axis represents the file size used and y-axis represents the time take to execute the Z-algorithm. From the above graph, we can see both Methods 1 & 2 perform better in terms of execution time for a constant pattern length.

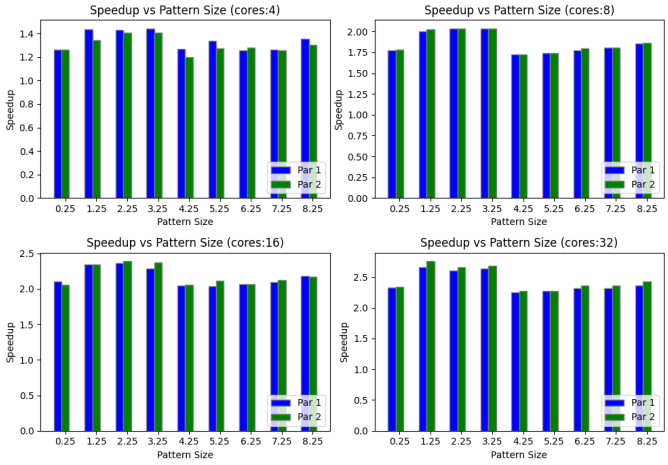

Fig. 6. Speedup of Method 1 vs Method2 by varying pattern size for each core

In the above graph, x-axis represents the increasing pattern size (from length 4 to 1024) and y-axis represents the Speedup achieved wrt to Sequential program.

Observing the data, it becomes evident that as the size of the pattern increases, Method 2 has better performance. Therefore, Method 2 offers improved space complexity over Method 1. Additionally, the plot demonstrates a significant increase in speedup as the number of cores increases from 4 to 32. This trend indicates that utilizing more cores can enhance the performance of the parallel Z-algorithm, up to a certain threshold. However, there reaches a saturation point where the communication overhead between processors outweighs the advantages of parallelization, particularly for very large patterns.

V. Conclusion and Future Work

In this paper, we introduced two parallel algorithms that show improved speedups. Experimental results show that these algorithms has achieved favorable speedup, and the results are consistent with theoretical analysis. Each method has its own advantages and disadvantages wrt to space requirement or depending on the pattern length. Method1 performs well irrespective of the pattern length but the memory usage can be high compared to Method2 and as the pattern length increases Method2 displayed a good performance with increase in number of cores as well. And in all the experiments both methods displayed the results accurately.

In the above second proposed algorithm, there is a limitation of our approach, particularly when the pattern size exceeds the local chunk size, leading the algorithm to produce incorrect results. As a future work, we can extend the research to address this issue of how to accommodate larger pattern sizes compared to the local size.

Additionally, as previously noted, for more scalability, instead of all the sub-task chunks being consolidated in the root process, an alternative approach could involve processes grouping in pairs and merging their arrays. Representatives of

these groups would then repeat this process iteratively until the entire Z-array is constructed within the root process.

VI. Acknowledgment

Some segments of the code were generated with the assistance of GitHub Copilot.

References

[1] F. Retkoceri, F. Idrizi, S. Ismaili, F. Imeri, and A. Memeti, "Analysis of pattern searching algorithms and their application," *International Journal of Recent Contributions from Engineering, Science IT (iJES)*, vol. 10, pp. 32–42, 12 2022.

[2] U. S. Alzoabi, N. M. Alosaimi, A. S. Bedaiwi, and A. M. Alabdullatif, "Parallelization of kmp string matching algorithm," in *2013 World Congress on Computer and Information Technology (WCCIT)*, 2013, pp. 1–3.

[3] "Pizza &chili corpus." [Online]. Available: https://pizzachili.dcc.uchile.cl/texts/dna/

[4] "Z-function and its calculation." [Online]. Available: https://cp-algorithms.com/string/z-function.html