Research Article

Improve the Performance of Search Processes in Large Linked Lists using Multi-threading and Caching Techniques

Rohul Amin[1], Md. Mehedi Rahman Rana[1], Md. Moazzem Hossain[1], S.M.A Sayem Prodhan[1], Abu Saleh Musa Miah[1]

1. Department of Computer Science and Engineering, Bangladesh Army University of Science and Technology (BAUST), Bangladesh

Efficiently searching on a large linked list is a critical challenge in various computational applications. Traditional methods grapple with the sequential organization and voluminous data, resulting in sluggish search operations. In response, this study introduces a novel approach leveraging multithreading for concurrent searches, facilitating parallel exploration of distinct segments within the linked list. Complementing this, we incorporate a caching mechanism to store frequently accessed elements, thereby optimizing RAM utilization during search processes. Through rigorous experimentation, our methodology showcases remarkable advancements in search efficiency and overall system performance compared to conventional techniques. These findings underscore the proposed framework's significance in revolutionizing large linked list exploration, offering promising avenues for enhancing computational operations across diverse domains.

Corresponding authors: Sarina Mansur, sarina.mansor@mmu.edu.my; Fahmid Al Farid, fahmid.farid@mmu.edu.my

1. Introduction

In the ever-evolving world of computer science, linked lists play a fundamental role in various applications, including personal storage, databases, search engines and artificial intelligence systems. Linked lists find extensive applications in various domains, making them a crucial data structure in modern computing[1]. They are the backbone of personal storage, enabling users to organize and

manage data efficiently. In databases, linked lists facilitate the storage and retrieval of records, contributing to smooth data operations[2]. Moreover, search engines heavily rely on linked lists to index and retrieve information from vast web databases[3]. Artificial intelligence systems also leverage linked lists to build complex data structures, efficiently storing and processing data during training and inference stages[3][4][5][6][7][8][9][10][11][12][13][14][15].

Searching is a fundamental operation within linked lists that enables the retrieval of specific data elements from the structure[16]. Efficient search algorithms enable seamless access to information in applications like databases, where quick response times are essential for a positive user experience[17]. Moreover, searching within linked lists in artificial intelligence systems facilitates efficient data processing during tasks like pattern recognition, decision-making and natural language processing.

The traditional linear search algorithm for linked lists has a time complexity of O(n)[18], which makes it inefficient for large-scale applications where the size of the dataset is significant[19]. As a result, a search algorithm for linked lists that can handle large datasets with minimal computational overhead and is more effective and scalable needs to be developed.

As computer systems continue to evolve rapidly, with innovations in multi-core processors, vectorized instructions and specialized hardware accelerators[20], the proposed algorithm is designed to take advantage of these advancements. It is optimized to efficiently utilize parallelism and leverage the full potential of modern hardware architectures. By doing so, the algorithm can exploit the available computational resources effectively, resulting in even faster search performance for large datasets. Furthermore, the algorithm's ability to handle large datasets with minimal computational overhead addresses one of the most critical challenges in contemporary computer science and data processing[21]. With the explosive growth of data in various domains, such as big data analytics, machine learning and artificial intelligence, the need for efficient search algorithms becomes paramount[22]. The proposed algorithm not only enhances the speed of searching within linked lists but also reduces the computational burden, making it well-suited for real-world, data-intensive applications.

To improve linked list-searching methods, this paper investigates the application of caching and parallel processing techniques. Caching is a method that stores frequently accessed data in a fast-access memory to reduce the time it takes to access it[23]. In contrast, parallel processing involves using multiple processing units or cores to perform tasks concurrently[24]. It begins by highlighting

the significance of linked lists as a widely used data structure in computer science, particularly for handling large datasets due to their dynamic memory allocation and efficient insertion and deletion capabilities[25]. However, the traditional sequential search algorithms for linked lists have limitations, especially for large-scale applications[19]. We propose using parallel processing and caching techniques to address this challenge to enhance search efficiency. The objective is to create an algorithm that is more effective and scalable. The proposed algorithm uses parallel processing to simultaneously search for the target data in each of the smaller chunks it divides the linked list into. They are cached in memory to speed up access to the chunks during subsequent searches. All the parallel search results are combined to determine whether the target data are in the linked list.

This paper evaluates the proposed algorithm's speed and scalability using various data sizes from[18]. The evaluation results establish the proposed algorithm's superiority over the traditional sequential search technique. It not only demonstrates its capacity to handle large datasets efficiently but also highlights its ability to maintain low computational overhead even when confronted with extensive data sizes. This is a critical achievement in linked list searching. Large-scale applications often involve massive datasets that can be time-consuming and resource-intensive to process using conventional methods[26]. By significantly outperforming the sequential search algorithm in terms of speed, the proposed algorithm paves the way for enhanced performance in a wide array of applications that rely on linked list searching[27].

In the first chapter, we discussed the liked list data structure, the linked list's significance, and its searching operation in real-world applications. The second chapter is about the related works in this field, highlighting the methodology and limitations of each research. Moving forward, the next chapter is about the methodology and implementation of the proposed algorithm. It consists of creating a dataset and applying the parallel search algorithm. The result analysis section represents and compares the experimental results with the linear search approach. Finally, the conclusion highlights the results and the future of the research conducted.

The major contributions of this study are given below:

i. Multi-threading is used for faster searching in large linked lists.
ii. Caching technique used for faster data processing in the linked list.
iii. Increased scalability due to the adaptation to different hardware by adjusting the number of threads used.

iv. Enabled load balancing optimization with configurable thread count and adjustable chunk size.

This research work is publicly available at GitHub Repository[28].

2. Literature Review

To address the scalability issue researchers have explored various optimization techniques including the use of parallel search algorithms.

2.1. Linked Lists and Search Algorithms

A linked list is a dynamic data structure comprising nodes where each containing a data element and a reference to the next node in the sequence. This structure is advantageous for operations such as insertion and deletion. These operations are performed with constant time complexity. However, searching for a specific element typically requires traversing the list that leads to a linear time complexity O(n). As the dataset grows the inefficiency of this approach becomes apparent and particularly in scenarios requiring frequent or time-sensitive searches.

o overcome the limitations of linear search various search algorithms have been developed. Binary search for example, offers a logarithmic time complexity O(log n), but it requires a sorted array and is not directly applicable to linked lists. Hashing is another approach that provides constant time complexity O(1) for search operations but introduces additional space overhead and does not maintain the order of elements.

Parallel search algorithms have emerged as a promising solution to the inefficiencies associated with traditional search methods in linked lists. By dividing the search task across multiple threads or processors, parallel search algorithms can significantly reduce the time required to locate a specific element in a large dataset. This approach leverages the capabilities of modern multi-core processors that enables concurrent processing of different segments of the linked list.

One of the key challenges in applying parallel search to linked lists is the dynamic nature of the data structure which complicates the division of nodes among threads. Despite these challenges several studies have demonstrated the effectiveness of parallel search algorithms in improving search efficiency.

2.2. Previous Work on Parallel Search in Linked Lists

Recent research has explored various strategies for implementing parallel search in linked lists. Yu et al. introduced a multiple learning backtracking search algorithm designed for estimating parameters in photovoltaic models, which incorporates parallelism to enhance efficiency[29]. Similarly, Dhulipala et al. developed theoretically efficient parallel graph algorithms that can be both fast and scalable, demonstrating the potential of parallel approaches in optimizing search tasks[30].

Fang et al. explored effective and efficient community search techniques over large heterogeneous information networks, employing parallel algorithms to handle the complexity of the task[31]. In another study, Fang et al. focused on optimizing query algorithms for large directed graphs, again highlighting the advantages of parallel processing[32]. Finally, Ezugwu proposed an enhanced symbiotic organisms search algorithm for scheduling in manufacturing, which utilizes parallelism to achieve near- optimal solutions with improved efficiency[33].

Building on this body of work, the current study aims to develop a novel parallel search algorithm specifically tailored for linked lists. The proposed algorithm will address the challenges associated with the dynamic nature of linked lists by effectively dividing the search task across multiple threads. By placing the search key at different positions within the linked list (beginning, middle and end), the study will evaluate the algorithm's performance in terms of time complexity and compare the results with those of traditional linear search methods. This research is expected to contribute to the field by providing an efficient alternative to existing search methods in linked lists, with potential applications in various domains that require the processing of large datasets.

3. Dataset

We created a new dataset according to the existing rules and protocol[18]. Datasets are generated of sizes 100k, 200k, 300k, and 500k of random numbers ranging from 0 to 9. The search key, which is 10, is placed as the first, middle and last element in the generated linked list. The following algorithm is used to generate the dataset.

Algorithm 1 Dataset Generation

Require: **N** = number of nodes
Ensure: Linked list with **N** nodes, each containing a random value between 0 to 9 and a search key

```
N ← number of nodes
Define a linked list structure
Make an initial pointer of the linked list as head
for i ← 1 to N − 1 do
    R ← random value between 0 and 9
    Create a new node
    Set the node value to R
    Add the node to the linked list
end for
Create a new node(contains search key)
Set the node value to 10
Add the node to the linked list (beginning, end or middle)
End
```

We used C++ built-in random function rand() is used to create a random dataset. A variable N indicates how many nodes should be created for a test-linked list. The first, middle and last nodes will have a known value, which will be searched for performance measurement. The nodes in between will have random values. The created data set will be like Figure 1.

After creating a linked list N-1 times with generated random numbers, a search key(integer 10) will be placed at the first, middle and end of the linked list for time complexity analysis. The time noted are then compared with the linear search results. A variable N indicates how many nodes should be created for a test-linked list. The first, middle and last nodes will have a known value, which will be searched for performance measurement. The nodes in between will have random values. The created data set will be like Figure 1.

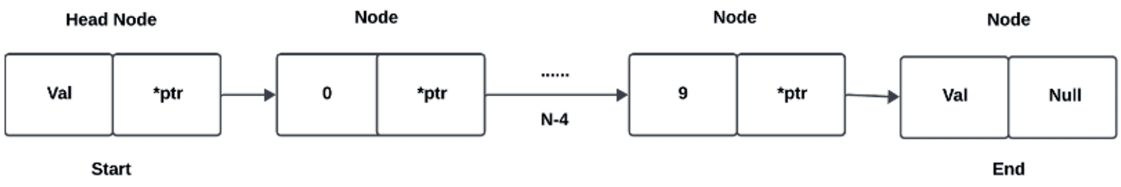

Figure 1. Linked List Structure.

doi.org/10.32388/XGNJ9L

4. Methodology

In the study, we proposed a search algorithm that can find the specific number from the large size of the link list. First, the datasets are passed to the parallel search function, and then keys are placed in the first, middle, and last sections of the linked list. After that, the search is performed to determine the specific goal with the information of time and space complexity. The observation of space and time of this operation is kept in a table and compared with the linear search based on this data.

4.1. Parallel Searching in a Linked List

The linked list utilizes the C++ struct idea[34], whereby struct Node *next designates the address of the linked list's next node, and int designates the integer data type that is included in the dataset. The generated random numbers are distributed to n - 1 nodes (n is the number of nodes that is taken from the user). Each random value is allocated to a node, which is then linked to the preceding node. The head of the node is if it is the first element in the dataset; if it is the last, the address portion of the node is given the value null. A search key that is not found in the dataset is added after the linked list with all the random values has been constructed. To do this study, the search is situated at the start, middle, and end. After the linked list and the search key is obtained, the parallel search algorithm is performed.

The proposed algorithm starts by initializing a variable to store the search key. If the search key is found in the cache, the result is returned immediately, otherwise the algorithm proceeds to the next step. The algorithm then initializes a pointer to the head of the linked list and traverses the list to determine chunk heads. The number of CPU threads is also initialized, and the size of each chunk is computed based on the number of threads and the total size of the list. Next, the algorithm initializes the required number of threads and assigns one chunk to each thread. Each thread then searches for the search key in its assigned chunk parallel. If the key is found, the data is cached, and the search key is returned. Finally, if the key is not found, the algorithm returns with a relative message.

Algorithm 2 Parallel Search

```
Start
Define searchKey variable
if searchKey is in cache then
    Return the result from cache
else
    Initialize ptr as head of the linked list
    Initialize the number of CPU threads, M
    Initialize chunk_size = (list_size + M − 1)/M
    Initialize M threads
    Initialize M chunk heads
    Assign one chunk to each thread
    while each thread is searching in its chunk do
        if searchKey == ptr → data then
            Cache the data and return searchKey
        end if
    end while
    Show "Not found"
end if
End
```

5. Experimental Evaluation

5.1. Environmental Setup

The experimental setups involve the use of an Intel i3 8th generation processor with 4 GB of RAM, an Intel i5 8th generation processor with 8 GB of RAM and an Intel i7 8th generation processor with 16 GB of RAM. Several readings were taken, and the results were averaged.

5.2. Performance Result

For the computations of the functions' execution times, the time.h module is used. This method is not very accurate in C++ programming language. There are variations in the complexity calculation of ±15% in all cases. Therefore, several readings(10-20) were taken, and the average was considered for the comparison.

Table 1 and table 2 show us the comparison between linear search and parallel search for the first search operation. The values are plotted in figure 3 and figure 4, respectively, for better understanding.

Figure 3 and figure 4 give us the graphical representation of the time complexity of linear search and parallel search for the first search operation. Parallel search, in this case, is slower than linear search because of the thread allocation. For both of the cases, the time complexity is linear. This is because the first node is at the first position of the list, and as soon as the pointer moves to the first node, the key is found. So, the time complexity would be the time that it takes for the pointer to move to the first node of the list. Although the differences are not that significant, we can say linear search performs better than parallel search for the first operation. This is the general case of the algorithm when the data is not at the cache memory. But in the case of being the data in the cache memory, the result is instantaneous. The time complexity when the cache hits is obviously constant and fast.

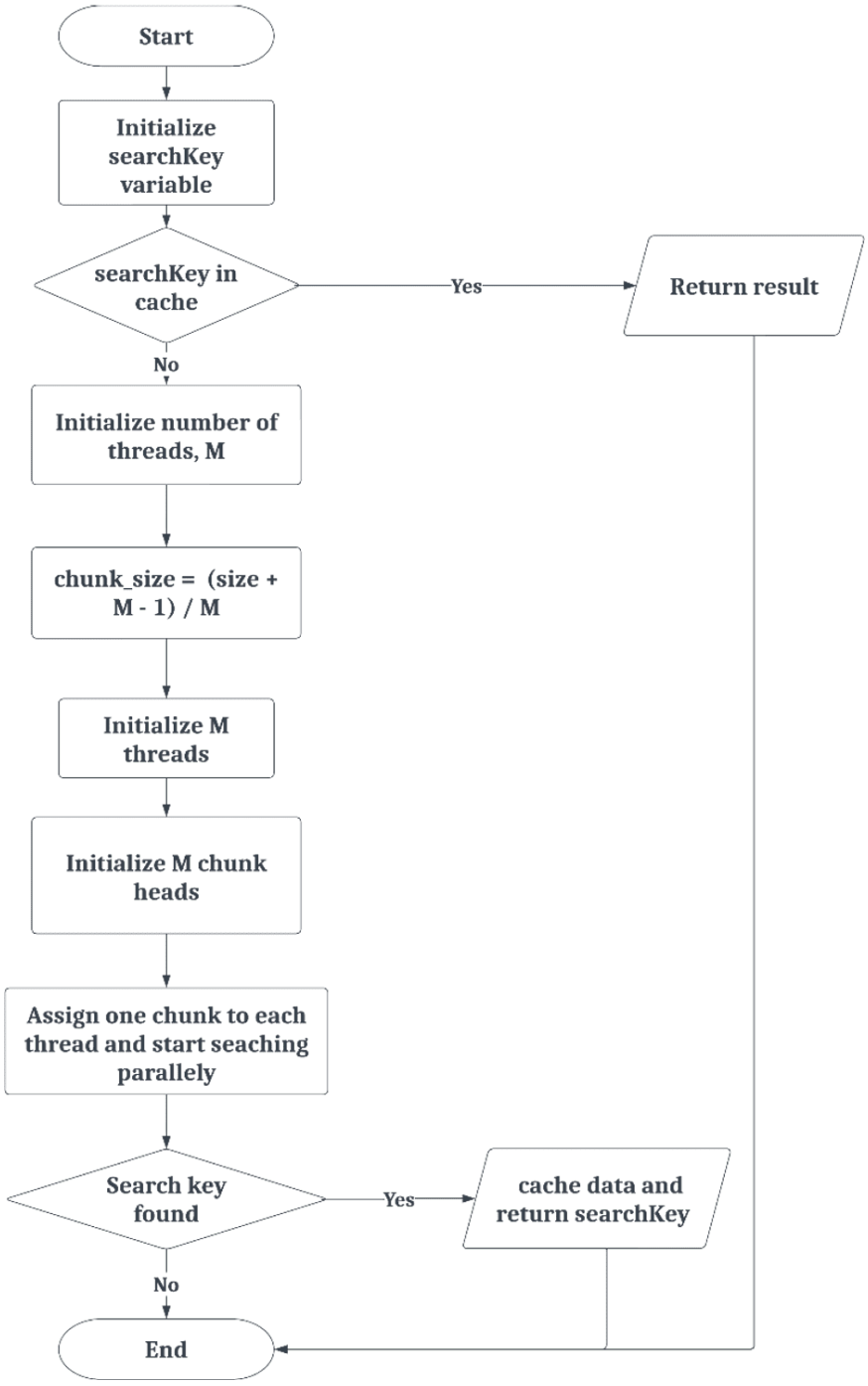

Figure 2. Flowchart of Parallel Search Algorithm.

doi.org/10.32388/XGNJ9L

Test Sample	Linear Search Time (in ms)	Parallel Search Time if Cache Miss (in ms)	Parallel Search Time if Cache Hit (in ms)
100k	3.5463	3.8186	0.0041
200k	3.6123	3.7675	0.0039
300k	3.6926	3.7685	0.0042
400k	3.6699	3.7654	0.0045
500k	3.5681	3.7321	0.0038

Table 1. Time Complexity for Search First Operation

Test Sample	Parallel compared to Linear
100k	96.669
200k	95.808
300k	97.985
400k	97.464
500k	95.607

Table 2. Calculation of the percentage of time necessary for the Search First Operation.

Table 3 and 4 show us the comparison between linear search and parallel search for search middle operation. The values are plotted in figure 5 and figure 6, respectively, for better understanding.

doi.org/10.32388/XGNJ9L

Test Sample	Linear Search Time (in ms)	Parallel Search Time if Cache Miss (in ms)	Parallel Search Time if Cache Hit (ms)
100k	5.3053	3.8188	0.0031
200k	6.5432	4.1261	0.0032
300k	7.4404	4.4498	0.0043
400k	8.4951	5.0673	0.0041
500k	9.4103	5.1318	0.0041

Table 3. Time complexity for Search Middle operation.

Test Sample	Parallel compared to Linear
100k	138.926
200k	158.581
300k	167.208
400k	167.645
500k	183.372

Table 4. Percentage time calculation for Search Middle Operation.

doi.org/10.32388/XGNJ9L

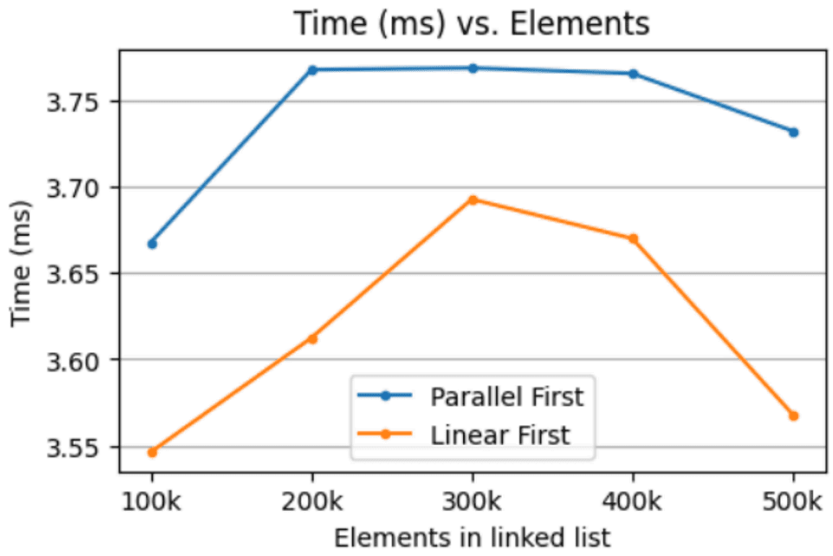

Figure 3. Graph Displaying the Time Complexity Involved in the Search First Operation.

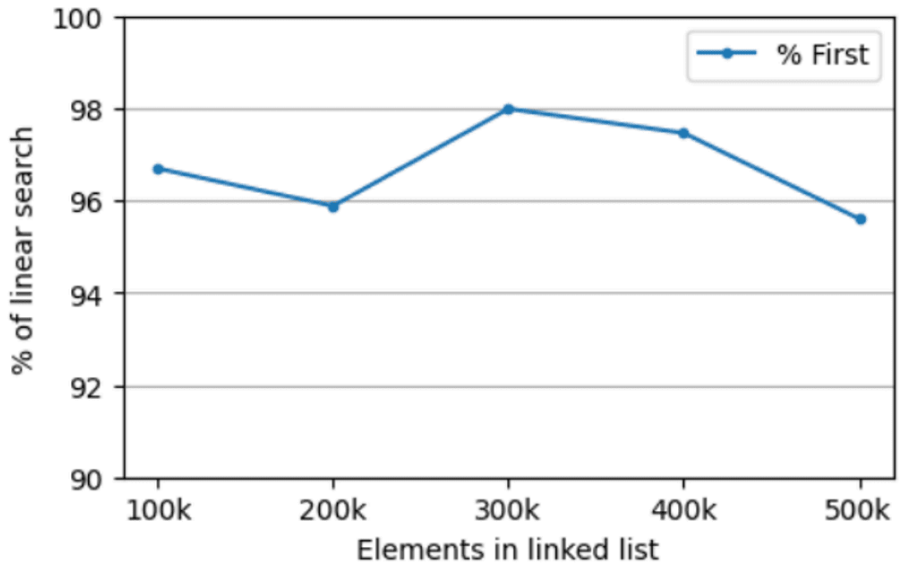

Figure 4. Percentage compared to Linear for Search First Operation.

Figure 5 and figure 6 gives us the graphical representation of time complexity of linear search and parallel search for search middle operation. In the case for searching the middle element in a linked list, parallel search algorithm outperforms linear search for every test sample. This is because of the

doi.org/10.32388/XGNJ9L

clustering effect. After dividing the linked list into clusters, the middle element does not remain the middle element for a specific cluster. In best cases the middle element might become the first element in a cluster. In that case, the time taken to find the middle element will be same as search first element. The cache hit case is the same as described before.

Table 5 and table 6 shows us the comparison between linear search and parallel search for search last operation. The values are plotted on figure 7 and figure 8 respectively for better understanding.

Figure 7 and figure 8 gives us the graphical representation of time complexity of linear search and parallel search for search last operation. Parallel search in this case is very efficient compared to linear search. As the data size increases the efficiency of parallel search increases. For 100k to 500k, the speed improves by ~ 35%to ~ 70% compared to linear search[7]. This shows that the parallel search algorithm becomes more and more efficient as the size of the dataset increases.

After analyzing the data, let's consider the following factors to better understand the time complexity:

Size of the dataset, n

Number of threads in CPU, num_threads

Time taken to search one chunk, chunk_search_time

Time taken for each node, K

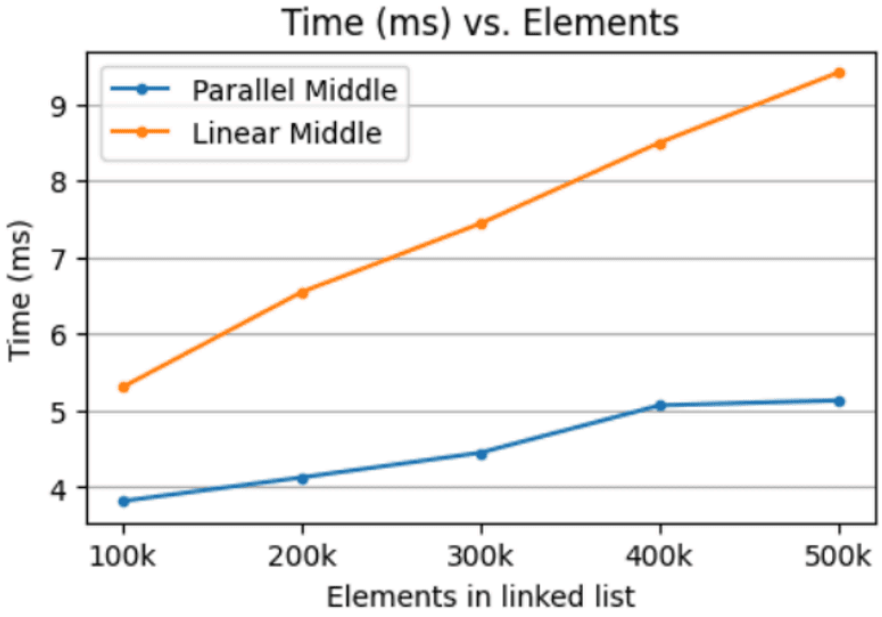

Figure 5. Time Complexity Graph for Search Middle Operation.

doi.org/10.32388/XGNJ9L

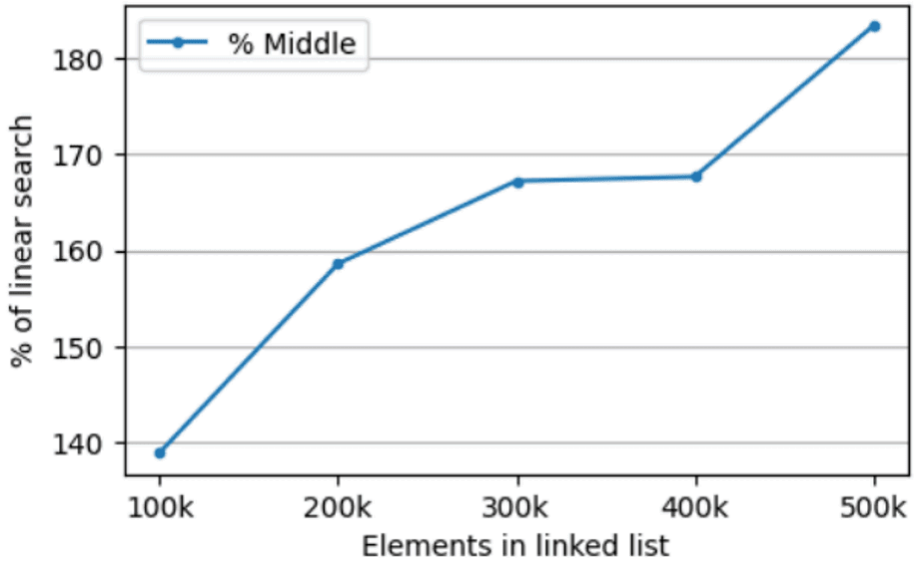

Figure 6. Percentage compared to Linear for Search Middle Operation.

Test Sample	Linear Search Time (in ms)	Parallel Search Time if Cache Miss (in ms)	Parallel Search Time if Cache Hit (in ms)
100k	6.7386	4.901	0.0032
200k	8.6410	6.209	0.0041
300k	10.324	7.118	0.0044
400k	11.937	7.501	0.0046
500k	13.548	8.019	0.0042

Table 5. Time complexity for Search Last operation.

doi.org/10.32388/XGNJ9L

Test Sample	Parallel compared to Linear
100k	137.494
200k	139.169
300k	145.041
400k	158.337
500k	168.895

Table 6. Calculation of the time percentage for the search last operation.

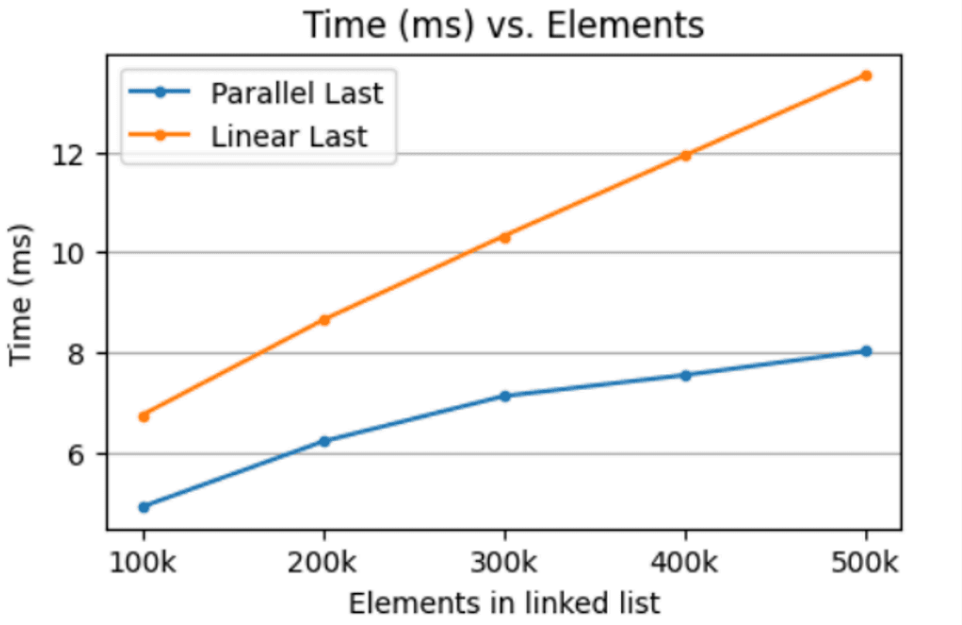

Figure 7. Time Complexity Graph for Search Last Operation.

doi.org/10.32388/XGNJ9L

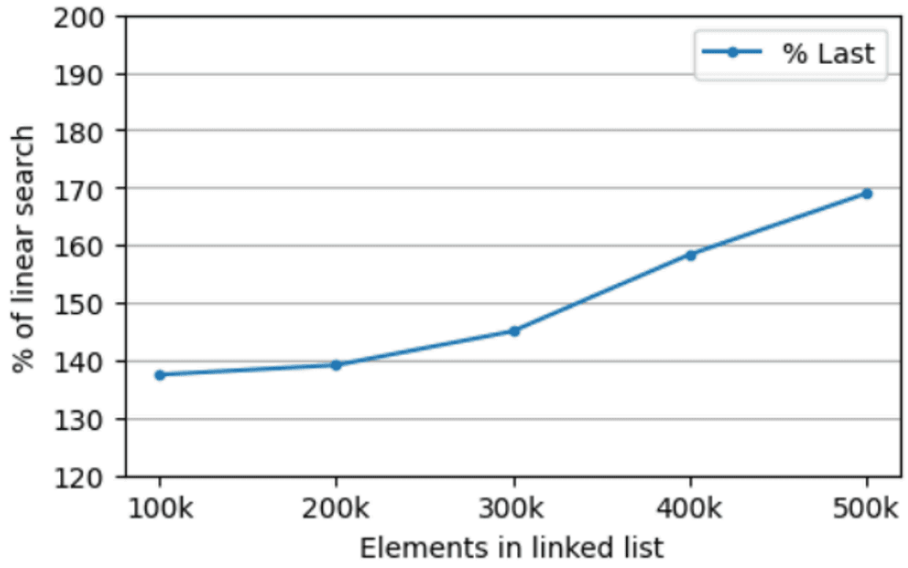

Figure 8. Percentage compared to Linear for Search Last Operation.

By using these, we will be able to determine the time complexity of the parallel search method for the best case, the average case, and the worst case respectively Best Case: In the best-case scenario, the key is found in the first node of the list, so the algorithm will have a time complexity of O(K) as only one node is traversed. As K is constant, we can also say the complexity if O(1).

Average Case: In the average case, the key is found in the middle of the list. For different sizes of datasets, the middle element may be in a different position of a chunk. Let the average position of the search key be in the middle p. So, the time required to find a key at p would be $O(K * p)$.

Worst Case: In the worst case, the search key would be at the last position of the last chunk. So, the time taken to access that would be equal to the time to traverse one chunk. This can be expressed as O(chunk_search_time). Note that the size of each chunk depends on the size of the dataset and the number of threads.

chunk_search_time = n/(num_threads ∗ K)

Compared to linear search, this algorithm has a better average and worst-case time com- plexity as it divides the list into chunks and assigns each chunk to a separate thread. This improves the parallelism of the algorithm, which can help reduce the search time.

doi.org/10.32388/XGNJ9L

In case of a cache hit, the Best, Average and Worst case complexity is O(1). This further improves the efficiency of the algorithm.

6. Conclusion

This paper presents a strategy for improving search efficiency in large linked lists using multi-threading and caching techniques. The research reveals improvements in time spent searching and overall system performance by efficiently storing frequently used elements in a cache and scanning multiple sections simultaneously. The scalability analysis and trade-offs between multi-threading and caching provide practical implementation ideas, making the methodology potentially useful for optimizing search operations in computational applications dealing with large linked lists.

Future work on the proposed methodology includes optimizing search efficiency, fine- tuning cache replacement policies and dynamically adapting data distribution strategies. Conducting extensive testing and benchmarking against other search algorithms and data structures will validate the algorithm's practical applicability. Integrating the parallel search algorithm into real-world applications can provide valuable insights into its scalability and responsiveness, ultimately improving its overall performance.

References

1. ^ *Hardiyana B, Fadilah L, Effendi D. Application of linked list algorithm based on multimedia. In Procee dings of the IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2020, Vol. 879, p. 012087.*
2. ^ *Osama M, Wijs A. Verifying Linked List Safety Properties in AWS C99 Package with CBMC. SAT COMPE TITION 2022, p. 72.*
3. a, b *Wikipedia. Accessed on 2023-01.*
4. ^ *Hossain MM, Chowdhury ZR, Akib SMRH, Ahmed MS, Hossain MM, Miah ASM. (2023). Crime Text Cla ssification and Drug Modeling from Bengali News Articles: A Transformer Network-Based Deep Learnin g Approach. In 2023 26th International Conference on Computer and Information Technology (ICCIT) (pp. 1-6). IEEE.*
5. ^ *Hossain MM, Noman AS, Begum MM, Warka WA, Hossain MM, Miah ASM. (2023). Exploring Banglade sh's Soil Moisture Dynamics via Multispectral Remote Sensing Satellite Image. European Journal of Envi*

ronment and Earth Sciences, 4(5), 10-16.

6. ^*Rahim MA, Farid FA, Miah ASM, Puza AK, Alam MN, Hossain MN, Karim HA. (2024). An Enhanced Hybrid Model Based on CNN and BiLSTM for Identifying Individuals via Handwriting Analysis. CMES-Computer Modeling in Engineering and Sciences, 140(2).*
7. ^a, ^b*Ali MS, Mahmud J, Shahriar SMF, Rahmatullah S, Miah ASM. (2022). Potential Disease Detection Using Naive Bayes and Random Forest Approach. BAUST Journal.*
8. ^*Zobaed T, Ahmed SRA, Miah ASM, Binta SM, Ahmed MRA, Rashid M. (2020). Real-time sleep onset detection from single channel EEG signal using block sample entropy. In IOP Conference Series: Materials Science and Engineering (Vol. 928, No. 3, p. 032021). IOP Publishing.*
9. ^*Kibria KA, Noman AS, Hossain MA, Bulbul MSI, Rashid MM, Miah ASM. (2020). Creation of a Cost-Efficient and Effective Personal Assistant Robot using Arduino Machine Learning Algorithm. In 2020 IEEE Region 10 Symposium (TENSYMP) (pp. 477-482). IEEE.*
10. ^*Rahman MR, Hossain MT, Nawal N, Sujon MS, Miah ASM, Rashid MM. (2020). A Comparative Review of Detecting Alzheimer's Disease Using Various Methodologies. BAUST Journal.*
11. ^*Miah ASM, Islam MR, Molla MKI. (2019). EEG classification for MI-BCI using CSP with averaging covariance matrices: An experimental study. In 2019 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2) (pp. 1-5). IEEE.*
12. ^*Miah ASM, Ahmed SRA, Ahmed MR, Bayat O, Duru AD, Molla MKI. (2019). Motor-Imagery BCI task classification using Riemannian geometry and averaging with mean absolute deviation. In 2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT) (pp. 1-7). IEEE.*
13. ^*Miah ASM, Islam MR, Molla MKI. (2017). Motor imagery classification using subband tangent space mapping. In 2017 20th International Conference of Computer and Information Technology (ICCIT) (pp. 1-5). IEEE.*
14. ^*Tusher MMR, Farid FA, Al-Hasan M, Miah ASM, Rinky SR, Jim MH, Mansor S, Rahim MA, Karim HA. (2024). Development of a Lightweight Model for Handwritten Dataset Recognition: Bangladeshi City Names in Bangla Script. Computers, Materials & Continua, 80(2), 2633-2656. Available at: http://www.techscience.com/cmc/v80n2/57586*
15. ^*Tusher MMR, Farid FA, Kafi HM, Miah ASM, Rinky SR, Islam M, Rahim MA, Mansor S, Karim HA. BanTrafficNet: Bangladeshi Traffic Sign Recognition Using A Lightweight Deep Learning Approach, Preprint.*

16. ^Koganti H, Yijie H. Searching in a Sorted Linked List. In Proceedings of the 2018 International Confere nce on Information Technology (ICIT). IEEE, 2018, pp. 120–125.
17. ^Mar Z, Oo KK. An Improvement of Apriori Mining Algorithm using Linked List Based Hash Table. In Pr oceedings of the 2020 International Conference on Advanced Information Technologies (ICAIT), 2020, pp. 165–169. doi:10.1109/ICAIT51105.2020.9261804.
18. a, b, cLokeshwar B, Zaid MM, Naveen S, Venkatesh J, Sravya L. Analysis of Time and Space Complexity of Array, Linked List and Linked Array (hybrid) in Linear Search Operation, 2022. doi:10.1109/ICDSAAI554 33.2022.10028872.
19. a, bOussous A, Benjelloun FZ. A comparative study of different search and indexing tools for big data. Jor danian Journal of Computers and Information Technology 2022, 8.
20. ^Long S, Fan X, Li C, Liu Y, Fan S, Guo XW, Yang C. VecDualSPHysics: A vectorized implementation of S moothed Particle Hydrodynamics method for simulating fluid flows on multi-core processors. Journal o f Computational Physics 2022, 463, 111234.
21. ^Brickley D, Burgess M, Noy N. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. In Proceedings of the The World Wide Web Conference, 2019, pp. 1365–1375.
22. ^Anand A, Trivedi NK, Abdul Wassay M, AlSaud Y, Maheshwari S. Application and Uses of Big Data Anal ytics in Different Domain. In Proceedings of the Machine Intelligence and Data Science Applications; Ska la V, Singh TP, Choudhury T, Tomar R, Abul Bashar M, Eds., Singapore, 2022; pp. 481–500.
23. ^Sancheti R, Ramesh K. Cache Design and Optimization Techniques. Research and Applications: Emergi ng Technologies 2022, 3.
24. ^Tousimojarad A, Vanderbauwhede W. Efficient Parallel Linked List Processing. Parallel Computing: On the Road to Exascale 2016, 27, 295.
25. ^Astrakhantseva IA, Astrakhantsev RG, Mitin AV. Randomized C/C++ dynamic memory allocator. In Pro ceedings of the journal of physics: conference series. IOP publishing, 2021, Vol. 2001, p. 012006.
26. ^ur Rehman MH, Yaqoob I, Salah K, Imran M, Jayaraman PP, Perera C. The role of big data analytics in industrial Internet of Things. Future Generation Computer Systems 2019, 99, 247–259.
27. ^Puntambekar A. Data Structures; UNICORN Publishing Group: Pune, 2020.
28. ^Amin R. labidz - Overview — github.com. https://github.com/labidz.
29. ^Yu K, Liang J, Qu B, Cheng Z, Wang H. Multiple learning backtracking search algorithm for estimating parameters of photovoltaic models. Applied Energy 2018, 226, 408–422. doi:10.1016/j.apenergy.2018.0 6.010.

30. ^*Dhulipala L, Blelloch GE, Shun J. Theoretically efficient parallel graph algorithms can be fast and scala ble. ACM Transactions on Parallel Computing (TOPC) 2021, 8, 1–70.*

31. ^*Fang Y, Yang Y, Zhang W, Lin X, Cao X. "Effective and efficient community search over large heterogen eous information networks". Proc. VLDB Endow. 2020; 13: 854–867. doi:10.14778/3380750.3380756.*

32. ^*Fang Y, Wang Z, Cheng R, Wang H, Hu J. "Effective and Efficient Community Search Over Large Directe d Graphs". IEEE Transactions on Knowledge and Data Engineering 2019; 31: 2093–2107. doi:10.1109/T KDE.2018.2872982.*

33. ^*Ezugwu AE. "Enhanced symbiotic organisms search algorithm for unrelated parallel machines manuf acturing scheduling with setup times". Knowledge-Based Systems 2019; 172: 15–32. doi:10.1016/j.knosy s.2019.02.005.*

34. ^*Parker A. Algorithms and Data Structures in C++; Routledge: New York, 2018. doi:10.1201/97813151371 48.*

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.