Statistics and Applications {ISSN 2454-7395 (online)}
Volume 23, No. 2, 2025 (New Series), pp 169–179
http://www.ssca.org.in/journal

VTPC: A Novel Virtual Tree Approach for Efficient Population Count in Binary Sequences

Samarth Godara, Prakash Kumar, Deepak Singh and Raju Kumar

ICAR-Indian Agricultural Statistics Research Institute, New Delhi-110012, India

Received: 04 October 2024; Revised: 03 December 2024; Accepted: 06 December 2024

Abstract

The task of counting the number of ones in a bit sequence, known as the "Population Count," is essential in various domains such as cryptography, database information retrieval, and machine learning. Traditional methods, whether hardware or software-based, often require direct interaction with the binary representation of data, which can be computationally intensive. This paper introduces a new technique, the Virtual Tree for Population Count Method (VTPC), which can calculate the population count of a number by leveraging a virtual tree structure without converting the number to its binary form. This method significantly reduces time complexity compared to existing algorithms. Our approach is evaluated against traditional methods, demonstrating its efficiency and accuracy in computing the population count. By eliminating the need to convert numbers to their binary form, VTPC streamlines processes and can lead to faster, more scalable implementations in various high-performance computing environments.

Key words: Population count; Virtual tree method; Bit counting; Cryptography; Algorithm efficiency.

AMS Subject Classifications: 62K05, 05B05

1. Introduction

Bit-counting, a crucial computer operation, involves determining the number of '1's in a bit sequence. This operation is fundamental for various applications that utilize bitsets or bitmaps, necessitating frequent population counts. The term "population count" (often abbreviated as 'popcount') also refers to the cardinality of a bitset, indicating the number of '1' bits representing elements in the set. The efficiency of bit-counting operations significantly impacts the performance of file systems, databases, machine learning algorithms, search engines, and information security systems.

In cryptography, population count functions are employed for randomness tests and generating pseudo-random permutations, aiding in the identification of duplicated web pages. Additionally, they are widely used in bioinformatics, ecology, chemistry, and various other

Corresponding Author: Prakash Kumar
Email: prakash289111@gmail.com

fields. As noted by Muła *et al.* (2018), recent developments have seen population count instructions integrated into fast sorting algorithms by researchers like Gueron and Krasnov.

Population count techniques are broadly classified into two categories: hardware implementations and software-based techniques. Hardware implementations involve specialized circuitry to execute popcount functions efficiently, whereas software-based techniques encompass a range of algorithms that do not require additional hardware. This paper focuses on proposing a novel software-based technique for bit-counting, which predicts the population count of a number directly from its decimal representation without converting it into binary form. The proposed method aims to improve the time complexity of population count operations compared to existing algorithms. The following is a detailed overview of some prominent existing methods for population count.

1. Naïve Method: As shown in Rosen (2019), the naïve approach involves examining each bit in a binary sequence individually. This method iterates through all bits of an integer, checking if each bit is set, and increments a count if it is. For a 32-bit integer, this approach may require up to 32 iterations in the worst case, making it relatively inefficient. The time complexity is $O(M)$, where M is the number of bits in the integer.

2. Table Lookup: A more efficient method involves precomputing the population counts for all possible byte values (0 to 255) and storing them in a lookup table, as explained by Fateman (1989). Each byte of the input number is then checked against this table, and the results are summed. This method reduces the number of required operations significantly. For larger chunks of data, a 16-bit or even larger lookup table can be used, though this increases memory usage. The time complexity is $O(M/L)$, where L is the number of bits processed in each lookup operation.

3. Divide and Conquer Strategy: Knuth (2014) introduced the divide and conquer method, that breaks down the problem into smaller parts, summing bits in a hierarchical manner. Initially, pairs of bits are summed, then these results are summed in pairs, and so on. This technique leverages the hierarchical nature of binary numbers and can perform the population count in $log_2(M)$ steps, where M is the number of bits. This method is efficient for systems that can perform parallel operations on bits.

4. Carry-Save Adder (CSA): The CSA technique uses a bitwise parallel carry-save adder, a hardware construct often used in binary multipliers, as shown by Muła *et al.* (2018). This approach processes elements in groups, reducing three words to two and then applying the population count operation to these two words. The final population count is obtained by summing the outputs. This method is particularly useful for handling large arrays of data efficiently.

5. Population Count Operations on Logical Results: In many practical applications, the population count is performed based on the result of logical operations, such as AND, OR, or XOR between two bitsets, as shown by Marton *et al.* (2017). This involves loading input bytes, generating temporary words through logical operations, and then determining their population count. These operations can be performed by either software methods or hardware implementations, depending on the specific application requirements.

Moreover, Polychroniou *et al.* (2015) noted that multiple recent advancements include vectorized algorithms that leverage modern microprocessor capabilities to process multiple bits in parallel These algorithms can significantly enhance performance by utilizing SIMD (Single Instruction, Multiple Data) instructions available in many modern processors. Vectorized methods are especially effective for large datasets and high-performance computing applications.

Furthermore, many solutions apply hardware implementations of population count functions involving specialized circuitry designed to perform bit-counting efficiently, as shown by Klarqvist (2021). Many modern processors include dedicated instructions for population count, such as the POPCNT instruction in x86 architectures. These implementations offer high-speed bit-counting by leveraging hardware-level parallelism and optimized circuitry.

These existing techniques highlight the diversity of approaches to the population count problem, each suited to different scenarios and performance requirements. The proposed Virtual Tree for Population Count Method (VTPC) aims to offer a novel software-based solution that improves upon the time complexity and practical implementation challenges associated with these traditional methods.

2. Virtual tree for population count method (VTPC)

Algorithm 1 Population count algorithm using VTPC

Require: n (a positive integer in decimal number system)
Ensure: m (the number of 1's present in the binary representation of n)
1: mask_size $\leftarrow 4$
2: mask $\leftarrow [0, 1, 1, 2]$
3: $m \leftarrow 0$
4: $i \leftarrow 0$
5: $d \leftarrow \lceil \log_{\text{mask\_size}}(n) \rceil$
6: $e \leftarrow \text{mask\_size}^d$
7: $r \leftarrow n$
8: **while** $d \geq 0$ **do**
9: $\quad i \leftarrow \lfloor (r/e) \times \text{mask\_size} \rfloor$
10: $\quad r \leftarrow r - \left(\frac{e}{\text{mask\_size}} \times i \right)$
11: $\quad e \leftarrow e/\text{mask\_size}$
12: $\quad m \leftarrow m + \text{mask}[i]$
13: $\quad d \leftarrow d - 1$
14: **end while**
15: **return** m

The VTPC method calculates the population count of a given number using a search operation on a virtual tree structure as given in Algorithm 1. Here, each node within this tree is represented by a vector of integers. The root node is termed the "mask vector," and each element within this vector corresponds to a child node. The mask vector has a length of 2^L, where L is any natural number. The elements of the mask vector are the first 2^L numbers in the sequence of population counts.

For example, using a mask vector of $\langle 0, 1, 1, 2 \rangle$, which represents the first four numbers

in the population count sequence of integers (population count of numbers 0, 1, 2 and 3), each child node is generated by adding the corresponding integer from the parent node to every integer in the mask vector. This resultant vector forms the child node. The leaf nodes of this virtual tree represent the population counts of the integers.

The algorithm uses a variable, i, to track and generate the necessary child node by skipping the appropriate number of elements to reach the desired position in the sequence. The algorithm operates in two main phases: selection and generation. During the selection phase, items from the current node are chosen for expansion. In the generation phase, the next node is generated by adding the selected element to the mask vector. This process iterates $\log(n) + 1$ times, where s is the length of the mask vector, and n is the input number. With each iteration, a new child node is generated, replacing its parent node, and the cycle continues until a leaf node is reached.

3. Case studies and simulation setup

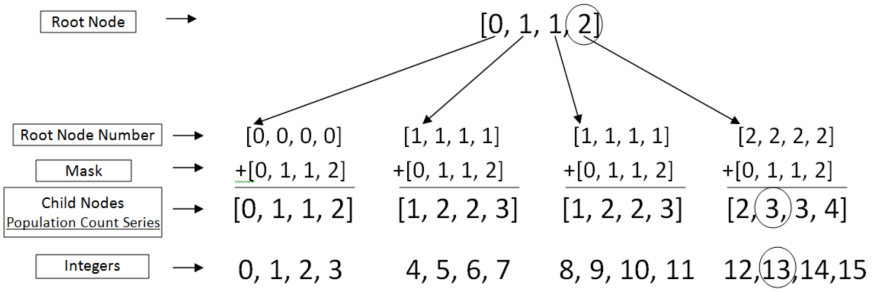

Figure 1: Complete virtual tree of depth 1 generated to compute population count of number 13. Integers of leaf nodes represent the population count series. The bottom most series in the figure represents the corresponding natural numbers from 0 to 15.

To illustrate the VTPC Method, we consider a case study to examine how the algorithm computes the population count for the number 13. Figure 1 presents the virtual tree for this problem. In the first iteration, the algorithm selects the fourth integer from the root node, which is 2. The child node is then generated by adding 2 to each integer in the mask vector, resulting in the vector $\langle 2, 3, 3, 4\rangle$. Although Figure 1 displays all potential leaf nodes for demonstration purposes, the algorithm only generates the required child node at each step. During the second iteration, the algorithm selects the second integer from the current node $\langle 2, 3, 3, 4\rangle$, which is 3. In the final iteration, the algorithm recognizes that the current node is a leaf node, indicating the end of the process. Consequently, the algorithm terminates its execution. The whole calculation process can be described as follows:

1. Initialization:

   (a) The mask vector is $\langle 0, 1, 1, 2\rangle$.

   (b) The input number n is 13.

   (c) Initialize m (the population count) to 0.

(d) Calculate d as $\lceil \log_4(13) \rceil$, which determines the depth or number of iterations. For $n = 13$, d is 2 because $\log_4(13)$ is approximately 1.85, and the ceiling function rounds it up to 2.

(e) Calculate e as $4^d = 4^2 = 16$.

(f) Initialize r (remainder) to 13.

2. First Iteration:

(a) Calculate the index i using the formula: $i = \left\lfloor \frac{r}{e} \times 4 \right\rfloor$.

(b) Substituting the values: $i = \left\lfloor \frac{13}{16} \times 4 \right\rfloor$.

(c) Simplifying further: $i = \lfloor 0.8125 \times 4 \rfloor = \lfloor 3.25 \rfloor = 3$.

3. Selection: According to the mask vector $\langle 0, 1, 1, 2 \rangle$, the element at index 3 (0-based index) is 2.

4. Updating Values:

(a) Update r by subtracting $\left(\frac{e}{4} \times i\right)$ from it: $r = 13 - \left(\frac{16}{4} \times 3\right) = 13 - 12 = 1$.

(b) Update e by dividing it by 4: $e = \frac{16}{4} = 4$.

(c) Update m by adding the selected mask element '2' to it: $m = 0 + 2 = 2$.

(d) Decrease d by 1: $d = 2 - 1 = 1$.

5. Second Iteration: In the second iteration, again the value of 'm' is updated using the above equations, and the resultant value '13' is outputted.

The selection of the fourth integer '2' in the first iteration is driven by the calculated index i. This calculation ensures that the algorithm navigates correctly through the virtual tree to determine the population count of the input number. This step-by-step process ensures that the population count of the number 13 is accurately determined by navigating through the virtual tree and selecting the appropriate child nodes at each iteration.

The time complexity of the proposed algorithm is $O(\log_s(n))$, where s is the size of the mask vector and n is the input number. This is because the while loop in the algorithm executes $\log_s(n) + 1$ times. An important aspect of the algorithm is its efficient space utilization. To search through the entire virtual tree, only the space for a single node is required at any given time. The algorithm uses the variable m to keep track of the integer to be added to the mask vector to generate the next child node. By increasing the size of the mask vector, the execution time can be reduced, as larger masks decrease the number of iterations needed. However, this improvement in time complexity comes at the cost of increased space consumption. To further illustrate this, consider an example where the input number is 368. Figure 2 demonstrates the virtual tree generated by the algorithm to find the population count of 368. This example shows how the algorithm navigates and constructs the tree efficiently, balancing time and space requirements based on the chosen mask size.

Figure 2 also illustrates the values of the algorithm's variables after each iteration. Initially, before the first iteration, the mask vector is set to the root node. The value of d is calculated to be 4, and it decrements by one with each iteration.

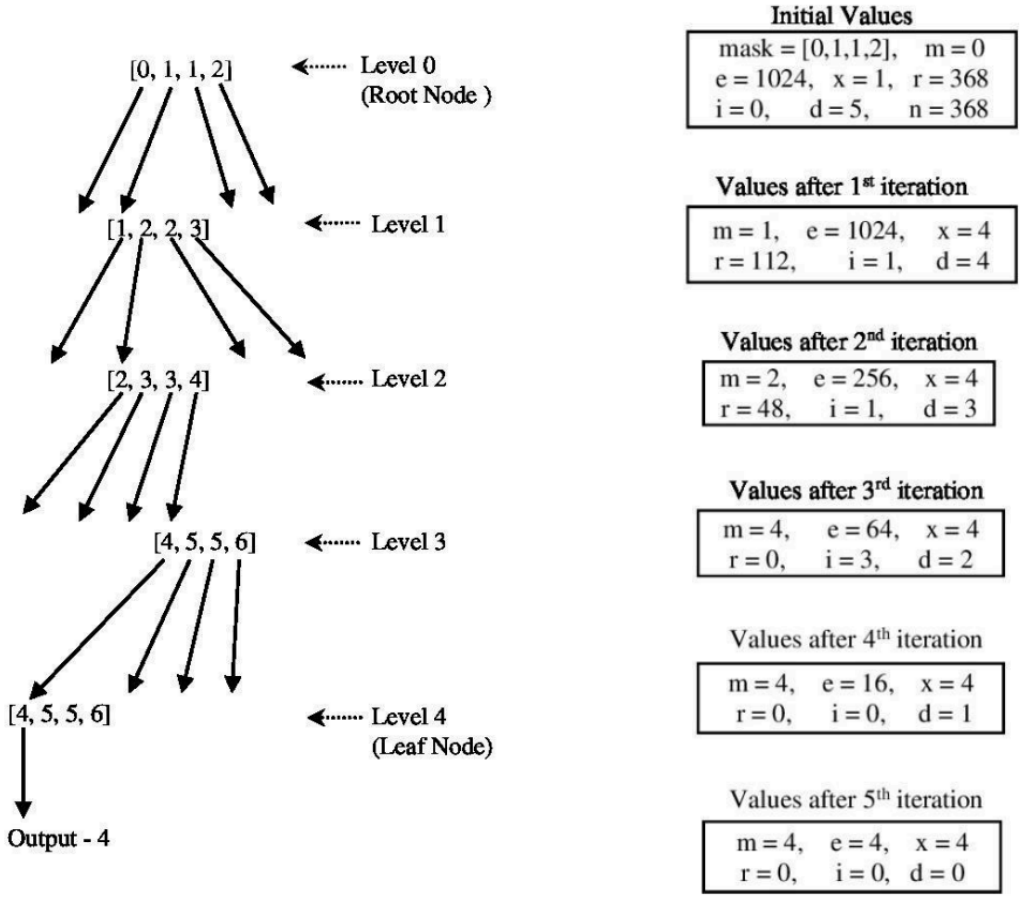

Figure 2: Virtual tree generated to find the population count of 368 (execution starts with the mask as root node). Execution Sequence of the algorithm to find the population count of the number 368. 'i' gives the position of the selected integer for next generation, 'd' controls the iterations, 'm' keeps track of the generated node and after the execution, it carries the output.

In the first iteration, the algorithm selects the integer '1' from the root node for expansion. This selection is based on the calculation described in the algorithm. After generating the first child node, the current node is updated to $\langle 1, 2, 2, 3\rangle$. Using the formula outlined in the algorithm, the next integer chosen from the current node is '2'. This selection leads to the generation of the child node $\langle 2, 3, 3, 4\rangle$. In the following iteration, the algorithm selects '3', resulting in the generation of the node $\langle 3, 4, 4, 5\rangle$. Finally, in the last iteration, '4' is selected from the current node, generating the node $\langle 4, 5, 5, 6\rangle$. In the last step, the algorithm identifies '4' as the first element of the final node, which is chosen as the population count result. This detailed step-by-step process shows how the algorithm navigates through the virtual tree, updating nodes and selecting elements based on predefined calculations to arrive at the final population count.

The proposed VTPC Method is evaluated through a simulation that verifies its accuracy across a range of integers from 0 to 10,000. The evaluation process involves the following steps (Algorithm 2). First, for each integer in the range, the population count is first calculated using the naïve method, which involves counting the number of set bits ('1's)

in the binary representation of the integer. Next, the population count for the same integer is then computed using the proposed VTPC method. Later, the results from both methods are compared. If the population counts match for every integer in the range, the evaluation algorithm returns '1'. If there is any mismatch, the algorithm returns '0'. In the conducted simulation, the evaluation algorithm consistently returns '1', indicating that the proposed VTPC algorithm produces accurate population count results for all tested integers. This validation confirms the correctness and reliability of the VTPC method.

Algorithm 2 Simulation for evaluation of the proposed VTPC approach

Require: R (number of simulation cycles to be executed)
Ensure: 1 if VTPC approach passes the evaluation, otherwise 0
1: $i \leftarrow 0$
2: **while** $i \leq R$ **do**
3: $d_1 \leftarrow$ Naïve_pop_count(i)
4: $d_2 \leftarrow$ VTPC_pop_count(i)
5: **if** $d_1 \neq d_2$ **then**
6: **return** 0
7: **end if**
8: $i \leftarrow i + 1$
9: **end while**
10: **return** 1

4. Time complexity comparison

The naïve method for population count involves examining each bit of the register individually, resulting in a time complexity of $O(M)$, where M is the total number of bits representing the input number. The lookup table method improves upon this by checking chunks of bits at a time. In this approach, the worst-case time complexity is $O(M/L)$, where L is the number of bits evaluated in each lookup operation. The Divide and Conquer technique offers further efficiency, with a time complexity of $O(\log_2(M))$. This is because the number of calculations required is halved at each step. The Carry-Save Adder (CSA) method operates similarly to the Divide and Conquer approach. It reduces the number of calculations by half after every iteration, achieving comparable efficiency. The proposed Virtual Tree for Population Count (VTPC) method has a worst-case time complexity of $O(\log_S(N))$, where N is the input number in decimal number system, and S is the size of the mask vector. Table 1 presents a comparison of the worst-case time complexities for these different algorithms, including the proposed VTPC algorithm, highlighting the efficiency gains of each method.

The line plot in Figure 3 illustrates the theoretical trade-off between space complexity (on the x-axis) and time complexity (on the y-axis) for the VTPC algorithm, across different values of N (8, 32, 128, and 512). Here N represents the decimal number corresponding to which the population count is to be calculated. From the figure, it is noted that for all values of N, as space complexity increases, time complexity decreases. This indicates a typical trade-off where more memory usage (space complexity) can reduce the time required to run the algorithm. Moreover, the decrease in time complexity becomes less significant as space complexity continues to increase. After a certain point (around 10–15 units of

Table 1: Comparison with different techniques based on their time complexity. Here, 'M' is the number of bits used to represent the input number 'N', 'L' is the number of bits used to address the elements of the lookup table, and 'S' is the size of the mask vector used in VTPC.

S. No.	Algorithm	Time Complexity
1	Naïve Method	$O(M)$
2	Look-up Table	$O(M/L)$
3	Divide and Conquer	$O(\log_2(M))$
4	CSA	$O(\log_2(M))$
5	VTPC	$O(\log_S(N))$

space complexity), the time complexity levels off, suggesting that additional space does not significantly improve time efficiency. In addition, higher values of N (larger data sizes) result in a higher baseline for time complexity at lower space complexities. However, the overall shape of the curve remains similar for each N, with each curve flattening at higher space complexity levels. As N increases, the benefit of increasing space complexity is more pronounced at lower levels of space complexity. This suggests that for larger problem sizes, allocating additional memory initially has a larger impact on reducing time complexity, but this benefit diminishes as more space is allocated.

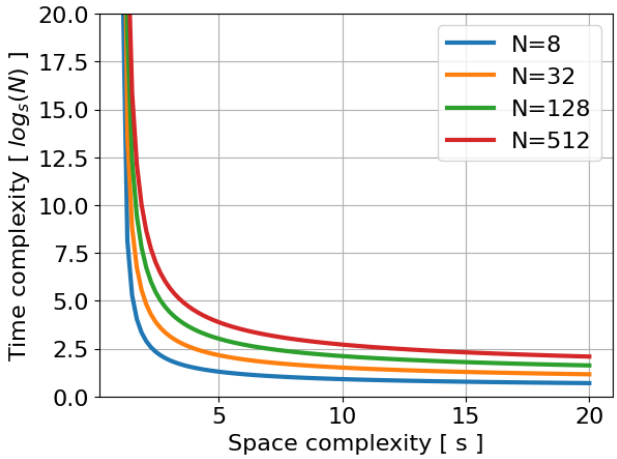

Figure 3: The trade-off between theoretical space complexity and time complexity by the VTPC algorithm.

In this context, Figure 4 shows a line plot that illustrates the relationship between mask size (representing a form of space complexity on the x-axis) and execution time (time complexity on the y-axis, measured in seconds) for different values of N (1000, 10,000, 100,000, and 1,000,000). The simulation for this study was implemented and executed using a Python 3.0 script on the Google Colab platform, running on a system equipped with a dual Intel® Xeon® CPU @ 2.20 GHz, 13 GB of RAM, and 108 GB of disk space.

From the figure, it is observed that for all values of N, there is a steep drop in execution time as the mask size increases from 0 to around 20. This suggests that increasing mask size significantly improves execution time initially, likely because additional memory allows for more efficient processing. Beyond a mask size of approximately 20, the execution time

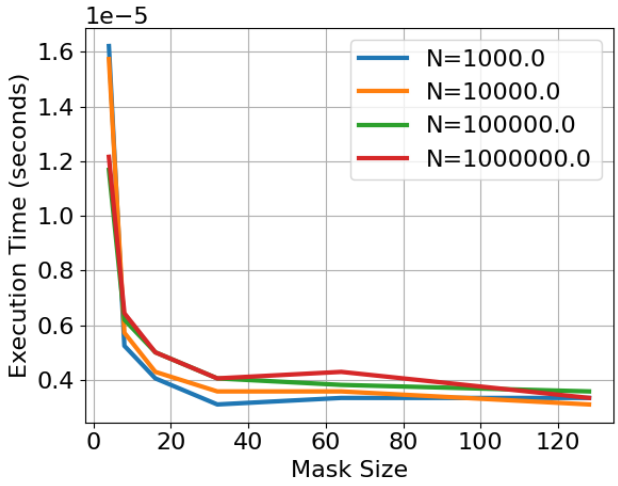

Figure 4: The trade-off between space complexity and time complexity observed through the simulation results.

stabilizes and does not show significant changes as the mask size continues to increase up to 120. This indicates diminishing returns, where increasing the mask size further does not result in noticeable improvements in execution time. Moreover, similar to Figure 3 observations, in Figure 4 also it is is observed that the higher values of N tend to have a slightly higher baseline execution time, but the general pattern of improvement with increasing mask size is similar across all values of N. This suggests that the relationship between mask size and execution time is consistent across different data sizes, with similar diminishing returns for larger datasets.

In summary, the simulation results demonstrate that while increasing mask size (space complexity) initially improves execution time (time complexity), the benefits taper off beyond a certain mask size. This relationship is corroborated by the theoretical time and space complexity graph shown in Figure 3. Furthermore, this trend is consistent across different dataset sizes, showing that the simulation reaches a point of optimal efficiency where further increases in mask size no longer reduce execution time.

The VTPC algorithm, as a software-based population counting method, falls within the complexity class P, meaning it can be solved in polynomial time. By aiming to improve time complexity over traditional population counting methods, VTPC could potentially offer a more efficient approach within this class, especially for large data sets. While it does not shift the problem into a different complexity class, its optimizations could impact practical applications by making previously infeasible or slower computations more accessible and efficient, possibly narrowing the performance gap in applications that require rapid bit-counting, such as cryptography and big data processing.

The VTPC method could also be adapted for parallelization by distributing segments of the bit sequence across multiple cores, allowing each core to independently compute partial population counts. Leveraging multi-core processors, the VTPC algorithm can perform bit-counting concurrently across different parts of the data, aggregating results at the end for a final count. This parallel approach would significantly improve processing speed, especially for large datasets or high-throughput applications.

Furthermore, the VTPC method could be adapted for quantum algorithms by leveraging quantum parallelism to evaluate multiple branches of the virtual tree simultaneously, potentially enhancing population count speed. Quantum algorithms like Grover's search or amplitude amplification might be incorporated to efficiently locate and count '1' bits within superposition states, potentially achieving a faster, more scalable solution for large-scale population count tasks.

5. Conclusion

The need for efficient population counting arises across cryptography, database management, and machine learning, prompting the development of the Virtual Tree for Population Count Method (VTPC) as a faster, hardware-independent solution, streamlining processes and enhancing scalability in high-performance computing environments. The proposed VTPC method offers an innovative approach to calculating the population count of integers. By leveraging a virtual tree structure and mask vector, the VTPC method efficiently navigates through the population count sequence, reducing the time complexity to $O(\log_S(N))$, where N is the input number and S is the size of the mask vector. This stands in contrast to traditional methods such as the naïve bit-by-bit approach, lookup tables, and Divide and Conquer techniques, all of which exhibit higher time complexities under certain conditions. The accuracy and reliability of the VTPC algorithm were validated through a comprehensive simulation, which compared its results with those of the naïve method across a wide range. The consistent match of results underscores the correctness of the VTPC method. Moreover, the VTPC algorithm balances time and space complexity effectively. While increasing the mask size can further reduce execution time, it is essential to consider the corresponding increase in space consumption. The method's flexibility allows it to be tailored to specific application needs, ensuring optimal performance. Overall, the VTPC method represents a significant advancement in population count algorithms, providing a robust, efficient, and accurate solution suitable for various computational tasks. Future work can explore further optimizations and extensions of this approach to other domains requiring efficient bit-level operations.

Acknowledgements

We extend our sincere gratitude to Dr. Anil Kumar and Dr. Sudeep Marwaha, ICAR-IASRI, New Delhi, India, and Dr. Pradip Basak, Uttar Banga Krishi Viswavidyalaya, West Bengal, India, for their invaluable guidance and support throughout this work. We are indeed thankful to the Editors for their guidance and counsel. We are very grateful to the reviewer for valuable comments and suggestions of generously listing many useful references.

Conflict of interest

The authors do not have any financial or non-financial conflict of interest to declare for the research work included in this article.

References

Fateman, R. J. (1989). Lookup tables, recurrences and complexity. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 68–73.

Klarqvist, M. (2021). Efficient analysis and storage of large-scale genomic data. In *University of Cambridge (Doctoral Dissertation)*.

Knuth, D. E. (2014). *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. Addison-Wesley Professional.

Marton, K., Raluca, B., and Suciu, A. (2017). Counting bits in parallel. In *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6. IEEE.

Muła, W., Kurz, N., and Lemire, D. (2018). Faster population counts using avx2 instructions. *The Computer Journal*, **61**, 111–120.

Polychroniou, O., Raghavan, A., and Ross, K. A. (2015). Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508.

Rosen, K. H. (2019). *Discrete Mathematics and Its Applications*. McGraw-Hill.