Computer Science and Information Systems 22(3):927–944 https://doi.org/10.2298/CSIS241201033L

Efficient algorithms for collecting the statistics of large-scale IP address data

Hui Liu[1], Yi Cao[1,⋆], Zehan Cai[1], Hua Mao[2], and Jie Chen[3]

[1] College of Electronic and Information, Foshan Polytechnic,
Foshan 528137, P.R. China
lxyliuhui@163.com
{cycaoyi, lxyliuhui}@hotmail.com
[2] Department of Computer and Information Sciences, Northumbria University,
Newcastle, NE1 8ST, U.K.
hua.mao@northumbria.ac.uk
[3] College of Computer Science, Sichuan University,
Chengdu 610065, P.R. China
chenjie2010@scu.edu.cn

Abstract. Compiling the statistics of large-scale IP address data is an essential task in network traffic measurement. The statistical results are used to evaluate the potential impact of user behaviors on network traffic. This requires algorithms that are capable of storing and retrieving a high volume of IP addresses within time and memory constraints. In this paper, we present two efficient algorithms for collecting the statistics of large-scale IP addresses that balance time efficiency and memory consumption. The proposed solutions take into account the sparse nature of the statistics of IP addresses while maintaining a dynamic balance among layered memory blocks. There are two layers in the first proposed method, each of which contains a limited number of memory blocks. Each memory block contains 256 elements of size 256×8 bytes for a 64-bit system. In contrast to built-in hash mapping functions, the proposed solution completely avoids expensive hash collisions while retaining the linear time complexity of hash-based solutions. Moreover, the mechanism dynamically determines the hash index length according to the range of IP addresses, and can balance the time and memory constraints. In addition, we propose an efficient parallel scheme to speed up the collection of statistics. The experimental results on several synthetic datasets show that the proposed method substantially outperforms the baselines with respect to time and memory space efficiency.

Keywords: large-scale IP addresses, memory blocks, hash table, sorting, network traffic.

1. Introduction

In recent years, the amount of network traffic has increased significantly because of the rapid development of emerging network services, such as video streaming, instant messaging, and online payment services. It is crucial to evaluate the potential impact of the features of user behavior on network traffic management. User behavior features are usually extracted from IP packets, which contain IP addresses. There is a close correspondence between informative user behavior features and the IP addresses that frequently

⋆ Corresponding author

appear in the packets. Hence, how to effectively obtain the statistics of large-scale IP address data in a timely manner, such as every few minutes, is a challenging problem in network traffic measurement. Obtaining the statistics of large-scale IP address data typically consists of two tasks: counting the number of occurrences of each IP address and sorting the results in a specific order.

Each IP address serves as a unique identifier for a device on a network. Analyzing large-scale IP address data can reveal vulnerabilities in the network traffic measurements; this enables administrators to discover and resolve weak spots before they are exploited. As a result, potential cybersecurity risks can be effectively reduced in various network applications. Moreover, analyzing such data helps in understanding traffic patterns and behaviors in network traffic measurement process. The collection and analysis of large-scale IP address data provide valuable insights that can guide more reliable and efficient network systems; this helps administrators balance loads and improve the overall performance of the network. Therefore, collecting statistics from large-scale IP address data is an essential task for the efficient, secure, and scalable management of networks.

A number of statistical algorithms for solving this problem have been studied in the last few decades [11], [15], [16], [8], [2]. A classic divide-and-conquer strategy has been proposed in which IP addresses are first divided into multiple subsets. Then, each subset is individually computed using a statistics collection method, such as a top k trie algorithm [12]. Finally, to merge and sort the results of the multiple subsets, sorting algorithms (e.g., bubble sort, insertion sort, merge sort, selection sort, or quick sort) are used [9], [6]. However, the reduction of computational cost is an intractable problem in the sorting procedure [21], [18], [1]. For example, the average complexity and worst-case complexity of the bubble, insertion, and selection sort algorithms are $O\left(n^2\right)$ [9], [24], where n represents the number of unsorted records. This indicates that the merging and sorting step of the multiple subsets occupies a large amount of memory and is extremely computationally costly to perform when collecting the statistics of millions or tens of millions of IP addresses. Therefore, statistics collection algorithms using the divide-and-conquer strategy still face several challenges caused by the rapid increase of the large-scale records, such as bounded memory and computational cost restrictions.

A hash table is an effective method for collecting the statistics of IP addresses [19]. It uses a hash function to compute a hash codes for an array of buckets with the statistical results. The hash function assigns each key to a unique bucket for each IP address. Unfortunately, the hash function can generate the same hash code for more than one IP address. With the increase in the generation of big data, millions or tens of millions of records have become ubiquitous in network traffic. Therefore, this approach could cause several hash collisions, especially for a large number of IP addresses. Although many strategies can be employed to avoid collisions, such as linear probing, quadratic probing, and double hashing, they require extra storage space and computation.

The statistics collection algorithm should be stable, effective, and efficient for large-scale records. Recent advancements in sorting techniques have concentrated on improving the efficiency and scalability of algorithms for processing large-scale data.To overcome the disadvantages of general statistics collection methods, a number of parallel techniques have been developed for large-scale records by optimizing the efficiency and complexity. For example, these algorithms have been extended to the corresponding parallel structures

for parallel hardware architectures, such as many-core and multi-core platforms [3], [27], [23],[26].

An IP address consists of a series of numbers separated by periods. However, the sorting techniques ignore the importance of the original characteristics of the unsorted IP records. Additionally, parallel statistics collection algorithms are simply parallel computing implementations of the original algorithms. The uniqueness of the IP addresses is vital for distinguishing between the different devices. An IPv4 address is typically represented in four parts. Consequently, collecting statistics from large-scale IP address data still face two significant challenges. First, the unique structure of unsorted IP records is ignored when large-scale IP address data are collected. Second, the extension of the collection task to the corresponding parallel computation paradigm deserves a further investigation into parallel hardware architectures.

In this paper, we present two efficient algorithms for collecting the statistics of large-scale IP address data. We can obtain the frequently occurring IP addresses from the statistics, which can be regarded as a pre-processing step of user behavior analysis in network traffic management. Because of the increasing volume and speed of network traffic, it has become expensive and impractical to handle all IP addresses contained the IP packets. By taking full advantage of the successive characteristics of memory addresses and the fixed range of each individual part of an IP address, we design two relationship mapping mechanisms between memory blocks and IP addresses for a four-dimensional sparse matrix. The sparse matrix stores the number of occurrences of the individual IP addresses, in which the positions of the rows and columns are employed to represent the mapping relationship between the memory blocks and IP addresses. Specifically, we construct a two-layer memory block (TLMB) to implement the first mapping mechanism for the IP addresses. In addition, we employ a single shared memory block (SSMB) for all IP addresses to implement the other mapping mechanism of the IP addresses. The mechanisms of the mapping relationship effectively remove the information about trivial user behaviors that are irrelevant for statistical analysis. The proposed methods can be extended to the corresponding parallel versions for specific hardware architectures. Extensive experiments on several synthetic datasets show the effectiveness of the proposed method.

Our contributions are summarized as follows.

1. We present two efficient methods for collecting the statistics of large-scale IP address data that use two different relationship mapping mechanisms, TLMB and SSMB, between memory blocks and IP addresses for a four-dimensional sparse matrix.
2. The computational cost of TLMB is linearly proportional to the number of IP addresses with a limited memory resource, and the memory use of SSMB remains almost unchanged with a reasonable computational cost, regardless of the number of IP addresses.
3. A parallel computation optimization scheme for multiple computers is proposed to effectively improve the computational efficiency and dramatically reduce memory use.
4. Our extensive experimental results using synthetic datasets demonstrate that our proposed method shows clear superior performance in comparison with the baselines, striking a balance between computational cost and memory use.

The remainder of this paper is organized as follows. We briefly describe some of the original sorting techniques as well as various parallel sorting algorithms in Section 2. In

Table 1. Time complexity of various sorting algorithms [6], [17], [7].

Algorithm	$Best\,(n)$	$Average\,(n)$	$Worst\,(n)$
Bubble sort	$O\,(n)$	$O\left(n^2\right)$	$O\left(n^2\right)$
Insertion sort	$O\,(n)$	$O\left(n^2\right)$	$O\left(n^2\right)$
Selection sort	$O\left(n^2\right)$	$O\left(n^2\right)$	$O\left(n^2\right)$
Merge Sort	$O\,(n \log n)$	$O\,(n \log n)$	$O\,(n \log n)$
Quick Sort	$O\,(n \log n)$	$O\,(n \log n)$	$O\left(n^2\right)$
Heap Sort	$O\,(n \log n)$	$O\,(n \log n)$	$O\,(n \log n)$

Section 3, we introduce the proposed method. The experimental results are presented in Section 4. Finally, we draw the conclusions of the study in Section 5.

2. Related Work

2.1. Classical Sort Techniques

Bubble sort is a classical sorting algorithm in which each element in a list is compared with its neighboring elements and swapped until they are in the desired order [25]. Bubble sort leads to $(n-1)$ number of passes and $\frac{n(n-1)}{2}$ number of iterations if n elements are given. Insertion sort is a simple and efficient sorting algorithm that iteratively takes one element and finds its appropriate position in the sorted list by comparing it with neighboring elements. It becomes less efficient as the number of records increases. Selection sort determined the smallest number in an unsorted list and swaps it with the first number in the sorted list. Then, it finds the next smallest element from the remaining list and swaps with the second element in the sorted list. Consequently, the number of sorted elements at the top of the list increases while the rest remain unsorted. Merge sort, which is based on the divide-and-conquer principle, repeatedly divides the array into two halves and then combines them in a sorted manner For more details of classical sort algorithms, such as quick sort and heap sort, we refer the reader to the comments in [10], [6]. It is well known that computational cost and required memory are the primary concerns in sorting algorithms [22], [14]. Table 1 shows the time complexity of the best case, average case, and worst case of several classical sorting algorithms [6], [17], [7]. These sorting algorithms can be used to find the first k IP addresses of the most frequent occurrences from the statistical results obtained for IP address data.

2.2. Accelerating Large-scale Sorting Techniques

Sorting is an essential part of modern computing. Significant efforts have recently been dedicated to accelerating large-scale sorting techniques [3], [13], [27]. For example, Alhabboub *et al.* improve the computation efficiency of the classical QuickSort algorithm by combining with parallel implementations [3]. The improved QuickSort algorithm can be applied on the sorting large-scale data, and exhibits slightly superior computational efficiency compared to classical sequential QuickSort. Jugé *et al.* proposed an adaptive ShiversSort algorithm for efficiently sorting partially sorted data, which is considered as a variant of the well-known algorithm TimSort [13]. Yang *et al.* proposed a high-performance

parallel sorting algorithm on a CPU-DSP heterogeneous processor [27]. These methods typically take into account different sorting settings, such as parallel sorting environments, data criterions, or specific CPU architectures. These large-scale sorting algorithms provide an efficient post-processing step for collecting the statistics of large-scale IP address data.

2.3. Parallel Hardware Architectures

The hardware architecture of modern processors usually consists of more than two independent central processing units (CPUs) or graphics processing units (GPUs). Parallel software platforms can be implemented using high-level programming frameworks for specific hardware architectures [5]. The Compute Unified Device Architecture (CUDA) is a parallel computing platform for general computing on GPUs. Most parallel sorting algorithms are variants of standard, well-known sorting algorithms adapted to GPU hardware architecture. For example, Cederman designed a quick sort for the GPU platform [4], and Peters proposed an adaptive bitonic sorting algorithm with a bitonic tree for GPUs [20]. The parallel computation of sorting algorithms is considered to be the most efficient way of sorting elements on parallel hardware architectures [26].

Fig. 1. Example of a memory block of size 256×8 bytes containing 256 elements

3. Proposed Method

3.1. Problem Formulation

IP flow data FD is a sequence of IP records, that is, $FD = \{(x_1, p_1), ..., (x_n, p_n)\}$ and $n \geq 1e^6$, where each pair of elements (x_i, p_i) $(i \in [1, n])$ consists of an IP address x_i and a set of corresponding user behavior attributes p_i. Given a finite set of IP addresses $X = \{x_1, x_2, ..., x_n\} \in \mathbb{R}^{m \times n}$, the purpose of the IP address statistics task is to efficiently determine the first k IP addresses of the most frequent occurrences in X, where m the dimensionality of an individual IP address and $k \ll n$.

3.2. IP Address Statistics Task

A standard IP address is composed of four decimal numbers ranging from 0 to 255 which are separated by dot symbols. An individual IP address is logically divided into four parts by splitting it with respect to each dot symbol, and each part of an IP address has an integer value. We create a four-dimensional array for the statistics of IP addresses, where the length of each dimension in the array is 256. Each element of the array can be employed

to store the number of occurrences of the IP address according to the relationship mapping between the index of each dimension of the array and the integer value of the corresponding part of the IP address. For example, consider the individual IP address 1.2.3.4 and the four-dimensional array fd_array. The number of occurrences of this IP address is stored in $fd\_array[1][2][3][4]$. However, individual IP addresses in the host logs often make up a small proportion of all IP addresses. The array can be considered to be sparse because most of its elements are zeros. Consequently, we can carefully design a four-dimensional sparse matrix to store the number of an individual IP address by taking full advantage of the successive characteristics of array addresses and the fixed range of an individual part of an IP address.

Algorithm 1 TLMB

Input:

A finite set of IP addresses $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^{m \times n}$, number $k > 1$.

Construct 256×256 memory blocks of size 128 MB for the first layer;
for $i = 1 : n$ **do**
  Assume that a, b, c and d each represent one of the integer values of the four parts of IP address x_i.
  Calculate the index of the memory block in the first layer: $p = a \times 255 \times 255 + b \times 255$.
  if the value of the j-th element is null in the p-th memory block **then**
    Create a memory block in the second layer, set all elements of the memory block to zero, and store the starting address of the memory block in the j-th element.
  else
    Obtain the starting address of the memory block m in the second layer by finding the j-th element of the p-th memory block.
  end if
  Add 1 to the d-th element of memory block m in the second layer.
end for
Construct a minimum heap of size k using each non-zero element of the memory blocks in the second layer.

Output:

Traverse the nodes of the heap to obtain the k IP addresses and their number of occurrences.

3.3. IP Usage Storage and Retrieval Strategies for the Four-dimensional Sparse Matrix

Assume that each memory address of a 64-bit system can be stored in an element 8 bytes in size, and the number of occurrences of an individual IP address is no more than 2^{64}. There is an array of size 256 elements that consists of 256×8 bytes of memory. The array is regarded as a memory block that contains a contiguous address space, as shown in Fig. 1. In other words, the addresses of all bytes of the array are sequential in the memory block. Therefore, the position of the array can be indexed by the integer value of a particular part of the IP address. We present two efficient methods to collect the statistics of large-scale IP address data, each of which contains a relationship mapping mechanism between memory blocks and IP addresses for the four-dimensional sparse matrix.

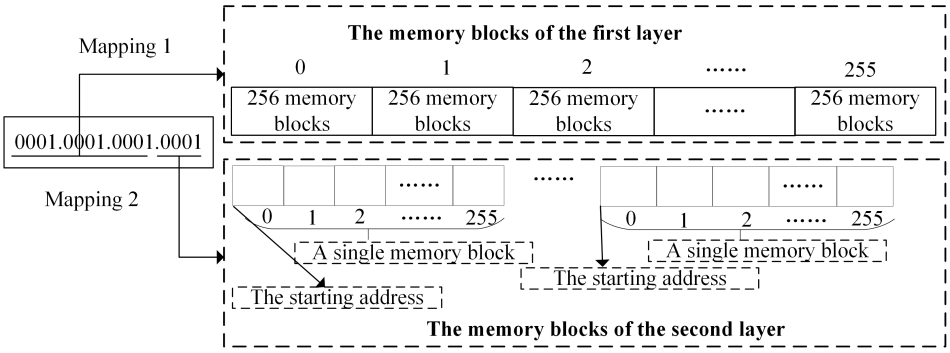


Fig. 2. An example of the mapping relationships between memory blocks and an IP address

First method: TLMB The first proposed mapping mechanism of IP addresses is TLMB. The four parts of the IP address are represented in four layers, where each layer is made up of one or more memory blocks. The first layer only contains one memory block, whereas the second layer contains 256 memory blocks. Each memory block contains 256 elements. Each element of the memory block in the first layer is employed to store the starting addresses of the corresponding 256 memory blocks in the second layer. Similarly, the third layer contains 256×256 memory blocks, the size of which is 128 MB in memory. Then, the element of each memory block in the third layer stores the starting address of the corresponding memory block in the fourth layer. This would be 32 GB in size if we adopted a pre-allocation strategy for all memory blocks in the four layers. Hence, we present an alternative pre-allocation strategy for the memory blocks. A memory block will be allocated only when the first three parts of an initial IP address have been given. In particular, pre-allocating a big memory block of size 128 MB containing 256×256 contiguous memory blocks is feasible in a modern computer. Consequently, the first two layers can be removed from this architecture if the third layer has contiguous memory blocks of 128 MB.

We formally present a storage strategy for IP addresses that consists of two layers that consist of a limited number of memory blocks. The first layer contains 256×256 memory blocks. The first three parts of the IP address can be mapped into the corresponding position of the element in a particular memory block of the first layer according to the individual values of the three parts. We allocate a memory block in the other layer for the IP address when its first three parts are initially given. Each element of a memory block in this layer stores the number of occurrences of the corresponding IP address. Figure 2 shows an example of the relationship mapping between the memory blocks of two layers and an IP addresses.

Consider the individual IP address 1.2.3.4, we have $1 \times 255 \times 255 + 2 \times 255 = 65535$, which represents the index of the memory block in the first layer. The positions of the first two dimensions of the sparse matrix can be mapped to the elements of the memory blocks included in the first layer. The third part of the IP address denotes the index of the memory block in the second layer. The positions of the final dimensionality of the sparse matrix

can be indexed by combining the starting address of the memory block in the second layer with the integer value of the four parts of the IP address.

Algorithm 2 SSMB

Input:

A finite set of IP addresses $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^{m \times n}$, number $k > 1$.

Construct a memory block of size 128 MB saved by all IP addresses;
All IP addresses are logically partitioned into q subsets according to the first part of each IP address.
for $i = 1 : n$ **do**
    Assume that a, b, c and d each represent one of the integer values of the four parts of IP address x_i.
    for $j = 1 : q$ **do**
        if $j == q$ **then**
            Calculate the position of the memory block in the first layer: $p = b \times 255 \times 255 + c \times 255 + d$.
            Add 1 to the p-th element of the memory block.
        end if
    end for
    Construct a minimum heap of size k using the each non-zero elements of the memory block.
end for

Output:

Traverse of the nodes of the heap to obtain the k IP addresses and their number of occurrences.

We traverse all elements of the memory blocks of the second layer to obtain the maximum number of occurrences of elements if $k = 1$. Otherwise, we construct a minimum heap of size k. The statistical results of the first k IP addresses are saved in the heap, which is a special binary tree and implemented by an array of size k. The construction of the heap is completed by traversing all elements of the memory blocks of the second layer. The tree node in the leap contains two important attributes: an IP address and its number of occurrences. The final IP addresses and numbers of occurrences can be obtained by a traversal of the nodes of the heap. The complete procedure for determining the first k of the most frequent IP addresses from host logs is outlined in Algorithm 1.

Second method: SSMB We also designed an SSMB that stores all IP addresses and their statistics. IP addresses are logically divided into at most 256 subsets according to the value of the first part of each individual IP address. We construct a single memory block of $256 \times 256 \times 256$ elements that is $256 \times 256 \times 256 \times 8$ bytes in size, that is, 128 MB. The memory block is shared by all subsets. For each subset, the last three parts of the IP address can be mapped into the corresponding position of the element in the shared single memory block, which is always initialized at the beginning of the relationship mapping. Then, the element of this memory block stores the statistics of the IP addresses in this subset.

We further perform a round traversal of the memory block to initialize a minimum heap of size k after the relationship mapping has been completed in the first subset. Then, we continue to perform a round traversal of the memory block to adjust the heap after the

relationship mapping has been completed for the subsequent subsets. Finally, we obtain the first k most frequently occurring IP addresses in the heap. The complete procedure for finding the first k of the most frequency IP addresses from host logs is outlined in Algorithm 2.

3.4. Memory Use and Complexity Analysis

We first evaluate the memory use and computational complexity of the first proposed method, TLMB. The size of each memory block is 256×8 bytes, and there are 256×256 memory blocks in the first layer. Hence, the size of the memory blocks in the first layer is 128 MB. Assume that the number of the distinct first three parts of the IP addresses is s. The number of memory blocks in the second layer is linearly proportional to s. Moreover, the memory size of the minimum heap is $(k + 8)$ bytes, where each tree node contains two attributes, that is, an IP address and the number of occurrences. The total size of the memory of the proposed method is approximately the sum of the three parts, that is, 128 MB, s KB, and $(k + 8)$ bytes. The computational complexity of the two layers for calculating the IP address statistics is $O(n)$ in Algorithm 1, where n is the number of IP addresses. In addition, the computational complexity of constructing a minimum heap of size k is $O(k \log k)$, where k is the number of tree nodes in the heap. Consequently, the overall computational complexity of the proposed algorithm is $O(k \log k + n)$.

We next evaluate the memory use and computational complexity of the second proposed method, SSMB. The memory size of SSMB is 128 MB. Assume that the number of the distinct first parts of the IP addresses is q. The computational complexity of the mapping mechanism of the IP addresses in Algorithm 1 is $O(qn)$, where n is the number of IP addresses. Similarly, the computational complexity of constructing a minimum heap of size k is $O(k \log k)$ for each subset. Consequently, the overall computational complexity of the proposed algorithm is $O(q(k \log k + n))$.

3.5. Parallel Computation Optimization Techniques

Parallel computation mechanism of TLMB In the worst case, the distinct first three parts of the IP addresses cover all the binary combinations. Hence, the size of the memory blocks in the second layer is 32 GB. We present a parallel computation scheme on multiple computers for improving the computational efficiency and reducing memory use. Assume that there are 2^r computers available for parallel computation, where r represents a positive integer. The task of collecting IP address statistics is then divided into multiple subtasks, which are performed by 2^r computers, respectively, according to the first r bits of the first part of the IP addresses. The number of memory blocks reduces to $(256 \times 256)/2^r$ for 2^r computers. Simultaneously, the number of memory blocks will decrease to $32/2^r$ GB in the second layer. For example, the number of memory blocks in the first layer is 256×64 for each computer when $r = 2$, and the size of memory blocks in the second layer is 8 GB in the worst case. In addition, if four computers perform the task of computing IP address statistics by partitioning the first two bits of the first part of the IP addresses, then the second layers of multiple computers are merged into a complete second layer, where is employed to construct a minimum heap of size k. Finally, the parallel computation results can be obtained in a manner similar to the last step of Algorithm 1.

Table 2. Statistics of the datasets

Data sets	IP Records	Individual IP Addresses	Size	Type
1	5, 000, 000	50, 000	77.5 MB	Synthetic
2	10, 000, 000	100, 000	155 MB	Synthetic
3	50, 000, 000	500, 000	775 MB	Synthetic
4	1, 114, 633	107, 988	14.4 MB	Real
5	1, 430, 258	133, 116	18.5 MB	Real

Parallel Computation Mechanism of SSMB Assume that all IP addresses are logically divided into q subsets according to the value of the first part of an individual IP address. Further assume are q computers for parallel computation, where the statistics collection task of each subset can be performed by an individual computer. A minimum heap of size k is shared among these computers. Hence, this greatly increases the computational efficiency of the task by q times.

4. Experiments

4.1. Experimental Settings

In this section, we evaluate the performance of the proposed methods [4] on two different types of datasets, i.e., three synthetic datasets and two real-world datasets. The three synthetic datasets contain 5 million, 10 million, and 50 million randomly generated IP records. Each individual IP address contains one or more of IP records. The average number of IP records is 100 for each individual IP address. In particular, the IPv4 addresses were specifically divided into four segments in the experiments. These experimental settings ensure a comprehensive evaluation of the capacity of TLMB and SSMB to efficiently collect statistics for large-scale IP address data. Parameter k represents the number of frequently occurring IP addresses. Additionally, two real-world network traffic datasets, provided by the Center for Applied Internet Data Analysis (CAIDA) [5], are collected from various parts of the internet. These two datasets are widely used in networking and traffic analysis research. The statistics of these datasets are summarized in Table 2.

We compared the proposed method with the following baselines:

- **Hash Mapping**. Each IP record is mapped into an entry with a statistical result using a hash table [6]. Next, the statistical results are used to construct a minimum heap of size k.
- **IP Mapping**. All IP records are partitioned into q subsets according to the first part of each IP address. The statistics of the IP records in each subset are mapped into an array, whose memory is pre-allocated on a computer according to the last three parts of each IP address. The first k most frequent IP addresses are chosen from each subset. Next, a minimum heap of size k is constructed using the $q \times k$ IP addresses.

[4] https://github.com/chenjie20/IPStatistics

[5] https://www.caida.org/catalog/datasets/ipv4_prefix_probing_dataset

[6] https://github.com/activesys/libcstl

Table 3. Computational cost (s) of different methods on the three synthetic datasets

Data	k	Hash Mapping	IP Mapping	Ours (TLMB)	Ours (SSMB)
1	10	1,458.14 (2.32)	15.18 (0.07)	**2.51** (0.01)	18.43 (0.05)
	100	1,458.56 (2.76)	15.21 (0.04)	**2.52** (0.01)	18.43 (0.06)
2	10	2,927.23 (11.64)	17.41 (0.11)	**4.92** (0.01)	27.59 (0.05)
	100	2,934.65 (28.93)	17.45 (0.05)	**4.95** (0.01)	27.65 (0.06)
3	10	14,547.09 (13.95)	35.33 (0.09)	**24.22** (0.07)	101.01 (0.17)
	100	14,566.36 (26.78)	35.39 (0.05)	**24.24** (0.04)	101.20 (0.2)

Table 4. Memory use (MB) of different methods on the three synthetic datasets

Data	k	Hash Mapping	IP Mapping	Ours (TLMB)	Ours (SSMB)
1	10	**43.63** (0.12)	16,332.81 (0.07)	189.61 (0.16)	139.77 (0.13)
	100	**43.73** (0.06)	16,332.77 (0.05)	189.64 (0.07)	139.79 (0.07)
2	10	**74.8** (0.12)	16,332.75 (0.12)	239.11 (0.1)	139.74 (0.13)
	100	**74.83** (0.05)	16,332.77 (0.07)	239.06 (0.15)	139.7 (0.12)
3	10	**324.45** (0.99)	16,332.59 (0.06)	628.33 (0.07)	139.6 (0.15)
	100	**324.4** (1.02)	16,332.6 (0.16)	628.27 (0.01)	139.71 (0.16)

Memory blocks were pre-allocated for TLMB and SSMB, with sizes ranging from small-scale (e.g., 5 million) to large-scale (e.g., 50 million) to test scalability. Two metrics were employed to evaluate the sorting performance, that is, computational cost and memory use. All experiments were implemented using the C language on a Windows platform with an Intel i7-9700k CPU and 32 GB RAM.

4.2. Experiment Results

Experimental Evaluation on Synthesized Data Parameter k was set to 10 or 100, and we repeated each experiment 10 times. The average computational costs and standard deviations are reported in Table 3, and the mean memory use and standard deviations are given in Table 4. The results show that TLMB consistently outperformed all the other methods in terms of computational cost. For example, TLMB achieves computational costs of 2.51 s and 2.52 s when $k = 10$ and $k = 100$, respectively. When the number of IP addresses increases from 5 million to 50 million with $k = 10$, the gap in computational costs of TLMB and IP Mapping are 12.6 s and 11.11 s, respectively. We also observed the same advantages when $k = 100$. In addition, SSMB shows competitive results when compared with the comparison methods in terms of memory use. The memory use of SSMB is always approximately 139 MB, even when the number of IP addresses changes. In contrast, the computational cost of SSMB substantially outperforms that of Hash Mapping under different numbers of IP addresses. IP Mapping obtained the lowest computation cost for all numbers of IP addresses. However, the highest memory use results of IP Mapping are consistent with expectations.

Experimental Evaluation on Real-World Data We evaluate the proposed and competing methods on two real-world datasets. Tables 5 and 6 show the computational costs and memory usages levels of different methods, respectively. TLMB consistently incurs lower

Table 5. Computational cost (s) of different methods on the two real-world datasets

Data	k	Hash Mapping	IP Mapping	Ours (TLMB)	Ours (SSMB)
4	10	1675.3	6.73	**0.61**	7.00
	100	1673.9	6.90	**0.62**	7.09
5	10	1996.8	7.33	**0.78**	6.88
	100	1989.9	7.47	**0.80**	7.02

Table 6. Memory use (MB) of different methods on the two real-world datasets

Data	k	Hash Mapping	IP Mapping	Ours (TLMB)	Ours (SSMB)
4	10	238.9	16,392.1	249.8	**183.3**
	100	238.8	16,392.1	249.8	**183.4**
5	10	266.7	16,391.9	281.4	**183.4**
	100	266.7	16,392.3	281.4	**183.4**

computational costs than do the other methods. SSMB and IP mapping demonstrate comparable computational costs. However, SSMB significantly reduces the memory requirements relative to IP mapping. Furthermore, Hash mapping incurs a higher computational cost than the competing methods do because of its use of hash computations, making it prohibitively time-consuming in practice. As the number of IP addresses increases across the two real-world datasets, SSMB maintains relatively stable memory usage. These findings highlight the effectiveness of both TLMB and SSMB.

4.3. Ablation study

To investigate the impact of the memory blocks in the proposed TLMB and SSMB methods, we performed ablation studies on the three synthetic datasets. Specially, we examined two particular cases in the experiments. The hash table was employed to replace the second part of TLMB and SSMB, respectively. The primary goal of the ablation study is to demonstrate the importance of the memory blocks on collecting the statistics of large-scale IP address data. The invariants of TLMB and SSMB corresponding to these two cases are referred to as TLMB$_{hash}$ and SSMB$_{hash}$, respectively.

Tables 7 and 8 show the results of the ablation study regarding the computational cost and memory use. TLMB$_{hash}$ exhibits similar computational cost and memory use compared to SSMB$_{hash}$ on the first two synthetic datasets. Additionally, the computational cost of TLMB$_{hash}$ is slightly higher than that of SSMB$_{hash}$, while its memory usage is marginally lower. TLMB$_{hash}$ and SSMB$_{hash}$ achieve an acceptable computational cost. However, TLMB and SSMB achieves superior performance in computational cost and memory use compared with those of TLMB$_{hash}$ and SSMB$_{hash}$. These results further emphasize that integrating a hash table scheme into TLMB and SSMB is both time-consuming and memory-intensive. Therefore, the results of the ablation study demonstrate the effectiveness of the memory blocks in the proposed TLMB and SSMB methods.

Table 7. Ablation study concerning the computational costs (s) incurred on the three synthetic datasets

Data	k	TLMB$_{hash}$	SSMB$_{hash}$	Ours (TLMB)	Ours (SSMB)
1	10	37.95	37.16	**2.51**	18.43
	100	37.9	37.34	**2.52**	18.43
2	10	75.68	74.51	**4.92**	27.59
	100	75.19	74.05	**4.95**	27.65
3	10	384.72	374.49	**24.22**	101.01
	100	384.23	375.9	**24.24**	101.20

Table 8. Ablation study concerning the memory use (MB) required for the three synthetic datasets

Data	k	TLMB$_{hash}$	SSMB$_{hash}$	Ours (TLMB)	Ours (SSMB)
1	10	3,397.3	3397.2	189.61	**139.77**
	100	3,397.5	3,397.2	189.64	**139.79**
2	10	6,654.4	6,654.3	239.11	**139.74**
	100	6,654.4	6,654.4	239.06	**139.7**
3	10	25,665.6	27,367.6	628.33	**139.6**
	100	26,008.6	26,517.5	628.27	**139.71**

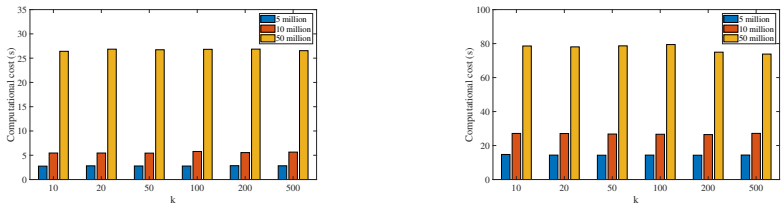
4.4. Empirical Investigation

We empirically examined the effect induced by varying the k considered in the proposed TLMB and SSMB methods. Here k was selected from the set $\{10, 20, 50, 100, 200, 500\}$. The computational cost and memory use were employed to evaluate TLMB and SSMB with different k values.

Fig. 3 shows the computational costs of TLMB and SSMB with different k values. As expected, the computational cost gradually increases as the number of IP records increases from 5 million to 50 million. Moreover, the computational costs of TLMB and SSMB remain relatively stable when k varies from 10 to 500 on each synthetic dataset. This finding demonstrates the stability of TLMB and SSMB for computational efficiency when collecting the statistics of large-scale IP address data. Fig. 4 shows the memory uses of TLMB and SSMB with different k values. We observe that TLMB requires more memory use as the number of IP records grows. In contrast, SSMB maintains relatively stable memory across varying numbers of IP records. This finding indicates that SSMB can satisfy certain memory requirements when handling varying numbers of IP records.

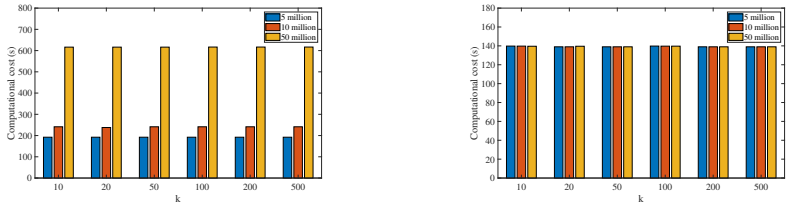
4.5. Discussion

The gap in computational cost between TLMB and Hash Mapping dramatically increases when the number of IP records increases from 5 million to 50 million. This is because TLMB avoids hash collisions when an IP address is mapped to the corresponding memory block. The computational cost of TLMB is linearly proportional to the number of IP addresses. Moreover, the memory use of SSMB remains almost unchanged regardless of the number of IP addresses. This is consistent with the theory underlying the second proposed mapping mechanism. There is a negligible effect on the computational costs of



(a) (b)

Fig. 3. The computational costs of the proposed TLMB and SSMB methods with different k values. (a) TLMB and (b) SSMB



(a) (b)

Fig. 4. The memory uses of the proposed TLMB and SSMB methods with different k values. (a) TLMB and (b) SSMB

TLMB and SSMB when k increases from 10 to 100. Moreover, the changes in the memory use of TLMB and computational cost of SSMB are tolerable in practical applications as the number of IP addresses increases. Consequently, TLMB and SSMB reach a reasonable balance between computational cost and memory use when compared with Hash Mapping and IP Mapping. This reveals that the two relationship mapping mechanisms for memory blocks and IP addresses are effective approaches for the design of the four-dimensional sparse matrix.

The memory blocks designed in TLMB and SSMB exhibit superior relationship mapping capabilities compared to those of hash mapping. The memory block takes fully advantages of the inherent property of the memory address, which is employed to corresponding to each part of an IP address. This indicates that integrating the memory block into TLMB and SSMB is both time-stable and memory-stable. In contrast, hash mapping uses a pair of key and value to store the statistics of IP address data. Unfortunately, IP addresses are often sparse in practical applications. Hash mapping requires additional memory to store the remaining two or three parts of the IP address as keys corresponding to TLMB and SSMB, respectively. This has a significant negative impact on the memory use of hash mapping. Therefore, the proposed memory block significantly enhances the capacity of TLMB and SSMB in collecting the statistics of large-scale IP address data.

5. Conclusion

The collection of the statistics of large-scale IP address data is one of the most fundamental problems in network traffic measurement. In this paper, we addressed this problem. Specifically, the two proposed methods present two different relationship mapping mechanisms between memory blocks and IP addresses to strike a balance between computational cost and memory use. They can be employed to search for frequently occurring IP addresses in practical applications. The extensive experimental results demonstrate the effectiveness of the proposed methods.

Acknowledgments. This work was supported in part by the Guangdong Province Science and Technology Innovation Strategy Special Fund under Grant PDJH2024B648, in part by the Guangdong Province Characteristic Innovation Project for Normal Universities under Grant 2023KTSCX338, and in part by the Province Ordinary Higher Education Engineering Technology Research (Development) Center under Grant 2024GCZX028.

References

1. Abdel-Hafeez, S., Gordon-Ross, A.: An efficient o(n) comparison-free sorting algorithm. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25(6), 1930–1942 (Jun 2017)
2. Agapitos, A., Lucas., S.M.: Evolving efficient recursive sorting algorithms. In: 2006 IEEE International Conference on Evolutionary Computation. pp. 2677–2684. Vancouver, BC, Canada (Jul 2016)
3. Alhabboub, Y., Almutairi, F., Safhi, M., Alqahtani, Y., Almeedani, A., Alguwaifli, Y.: Accelerating large-scale sorting through parallel algorithms. Journal of Computer and Communications 12(1), 131–138 (Jan 2024)
4. Cederman, D., Tsigas, P.: A practical quicksort algorithm for graphics processors. In: European Symposium on Algorithms. pp. 246–258 (2008)
5. Chen, S., Qin, J., Xie, Y., Zhao, J., Heng, P.A.: A fast and flexible sorting algorithm with cuda. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 281–290. Taipei, Taiwan, China (Jun 2009)
6. Cormen, T., C.Leiserson, Rivest, R., C.Stein: Introduction to Algorithms. MIT press (2009)
7. Faujdar, N., Ghrera, S.P.: Analysis and testing of sorting algorithms on a standard dataset. In: 2015 Fifth International Conference on Communication Systems and Network Technologies. pp. 1–10. Gwalior, India (Apr 2015)
8. Fredman, M.L.: An intuitive and simple bounding argument for quicksort. Information Processing Letters 3(114), 137–139 (Mar 2014)
9. Hammad, J.: A comparative study between various sorting algorithms. International Journal of Computer Science and Network Security 15(3), 358–367 (Mar 2015)
10. Idrizi, F., Rustemi, A., Dalipi, F.: A new modified sorting algorithm: A comparison with state of the art. In: 2017 6th Mediterranean Conference on Embedded Computing. pp. 1–6. Bar, Montenegro (Jul 2017)
11. Jing, Y.N., Tu, P., Wang, X.P., Zhang, G.D.: Distributed-log-based scheme for ip traceback. In: The Fifth International Conference on Computer and Information Technology. pp. 711–715. Shanghai, China (Dec 2005)
12. Jing, Y.N., Tu, P., Wang, X.P., Zhang, G.D.: Space-efficient data structures for top-k completion. In: Proceedings of the 22nd international conference on World Wide Web. pp. 583–594. Rio de Janeiro, Brazil (May 2013)

13. Jugé, V.: Adaptive shivers sort: an alternative sorting algorithm. ACM Transactions on Algorithms 20(4), 1–55 (Aug 2024)
14. Jukna, S., Seiwert, H.: Sorting can exponentially speed up pure dynamic programming. Information Processing Letters 159, 451–469 (Apr 2020)
15. Kapur, E., Kumar, P., Gupta, S.: Proposal of a two way sorting algorithm and performance comparison with existing algorithms. International Journal of Computer Science, Engineering and Applications 2(3), 61–78 (Jun 2012)
16. Klein, S.T.: On the connection between hamming codes, heapsort and other methods. Information Processing Letters 113(17), 617–620 (May 2017)
17. Kocher, G., Agrawal, N.: Analysis and review of sorting algorithms. International Journal of Scientific Engineering and Research 2(3), 81–84 (Mar 2014)
18. Louza, F.A., Gog, S., Telles., G.P.: Optimal suffix sorting and lcp array construction for constant alphabets. Information Processing Letters 118, 30–34 (Sep 2017)
19. Múller, I., Sanders, P., Lacurie, A., Lehner, W., Fárber, F.: Cache-efficient aggregation: Hashing is sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. p. 1123–1136. Melbourne, Bictoria, Australia (May 2015)
20. Peters, H., Schulz-Hildebrandt, O., Luttenberger, N.: Fast in-place, comparison-based sorting with cuda: a study with bitonic sort. Concurrency and Computation: Practice and Experience 23(7), 681–693 (Jan 2011)
21. Puglisi, S.J., Smyth, W.F., Turpin, A.: The performance of linear time suffix sorting algorithms. In: Data Compression Conference. pp. 358–367. Snowbird, UT, USA, USA (Mar 2005)
22. Rusu, I.: Sorting signed permutations by reversals using link-cut trees. Information Processing Letters 132, 44–48 (Apr 2018)
23. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–10. Rome, Italy (May 2009)
24. Shabaz, M., Kumar, A.: Sa sorting: a novel sorting technique for large-scale data. Journal of Computer Networks and Communications pp. 1–7 (2019)
25. Shutler, P.M.E., Sim, S.W., Lim, W.Y.S.: Analysis of linear time sorting algorithms. The Computer Journal 51(4), 451–469 (Jul 2008)
26. Singh, D.P., Joshi, I., Choudhary, J.: Survey of gpu based sorting algorithms. International Journal of Parallel Programming 46(6), 1017–1034 (Apr 2018)
27. Yang, M., Zhang, P., Fang, J., Liu, W., Huang, C.: thsort: an efficient parallel sorting algorithm on multi-core dsps. CCF Transactions on High Performance Computing 20(4), 1–16 (Jan 2024)

References

1. Abdel-Hafeez, S., Gordon-Ross, A.: An efficient o(n) comparison-free sorting algorithm. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25(6), 1930–1942 (Jun 2017)
2. Agapitos, A., Lucas., S.M.: Evolving efficient recursive sorting algorithms. In: 2006 IEEE International Conference on Evolutionary Computation. pp. 2677–2684. Vancouver, BC, Canada (Jul 2016)
3. Alhabboub, Y., Almutairi, F., Safhi, M., Alqahtani, Y., Almeedani, A., Alguwaifli, Y.: Accelerating large-scale sorting through parallel algorithms. Journal of Computer and Communications 12(1), 131–138 (Jan 2024)
4. Cederman, D., Tsigas, P.: A practical quicksort algorithm for graphics processors. In: European Symposium on Algorithms. pp. 246–258 (2008)
5. Chen, S., Qin, J., Xie, Y., Zhao, J., Heng, P.A.: A fast and flexible sorting algorithm with cuda. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 281–290. Taipei, Taiwan, China (Jun 2009)

6. Cormen, T., C.Leiserson, Rivest, R., C.Stein: Introduction to Algorithms. MIT press (2009)
7. Faujdar, N., Ghrera, S.P.: Analysis and testing of sorting algorithms on a standard dataset. In: 2015 Fifth International Conference on Communication Systems and Network Technologies. pp. 1–10. Gwalior, India (Apr 2015)
8. Fredman, M.L.: An intuitive and simple bounding argument for quicksort. Information Processing Letters 3(114), 137–139 (Mar 2014)
9. Hammad, J.: A comparative study between various sorting algorithms. International Journal of Computer Science and Network Security 15(3), 358–367 (Mar 2015)
10. Idrizi, F., Rustemi, A., Dalipi, F.: A new modified sorting algorithm: A comparison with state of the art. In: 2017 6th Mediterranean Conference on Embedded Computing. pp. 1–6. Bar, Montenegro (Jul 2017)
11. Jing, Y.N., Tu, P., Wang, X.P., Zhang, G.D.: Distributed-log-based scheme for ip traceback. In: The Fifth International Conference on Computer and Information Technology. pp. 711–715. Shanghai, China (Dec 2005)
12. Jing, Y.N., Tu, P., Wang, X.P., Zhang, G.D.: Space-efficient data structures for top-k completion. In: Proceedings of the 22nd international conference on World Wide Web. pp. 583–594. Rio de Janeiro, Brazil (May 2013)
13. Jugé, V.: Adaptive shivers sort: an alternative sorting algorithm. ACM Transactions on Algorithms 20(4), 1–55 (Aug 2024)
14. Jukna, S., Seiwert, H.: Sorting can exponentially speed up pure dynamic programming. Information Processing Letters 159, 451–469 (Apr 2020)
15. Kapur, E., Kumar, P., Gupta, S.: Proposal of a two way sorting algorithm and performance comparison with existing algorithms. International Journal of Computer Science, Engineering and Applications 2(3), 61–78 (Jun 2012)
16. Klein, S.T.: On the connection between hamming codes, heapsort and other methods. Information Processing Letters 113(17), 617–620 (May 2017)
17. Kocher, G., Agrawal, N.: Analysis and review of sorting algorithms. International Journal of Scientific Engineering and Research 2(3), 81–84 (Mar 2014)
18. Louza, F.A., Gog, S., Telles., G.P.: Optimal suffix sorting and lcp array construction for constant alphabets. Information Processing Letters 118, 30–34 (Sep 2017)
19. Múller, I., Sanders, P., Lacurie, A., Lehner, W., Fárber, F.: Cache-efficient aggregation: Hashing is sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. p. 1123–1136. Melbourne, Bictoria, Australia (May 2015)
20. Peters, H., Schulz-Hildebrandt, O., Luttenberger, N.: Fast in-place, comparison-based sorting with cuda: a study with bitonic sort. Concurrency and Computation: Practice and Experience 23(7), 681–693 (Jan 2011)
21. Puglisi, S.J., Smyth, W.F., Turpin, A.: The performance of linear time suffix sorting algorithms. In: Data Compression Conference. pp. 358–367. Snowbird, UT, USA, USA (Mar 2005)
22. Rusu, I.: Sorting signed permutations by reversals using link-cut trees. Information Processing Letters 132, 44–48 (Apr 2018)
23. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–10. Rome, Italy (May 2009)
24. Shabaz, M., Kumar, A.: Sa sorting: a novel sorting technique for large-scale data. Journal of Computer Networks and Communications pp. 1–7 (2019)
25. Shutler, P.M.E., Sim, S.W., Lim, W.Y.S.: Analysis of linear time sorting algorithms. The Computer Journal 51(4), 451–469 (Jul 2008)
26. Singh, D.P., Joshi, I., Choudhary, J.: Survey of gpu based sorting algorithms. International Journal of Parallel Programming 46(6), 1017–1034 (Apr 2018)
27. Yang, M., Zhang, P., Fang, J., Liu, W., Huang, C.: thsort: an efficient parallel sorting algorithm on multi-core dsps. CCF Transactions on High Performance Computing 20(4), 1–16 (Jan 2024)

Hui Liu received the PhD degree in Information and Communication Engineering from Hohai University, Nanjing, China in 2021. From 2003 to 2021, he served as a full-time faculty member at Jiangxi University of Science and Technology. He is currently an Associate Professor with the School of Electronic Information, Foshan Polytechnic, China. His current research interests include deep learning, computer image processing, and big data analysis.

Yi Cao received the MSc degree in Software Engineering from the School of Computer Science and Information Engineering, Anhui Normal University, in 2022. From 2022 to date, he has been a full - time teacher, assistant professor, and senior engineer at the School of Electronics and Information, Foshan Vocational and Technical College. His current research interests include blockchain technology and big data analytics.

Zehan Cai is a Class of 2022 student at the School of Electronic Information, Foshan Polytechnic, China. His research focuses on machine learning.

Jie Chen received the BSc degree in Software Engineering, MSc degree and PhD degree in Computer Science from Sichuan University, Chengdu, China, in 2005, 2008 and 2014, respectively. From 2008 to 2009, he was with Huawei Technologies Co., Ltd. as a software engineer. He is currently an Associate Professor with the College of Computer Science, Sichuan University, China. His current research interests include machine learning, big data analysis, and deep neural networks.

Hua Mao received the B.S. degree and M.S. degree in Computer Science from University of Electronic Science and Technology of China (UESTC) in 2006 and 2009, respectively. She received her Ph.D. degree in Computer Science and Engineering from Aalborg University, Denmark in 2013. She is currently a Senior Lecturer in Department of Computer and Information Sciences, Northumbria University, U.K. Her current research interests include Deep Neural Networks and Big Data.

Received: December 01, 2024; Accepted: February 28, 2025.