



## Investigating the efficiency of multithreading application programming interfaces for parallel packet classification in wireless sensor networks

Mahdi ABBASI<sup>1,\*</sup>, Milad RAFIEE<sup>1</sup>, Mohammad R. KHOSRAVI<sup>2,3</sup>

<sup>1</sup>Department of Computer Engineering, Faculty of Engineering, Bu-Ali Sina University, Hamedan, Iran

<sup>2</sup>Department of Computer Engineering, Persian Gulf University, Bushehr, Iran

<sup>3</sup>Telecommunications Group, Department of Electrical and Electronic Engineering, Shiraz University of Technology, Shiraz, Iran

Received: 25.10.2019

Accepted/Published Online: 08.01.2020

Final Version: 08.05.2020

**Abstract:** This paper investigates the most appropriate application programming interface (API) that best accelerates the flow-based applications on the wireless sensor networks (WSNs). Each WSN include many sensor nodes which have limited resources. These sensor nodes are connected together using base stations. The base stations are commonly network systems with conventional processors which are responsible for handling a large amount of communicated data in flows of network packets. For this purpose, classification of the communicated packets is considered the primary process in such systems. With the advent of high-performance multicore processors, developers in the network industry have considered these processors as a striking choice for implementing a wide range of flow-based wireless sensor networking applications. The main challenge in this field is choosing and exploiting an API which best allows multithreading; i.e. one which maximally hides the latency of performing complex operations by threads and increases the overall efficiency of the cores. This paper assesses the efficiency of Thread, Open Multiprocessing, and Threading Building Blocks (TBB) libraries in multithread implementation of set-pruning and grid-of-tries packet classification algorithms on dual-core and quad-core processors. In all cases, the speed and throughput of all parallel versions of the classification algorithms are much more than the corresponding serial versions. Moreover, for parallel classification of a sufficiently large number of packets by both classification algorithms, TBB library results in higher throughput and performance than the other libraries due to its automatic scheduling and internal task stealing mechanism.

**Key words:** Packet Classification, Multi-threading, Thread, Open Multiprocessing, Threading Building Blocks, wireless sensor network, efficiency

### 1. Introduction

Wireless sensor networks (WSNs) are ad hoc networks consisting mainly of small sensor nodes with limited resources and one or more base stations, which are computationally more powerful nodes that connect the sensor nodes to other parts of the network [1–6]. The rapid development of the WSN based on the Internet of Things has brought considerable contest and complex problems in processing data in high-speed networks [7–9]. The continually growing number of sensor nodes in WSN has increased the transmitted traffic and the variety of monitored parameters. This extensible complexity in the information of the WSN should be handled at the base stations [7, 10–13].

To accelerate the complex functions in analyzing high-dimensional and self-correlated data over WSN, packet classification, a primary process inherited form software-defined networking (SDN), is inevitable [7,

\*Correspondence: [abbasi@basu.ac.ir](mailto:abbasi@basu.ac.ir)

10, 11, 14–17]. This process accelerates the overall process of base stations by provisioning fast flow-based network functions instead of slow packet-based ones [18, 19]. Packet classification is nothing but discriminating packets according to a set of predefined rules [20, 21]. Accelerating this fundamental process directly affects the overall performance of the WSN systems so that the packet loss, delay, and buffer requirement are reduced [4, 11, 14, 18, 22, 23].

Work done on the parallelization of packet classification algorithms in wireless sensor networks is few and far between. The work of Rjakamal et al. can be considered the preliminary work in using parallel packet classification in WSNs [24]. They introduce the design of a parallel coprocessor which could assist the network processor of a base station in a WSN to classify network packets very fast. Xianyang et al. introduced using classification techniques for provisioning the quality of service in WSN [25]. Some researchers like Letswamotse et al., introduce the idea of software-defined wireless sensor networks (SDWSN) as a network computing paradigm for applying software-defined networking (SDN) strategies to WSNs [26]. This network paradigm is enriched with the idea of the parallel classification of network packets.

The review of the literature shows a few studies that considered parallel packet classification techniques in WSNs. Moreover, none of them considered the limited computational resources of the base stations of WSN for parallel packet classification. This key challenge has been the main motivation of the proposed study which investigates the best multithreading programming interface which lets optimum using of multicore processors on the base stations of the WSNs to accelerate the flow-based applications.

Recently due to increasing volume of incoming packets, multicore processors are used in the architecture of base stations to speed up packet classification [10, 27, 28]. Interestingly, while the concept of parallel programming is almost as old as the computer itself, the design and implementation of parallel algorithms is still a challenge for many developers. Various application programming interfaces (APIs) such as Open Multiprocessing (OpenMP) and Threading Building Blocks (TBB) have so far been produced for code parallelization in different programming languages [29–31]. However, an important task ahead of the developers of parallel programming on multicore systems is to select the most appropriate API to get a high-performance parallel application.

In this regard, this paper examines the most common APIs (C++ 11, OpenMP, TBB) that support code parallelization in different programming languages with the aim of parallelizing packet classification algorithms. The `std::thread` class from the standard library of C++ 11 easily enables the programmer to create threads from the thread library in the program code. In the simplest case, the task to be accomplished by the thread is a recursive or void function with or without parameters. In this case, the function is executed by the thread until the recursion, and then the thread stops. OpenMP API is another flexible API that allows for multithreaded parallelization with shared memory architecture. OpenMP includes a set of compiler instructions, variables, and library functions for parallelization of programs as used in computational applications in the fields of science and engineering [32]. A great advantage of OpenMP is its simple programming method using compiler instructions. However, the programmer should have a good command of these instructions to decide how and when to use them [33]. TBB is a C++ library for parallelization that extends C++ through abstraction of thread management and simple parallel programming. To use this library, the programmer specifies tasks rather than threads and allows the library itself to efficiently assign tasks to threads. One of the advantages of this strategy is its scalability. This means that the programmer can use more processor cores with higher performance [30].

The packet classification algorithms examined in this paper are the set-pruning trie algorithm and the grid of tries (GOT) algorithm. These two algorithms use a data structure based on source and destination prefix-address decision trees to solve packet classification issues [20]. The reason for choosing these two algorithms is the difference in the tree structures produced as well as their method of traversal to classify an Internet packet based on the fields of source and destination IP address. This difference in data structure and type of processing during parallelization with these APIs results in different performances.

The innovations of this article are as follows:

- This paper examines for the first time the classification algorithms of set pruning and GOT in terms of their parallelization on multicore processors.
- Different parallelization techniques such as thread, OpenMP, and TBB were used to implement the parallel version of the above-mentioned algorithms.
- Having been parallelized with the mentioned APIs, the classification algorithms were then implemented on a number of multicore processors and the results were compared according to several evaluation criteria.

The structure of the paper is organized as follows. In Section 2, the structure of the set-pruning trie and GOT algorithms is reviewed. Section 3 discusses thread, OpenMP, and TBB parallelization libraries. In the next section, related work on parallelization of classification algorithms on multicore processors is reviewed. The proposed method for parallelizing the packet classification algorithms as well as how to implement and evaluate the performance of each API in the parallelized algorithms is described in Section 5 after a brief discussion of the performance indicators. The final part of the paper consists of some concluding remarks and suggestions for further research.

## 2. Related tools and algorithms

This section describes set-pruning trie algorithm and GOT algorithm as well as their method of packet classification. In these examples, the filter set in Table 1 is used to describe the algorithms.

**Table 1.** Example filter set.

Filter no	Source prefix	Destination prefix	Source port	Destination port	Protocol
Filter 1	*	0*	53	1024-65535	6
Filter 2	1011*	01*	53	443	17
Filter 3	0010*	01100*	0-65535	5632	4
Filter 4	01*	*	0-65535	25	17
Filter 5	*	10*	53	443	6
Filter 6	10110*	110*	0-65535	25	17

### 2.1. Set-pruning trie algorithm

The set-pruning trie algorithm is an improved hierarchical trie algorithm. To produce a hierarchical trie from the prefixes of source and destination addresses in the classifying filters, a trie is first created based on the distinctive prefixes of the source address. Each node in the source hierarchical trie which corresponds to one or more filters with the same prefix is linked through a pointer to the root of a binary trie containing the destination address prefixes of the same filters. The search function in the hierarchical trie is done sequentially

in each trie. Thus, for each matching node in the source tree with the source address of the incoming packet, the search continues on the destination trie which is attached to that node. After the target tree is scanned, the search is returned to a node in the source trie at which the algorithm had jumped to the destination trie. This process is called backtrack. It continues until it reaches a leaf node in the source trie. Backtrack is one of the factors in a hierarchical trie that increases the amount of memory access and, thus, the time of traversal[34, 35].

The set-pruning algorithm avoids backtracking by copying the filters of the target trie of the ancestors of each child node into the target trie of that node. The set-pruning trie corresponding to the hierarchical trie of Figure 1a which was created for the filter set in Table 1 is shown in Figure 1b. In Figure 1a, for example, the root node of the source trie is an ancestor for all other nodes. Moreover, the node corresponding to the prefix 01\* is an ancestor for the node 0010\* and the node 1011\* is an ancestor for the node 10110\*. As shown in Figure 1b, the entire target trie of ancestor nodes are copied and merged into the target trie of child nodes. Filters copied from the ancestors into the target trie are displayed in green [36].

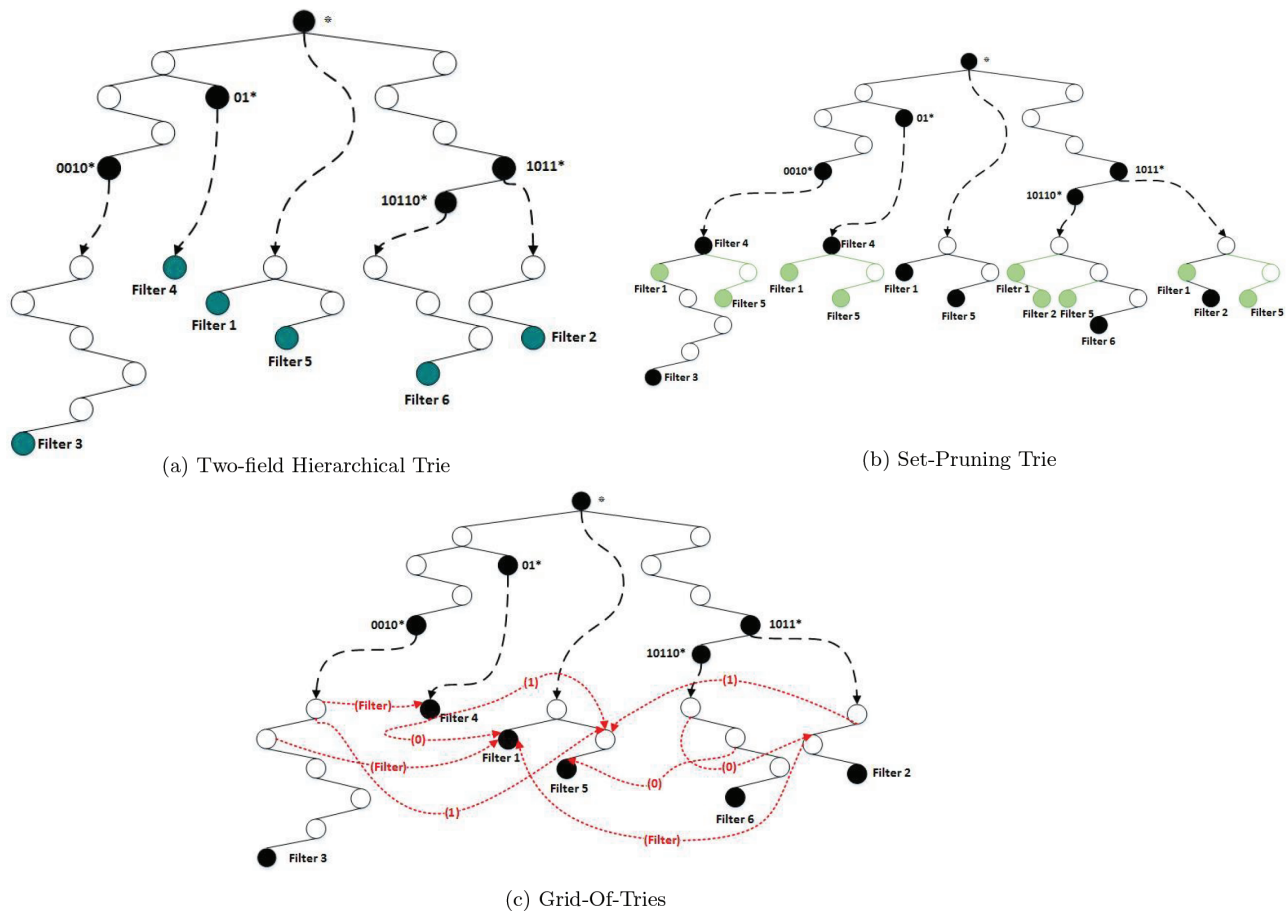


Figure 1. The structure of hierarchical trie, set-pruning trie, and GOT algorithm

## 2.2. Grid of tries (GOT) algorithm

The hierarchical trie algorithm imposes high costs as a result of backtracking. On the other hand, a set-pruning trie algorithm needs too much storage space although it removes many of the back-tracks of the previous

algorithm. The grid-of-tries approach uses the key ideas of precomputation and switch pointers to speed up the search. By these means, GOT jumps from the failed F2 trie to another F2 trie. In the preprocessing step, all of failure points in the F2 tries are identified and all of the required switch pointers are allocated. This pointer in any failed F2-trie allows the search algorithm to jump directly to the next possible ancestor F2 trie that contains a matching filter. The jump occurs at the lowest point in the ancestor F2 trie that has at least as good an F2 prefix match as the current node. Furthermore, such a jump allows skipping over all filters in the next ancestor F2 trie with shorter prefixes in the F2 field. This change in the data structure of the hierarchical trie is shown in Figure 1a, and the results of its conversion to GOT trie data structure is indicated in Figure 1c. The most important achievement of GOT algorithm is that its memory consumption is approximately equal to that of the hierarchical trie algorithm while its search complexity is equal to that of the set-pruning trie algorithm. The search function in this trie is executed in the same way as a set-pruning algorithm. For example, for the input (10110, 11011, 53, 25, 17), the total number of traversed nodes is 10 [36].

### 3. Parallelization libraries

#### 3.1. Thread

One of the easiest ways to implement parallelization on multicore processors is to use the C++ standards. C++ 11 is a shared-memory-concurrency language with explicit thread creation and implicit sharing. This means that every address can be read and written by defined threads. Therefore, the programmer is responsible for preventing interference among the threads, or the race state. Programming using a primary threading interface, such as thread or Pthreads, is an option many programmers use for parallelization. Thread is part of the standard C++ library. This library uses the `std::thread` class to manage thread execution. This class can start a new thread and wait until the execution of that thread is complete. The thread processing library provides a one-to-one mapping from `std::thread` onto the threads of the operating system. It also provides simple instructions for working with operating system threads [37, 38].

#### 3.2. OpenMP

OpenMP is a shared memory programming interface in C, C++, and Fortran on all major compilers and platforms. The library includes parallelization patterns such as parallel blocks, loop, and tasks supported by basic concepts like data sharing and synchronization. Parallel algorithms can be expressed in simple, compact pieces of code to improve productivity. In addition, OpenMP provides a more abstract interface than many common schedulers used in applied programming. This is a great advantage for implementation of scientific and analytical algorithms. The underlying concept of this library is the Fork/Join model for parallelization which is shown in Figure 2. The command used to interpret parallel blocks is `#pragma omp parallel` [39].

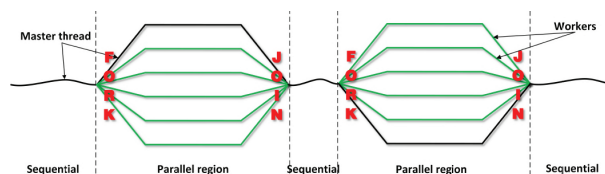


Figure 2. OpenMP parallelization model.

In this model, each parallel block has a master thread, a number of slave threads, and a specified operation. The master thread and the slave threads are collectively called a team. Each team has a certain size which is

expressed in terms of the number of master and slave threads. At the beginning of each parallel block, there is a fork to synchronize the master and slave threads. After all the threads of the team have reached the fork, each member of the team begins to execute part of the team’s operations that is assigned to it. These parts are assigned by the compiler at compile time. At the end of each parallel block, a join barrier is used to synchronize the master and slave threads so that the execution of the program continues sequentially through the master thread. Therefore, it can be said that processing in this model is divided into three main steps: the first stage is forking. At this step, the master thread first takes a team from the pool of defined teams or creates a new team. Then it sets the team size to a specific value and starts to run. The second step is execution in which the master thread processes along with slave threads that part of the team’s operations which is assigned to it. The last step is joining. Here, the master thread creates a barrier and waits for all the slave threads to complete their work. Eventually, the team is collected; that is, it is either returned to the pool team or destroyed [39].

### 3.3. Threading Building Blocks (TBB)

TBB was first introduced by Intel in 2006 as a parallel programming library. Figure 3 illustrates the general structure of TBB and how it works to create threads and balance their workloads. This library allows parallelization to be interpreted both explicitly and implicitly. In the explicit mode, spawning provides the programmer with full control over the work of each task. The implicit state can be achieved using patterns such as `parallel_for` or `parallel_reduce` that accelerate code-writing. Tasks created explicitly or implicitly are added to the queue of thread tasks in an abstract space called threads arena. These tasks are carried out by the master thread or by other workers through a mechanism called theft. In what follows we shall have a look at this concept.

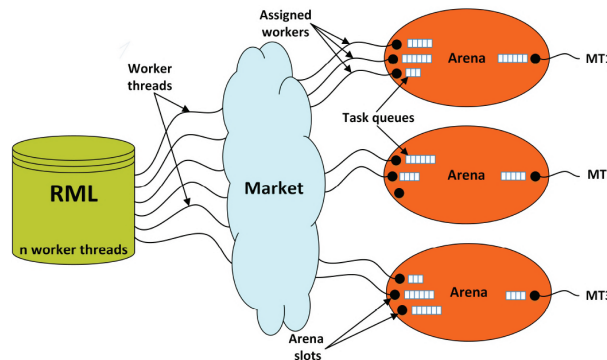


Figure 3. Components of TBB’s task scheduler.

TBB’s master thread represented by MT in this figure is a software thread that instantiates the `TBB::task_scheduler_init` object. All threads that are created by MT and are used to complete MT’s task are referred to as worker threads. The resource management layer (RML) is the host of a pool of worker threads. The role of the Market is to distribute the workloads of master threads as well as to assign workers to the arenas of master threads for carrying out the allocated workloads. The number of available worker threads is always one less than the maximum of `tbb::task_scheduler_init` argument and the total number of logical cores on the system CPU. The next structure is the arena of each MT that encapsulates all the tasks and resources available (worker threads) to execute a master thread. A number of slots are assigned to each arena that represent the number of worker threads which are needed to complete the parallel tasks of the MT. If the

total number of slots needed for all master threads exceeds the number of workers in the RML pool, allocation of slots to the arena of MTs will be tailored to the needs. When the task of the master thread comes to an end, the threads produced at the time of creating each arena are either destroyed or assigned by RML to active arenas. Each working thread, when run in an arena, executes a scheduling procedure called `wait_for_all()` which consists of three nested loops. The inner loop executes the current task by calling the `execute()` method. Upon completion of this task, if no further task is called, the program exits the inner loop. In the middle loop, the `get_task()` method attempts to dequeue local tasks in a last-in-first-out (LIFO) order. If successful, the inner loop will be called again. Otherwise, the thread exits the middle loop and the outer loop activates the stealing mechanism by calling the `receive_or_steal_task()` method. This method searches for all tasks on this level. The search includes sending tasks through the task-thread dependency mechanism, reloading tasks without uploading priority, or reloading tasks left by other workers. If the search does not return a task to run, this method steals from a victim thread that is randomly selected at the current location. If the failures of a worker thread to steal exceed a certain threshold (a default value of 100) and the arena of the MT is empty, the failed worker is released and returned to the pool of RML. The details of how tasks are stolen can be found in many sources such as [29, 40].

#### 4. Related works

In this section, we shall review studies that have parallelized packet classification algorithms on multicore processors. In 2011, Pong et al. parallelized the algorithm based on Hashing Round-Down Prefixes (HaRP) and HyperCuts trie-based algorithm by using multicore processors [41]. Although the parallelization library used in their research is not specified, their results at best represent a rate of 14.3 million and 4.07 packets per second for HaRP and HyperCuts algorithms, respectively. Zhou et al. implemented in 2013 two packet classification algorithms called linear search algorithm and domain tree search algorithm on a multicore processor [31]. In their study, the algorithms were implemented using the Pthread parallelization library on a 16-core processor. The maximum throughput obtained in this study was about 11.5 Gbps.

In 2015, Qu et al. implemented the bit-vector algorithm, which is a decomposition-based algorithm, in a parallel manner on a multicore processor [42]. Decomposition-based algorithms have two phases of search and combination of results. They used the OpenMP library to parallelize the algorithm. The maximum throughput obtained in this study was 14.7 million packets per second.

A review of related research suggests that algorithms based on hierarchical set-pruning tries and GOT have not yet been parallelized on multicore processors. Moreover, despite the extensive capabilities of the TBB library compared to other parallelization libraries, none of the studies have made use of the TBB library to parallelize classification algorithms on multicore processors. Therefore, the present study for the first time implements and compares the performance of packet classification algorithms using C++ parallelization libraries thread, OpenMP, and TBB. In the next section, we examine the parallelization of the set-pruning trie and GOT trie algorithms using the above-mentioned parallelization libraries.

#### 5. Implementation and assessment

To classify the input packets of a classifier on a single-core processor, at each step one input packet is received and classified by traversing the tree created from the filters based on the fields extracted from the packet header. In this case, it is obvious that increase in the number of packets increases the time of classification linearly. Therefore, in order to reduce the time of classifying the set of input packets, we distribute them according to



the mechanisms reviewed in Section 3 over the active threads on the cores of a multicore processor so that the cores could classify the packets in a parallel manner. The main reason for choosing this approach is that when a classification algorithm is divided into several parts which should be executed in parallel, the results of each part may affect the functioning of other parts due to dependencies of data or controlling. Needless to say, this dependency would reduce the efficiency of parallelization. To solve this problem, we parallelize the algorithm in a way that the results of each part could not affect the other parts. This way, the highest performance will be obtained. Therefore, in the proposed parallelization method, packets are distributed among cores that share a copy of the classifying trie and perform their tasks independently.

The pseudocode of the proposed parallel implementation method is shown in Algorithm 1. The input of this algorithm is filters  $F$  and header of packets  $P$ . This method stores the index of the filter that best matches each packet in an output array (BMF\_Result). In the preprocessing step, the tree is created using the filters (lines 1-2). In the proposed method, the packets are divided between threads and each thread classifies the number of specified packets (lines 3-10). At first, each thread computes a given packet index based on the total number of threads and the number of packets, and stores it in the  $Pindex$  variable (line 5). Then, the specified packet is classified by traversing the tree and the index of the best matching filter is stored in the  $fIdx$  variable (line 6). Finally, the algorithm stores the result in the proper element of the output array (lines 7-9).

Due to data-parallel nature of the proposed parallelization model, it is applicable to other algorithms like decomposition-based, tuple space-based, etc.

```

Input : Filters  $F$ , Packets  $P$ 
Output:  $BMF\_Result$ 
Pre – processing construct  $T$  from  $F$ 
ParallelPacketClassification
for  $j \leftarrow 0$  to  $\lceil \frac{|P|}{|threads|} \rceil$  do
     $Pindex \leftarrow (j, |threads|, |P|)$ ,  $fIdx \leftarrow Null$ 
     $fIdx \leftarrow Classify(Pindex, T, F)$ 
    if  $fIdx \neq Null$  then
         $BMF\_Result(Pindex, fIdx)$ 
    end
end

```

**Algorithm 1:** Parallel implementation of packet classification algorithm

### 5.1. Implementation

In this study, according to the architecture of the processors, whose exact specifications are listed in Table 2, we have defined eight threads each of which selects and classifies a number of input packets corresponding to its index. Moreover, we have used the thread, OpenMP, and TBB libraries for parallelization.

The proposed methods were implemented on two different multicore systems which are described in Table 2. The Classbench tool was used to create the required data set [43]. This tool generates the dummy filters required for the classifier as well as the corresponding dummy header sets that are required as test input packets. This tool meets the needs of the developers of packet classification algorithms to genuine and heterogeneous filters used in firewalls, IP chains, and access control lists. Due to lack of real datasets, in the majority of the studies [44–50], the ClassBench tool has been used for producing the required data structure due to a need for filters and packets that are close to reality in terms of structural characteristics and statistical distribution. We



**Table 2.** Specifications of the processors

Specifications	First processor	Second processor
Name	Intel Core i7-740QM	Intel Core i7-3770K
Clock speed	1733 MHz	3500 MHz
L1 data cache	4 * 32 KB	4 * 32 KB
L1 inst. Cache	4 * 32 KB	4 * 32 KB
L2 cache	4 * 256 KB	4 * 256 KB
L3 cache	6 MB	8 MB
Main memory	4 GB DDR3	32 GB DDR3
# of Cores	4	4
Operation system	Windows 7 Ultimate, 64-bit	

produced a set of filters corresponding to the ACL2 parameter containing 1k and 10k filters with 1k to 64k packets to assess the proposed method.

**5.1.1. Results**

The results of the implementation of the parallelization methods of set-pruning and GOT algorithms on 1k filters and 10k filters are presented in Figures 4–11. The criteria for assessing the algorithm on each processor include throughput, speedup, and efficiency which are presented by formulas (1)–(3), respectively. Throughput refers to the number of packets classified per second, and speedup refers to the difference in packet classification time between parallel and serial modes. Here, the unit of throughput is million packets per second.

$$Throughput = \frac{NumberofPackets}{ClassificationTimeinsecond} \tag{1}$$

$$Speedup = \frac{SequentialClassificationTime}{ParallelClassificationTime} \tag{2}$$

$$Efficiency = \frac{Speedup}{NumberofCores} \tag{3}$$

In this way, Figures 4–7 show the throughput and speedup of the two algorithms on the first processor and Figures 8–11 plot the same criteria for the algorithms as implemented on the second processor for input packets from 1k to 64k.

According to the results shown in the graphs of speedup and throughput for each threading library, thread needs the highest time for parallel classification in different modes of equal input packets. Moreover, when the number of input packets is small, the time needed by the thread-parallelized algorithm is higher than the time needed by the serial version of the same algorithm for classification of the same packets. In other words, in cases where the number of available components for parallel implementation is low, the relative increase in the complexity of the timing of the threads in this library in comparison to the complexity of execution would reduce its time efficiency. This can be seen in graph b in all Figures (4–11) when the number of input packets in the parallel version of the algorithms on both processors is less than or equal to 4096. In graph b in Figures

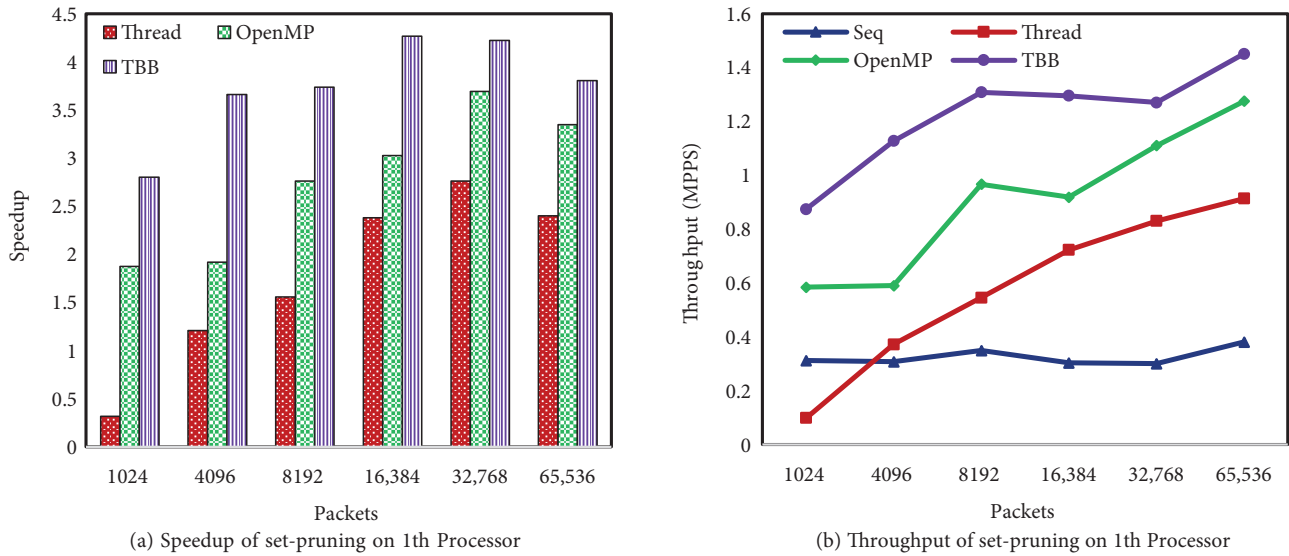


Figure 4. Results of implementation on 1k filters (set-pruning\_1th processor).

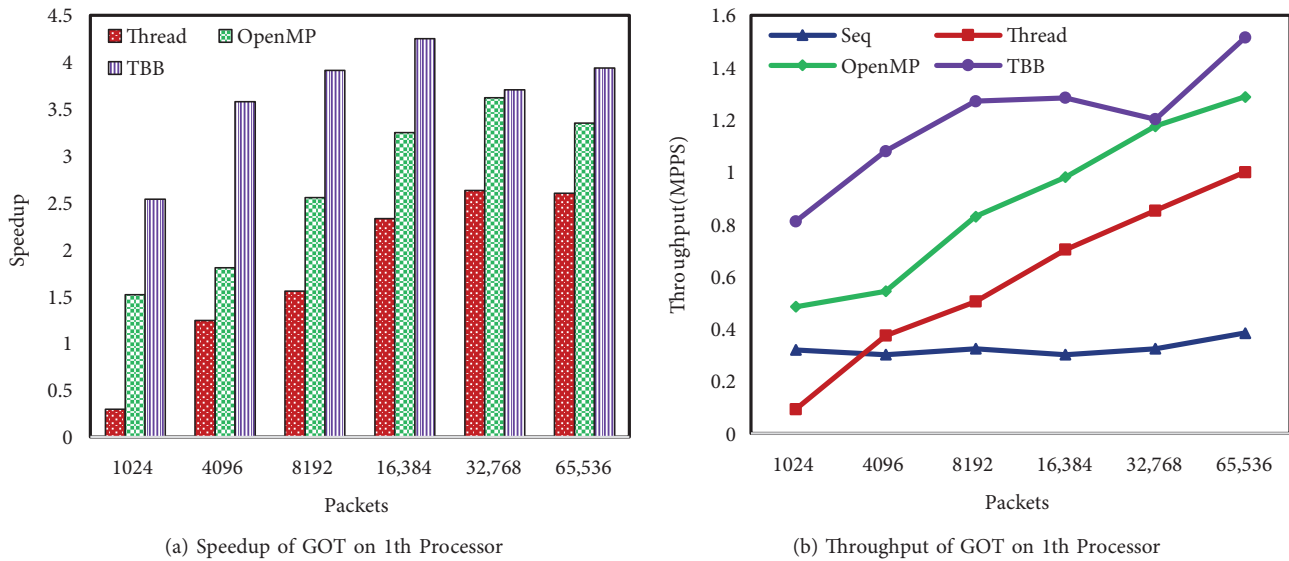


Figure 5. Results of implementation on 1k filters (GOT\_1th processor).

9 and 11 which address the implementation of GOT algorithm on the second processor, for example, when the number of packets is 1k and 4k, the throughput of thread is less than the serial version but this has not happened in OpenMP and TBB and, even with a small number of packets, these libraries have led to a higher throughput compared to the serial version.

With regard to the throughput graph b in Figures 4–11, TBB library shows the best performance in the packet classification. In these graphs, the throughput of OpenMP is not optimum because the programmer himself should choose a dynamic or static scheduling policy to execute the loops. In the meanwhile, the TBB Scheduler automatically defines the execution of the loops through defined threads by using divide-and-conquer

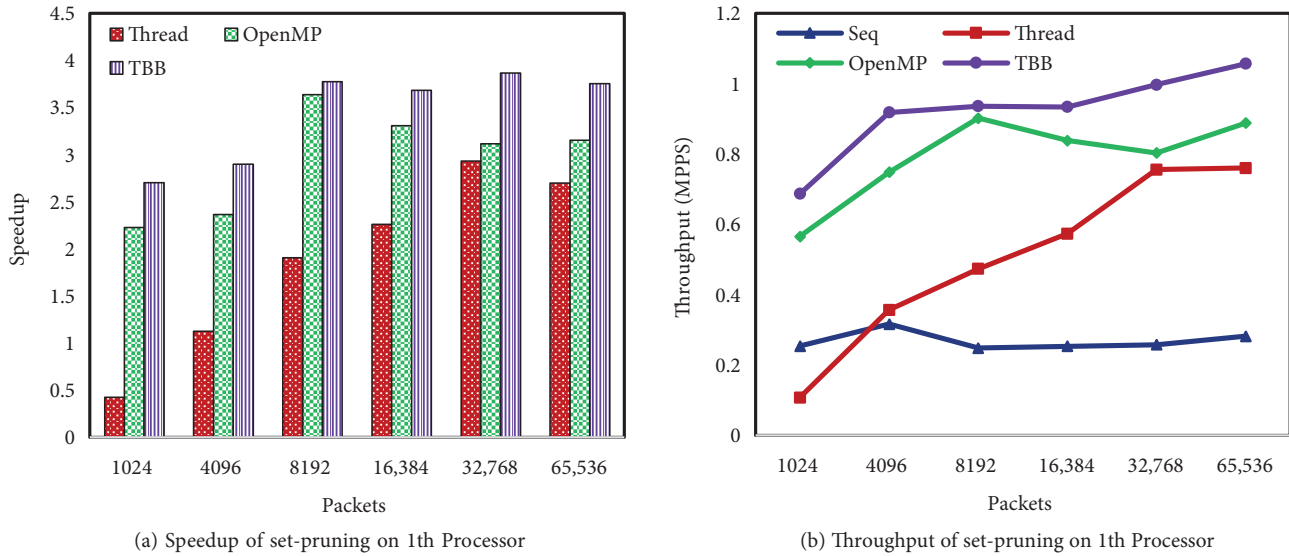


Figure 6. Results of implementation on 10k filters (set-pruning\_1th processor).

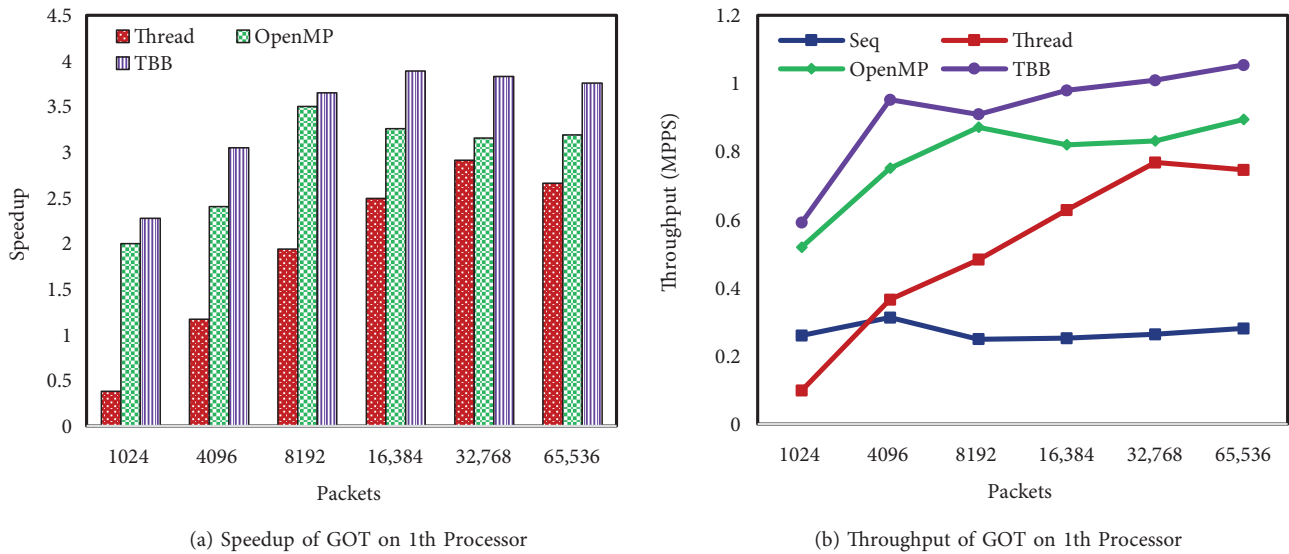


Figure 7. Results of implementation on 10k filters (GOT\_1th processor).

method. In addition, TBB becomes more efficient than dynamic and static schedules by using the technique of stealing tasks during the execution of threads. Comparison of the throughput of the corresponding data on both processors as represented in Figures 4–11 indicates that the throughput of both classification algorithms with all of the three libraries (i.e. OpenMP, thread, and TBB) is better on the second processor than on the first one. The main reason for this superiority is that the classification time of the second processor is much less than the first processor. The maximum throughput for set-pruning algorithm using thread, OpenMP, and TBB is 4.6, 6.37, and 8.09 million packets per second, respectively. This value for GOT algorithm and using the same libraries is 4.96, 6.5, and 7.97 million packets per second.

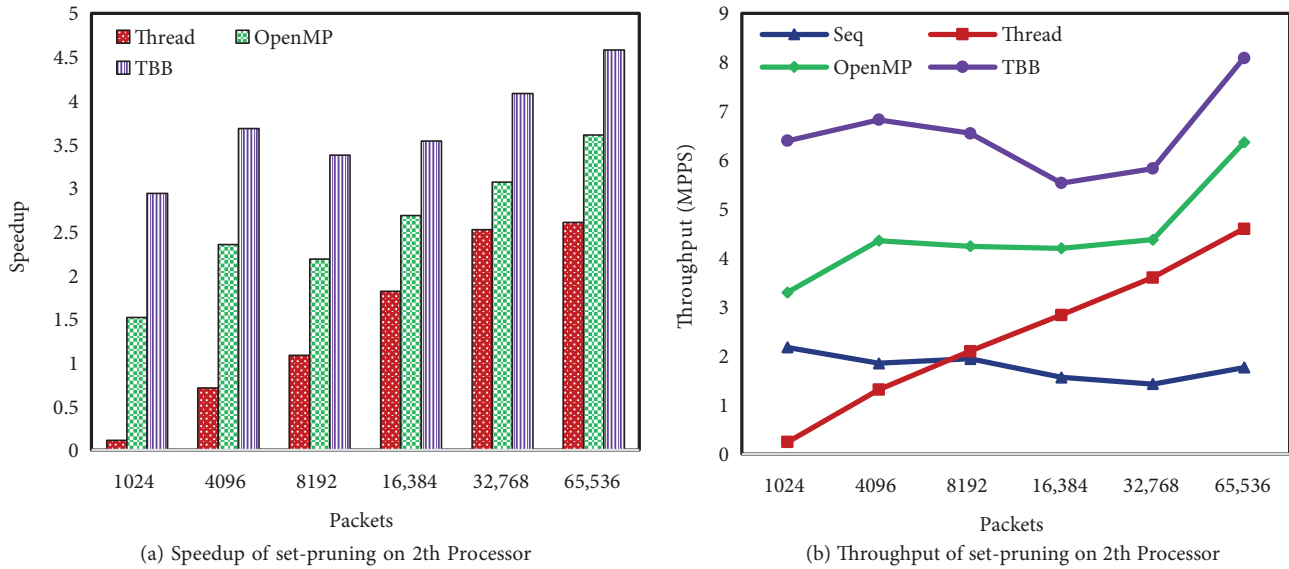


Figure 8. Results of implementation on 1k filters (set-pruning\_2th processor).

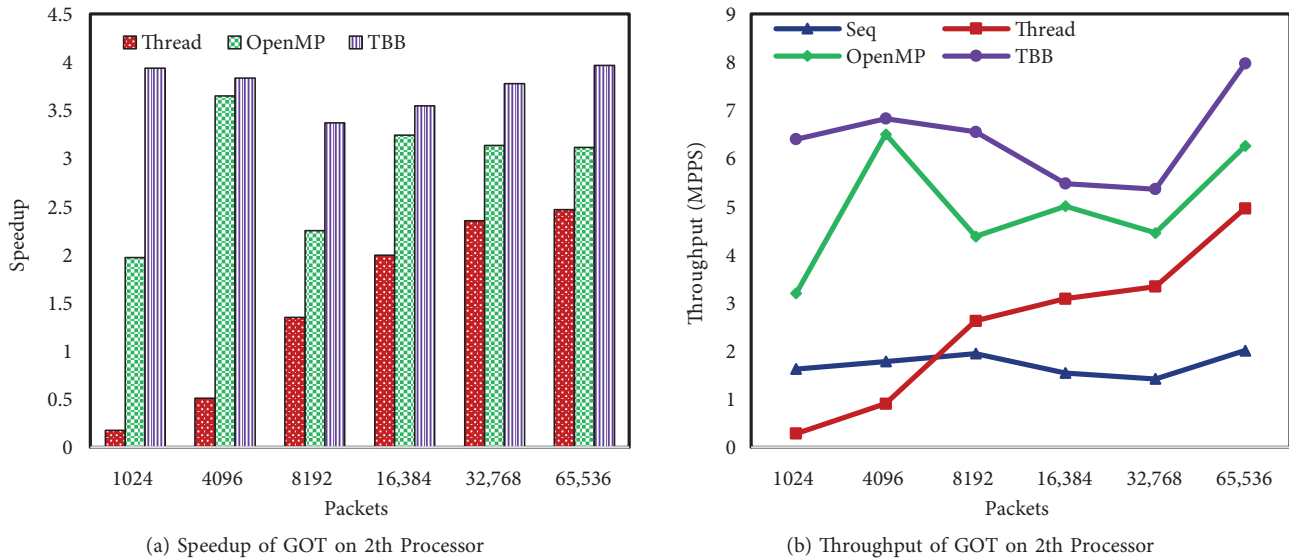
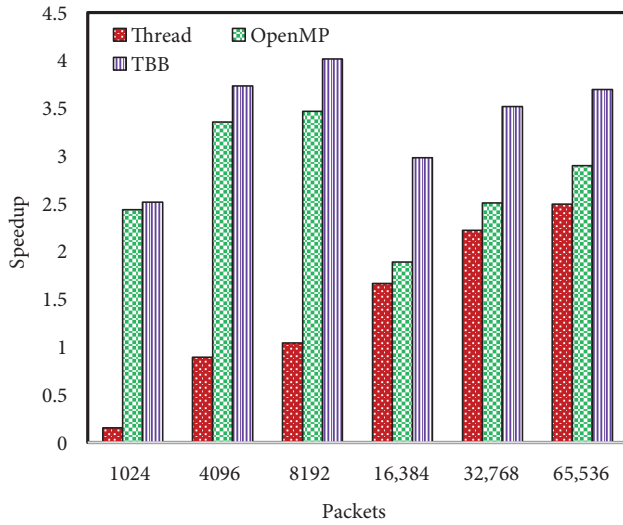
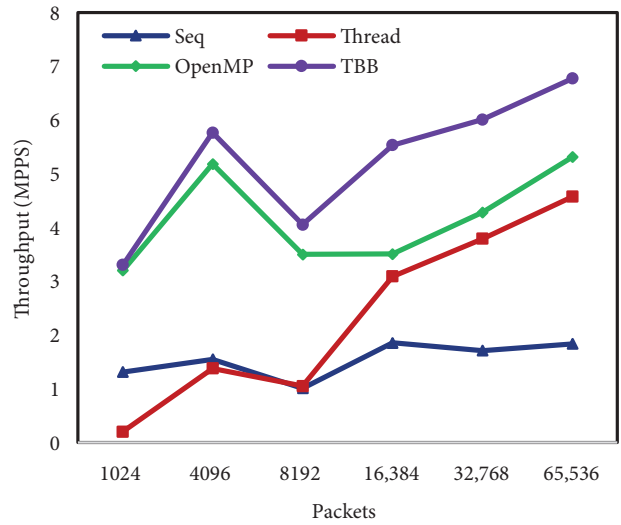


Figure 9. Results of implementation on 1k filters (GOT\_2th processor).

Moreover, comparison of the speedup in parallelization of the algorithms using the three libraries as shown in graph a of Figures 4–11 would indicate that TBB yielded better results for both algorithms and on both processors. Moreover, in most scenarios for both evaluated algorithms, the speedup of classifying a certain number of packets based on 1k or 10k filter sets in the first processor is higher than in the second processor. This observation could be best justified by the higher classification time of the serial version on the first processor in comparison with the second one as well as the greater time difference between the serial and parallel versions of the algorithm on the first processor. The best speedup value obtained by using thread, OpenMP, and TBB for the set-pruning algorithm is 2.93, 3.69, and 4.58 times, respectively, and for GOT 2.91, 3.65, and 4.2 times, respectively.

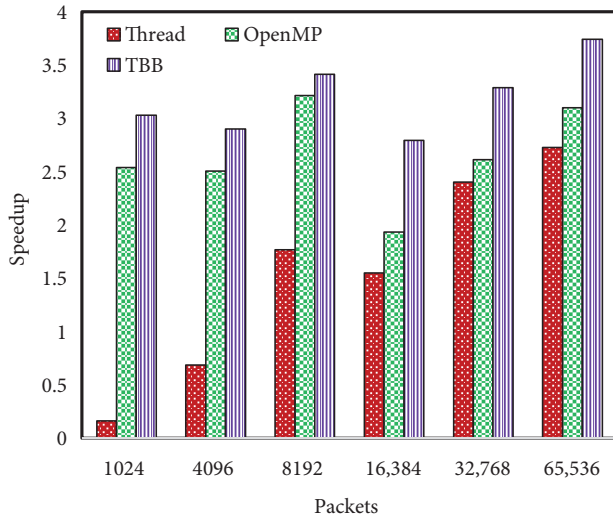


(a) Speedup of set-pruning on 2th Processor

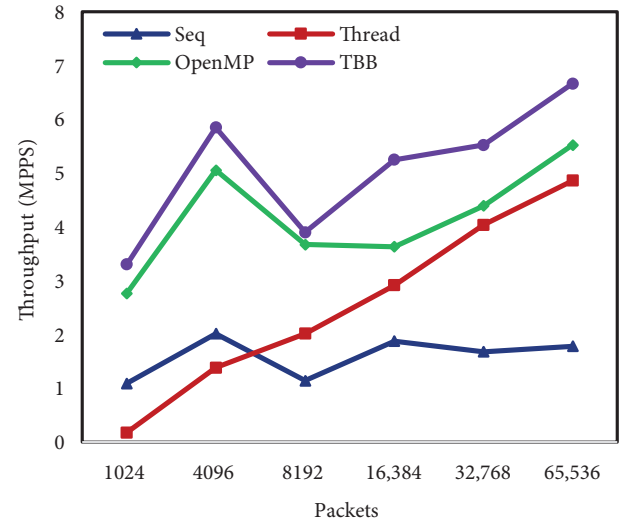


(b) Throughput of set-pruning on 2th Processor

**Figure 10.** Results of implementation on 10k filters (set-pruning\_2th processor).



(a) Speedup of GOT on 2th Processor



(b) Throughput of GOT on 2th Processor

**Figure 11.** Results of implementation on 10k filters (GOT\_2th processor).

As can be seen from Figures 4–11, the throughput and speedup of parallelization for both algorithms using the three libraries has reduced in the classification of a certain number of input packets. This reduction is partly due to the values of the header fields of the packets in a class as well as the number of threads defined on the processor that executes classification. When the address field of the packet header can be matched with longer prefixes in the corresponding fields of classifying filters, the first factor above can increase the frequency of memory access and classification time as in the serial version of the algorithm. In Graph b in Figures 10 and 11, for example, the throughput of the serial version of set-pruning and GOT algorithms on both processors is significantly reduced in the 8k input packet classification as compared to the 4k packet classification.

On the other hand, given the fact that packets are divided among the defined threads for classification, the volume of thread tasks varies with the number of packets to be classified. For these reasons, the performance of the parallel libraries in question depends on the number of packets.

Table 3 compares the maximum, minimum, and the average efficiency of competitor multithreading APIs in parallel classification of different volumes of packets. According to this table, the maximum, minimum, and the average efficiency of the APIs in parallelizing GOT are more than those in parallelizing set-pruning. In all cases, the efficiency of TBB is the highest whereas the OpenMP and Thread stand in second and third ranks, respectively.

**Table 3.** Efficiency of parallel packet classification methods.

Algorithm	Efficiency	1th Processor			2th Processor		
		Thread	OpenMP	TBB	Thead	OpenMP	TBB
Set-pruning	Max	0.73276	0.76588	0.90941	0.65169	0.76636	0.90185
	Min	0.03981	0.27891	0.23429	0.02771	0.37903	0.37903
	Avg	0.34725	0.50614	0.54418	0.36072	0.60507	0.66607
GOT	Max	0.72832	0.73783	0.87527	0.68212	0.77172	0.9127
	Min	0.0367	0.24982	0.18957	0.04073	0.478	0.48337
	Avg	0.35171	0.47526	0.53249	0.37813	0.58382	0.69321

It can generally be concluded that the TBB library is better than other libraries in parallelizing the two algorithms on different processors so that it results in the highest throughput and speedup in classification of 1k to 64k test packets by matching them with 1k to 10k classifying filters. Moreover, the OpenMP achieves the second rank in accelerating the packet classification algorithms on both of multicore processors. Moreover, the second processor, due to higher performance parameters such as clock frequency, shows better results than the first processor.

## 6. Conclusion

In this paper, set-pruning and GOT algorithms which use decision trees to classify network packet headers were parallelized with thread, OpenMP, and TBB libraries and the performance of these libraries for this task was investigated and assessed. To this end, due to the different capabilities of the above-mentioned libraries, the parallelized versions of the two algorithms were executed on two multicore processors to classify various numbers of input packets (from 1k to 64k) by matching them with filter sets of varying sizes (from 1k to 10k). The aim was to measure and compare the speedup and throughput in all scenarios. It can be inferred from the obtained values of speedup and throughput that all parallel versions of the classification algorithms were much more efficient than the serial versions. Moreover, comparison of the parallelization libraries in corresponding scenarios would indicate that the number of input packets of the parallel classifier is so effective on the thread library that, in some cases, the small number of packets causes the performance of the library to become lower than the serial version of the algorithms. In general, using the TBB library to parallelize both classification algorithms resulted in higher throughput and performance than other libraries due to its automatic scheduling and internal task stealing mechanism. The highest throughput rates obtained by the set-pruning algorithm and the GOT algorithm were 8.09 and 7.97 packets per second, respectively. In future research, we will parallelize the algorithms on graphics processing unit (GPU) and compare the results with the best parallelization scenario

obtained in the present study which was implemented with the help of the TBB library. Investigation of the efficiency of different multithreading APIs on clusters of multicore processing systems is one of the most important feature works.

### References

- [1] Belfkih A, Duvallet C, Sadeg B. A survey on wireless sensor network databases. *Wireless Networks* 2019; 25(8): 4921-4946.
- [2] Menon VG, Pathrose PJ, Priya J. Ensuring reliable communication in disaster recovery operations with reliable routing technique. *Mobile Information Systems* 2016; 1: 1-10.
- [3] Prathap PJ. Comparative analysis of opportunistic routing protocols for underwater acoustic sensor networks. In: *International Conference on Emerging Technological Trends (ICETT)*; Kollam, India; 2016. pp. 1-5.
- [4] Borges LM, Velez FJ, Lebres AS. Survey on the characterization and classification of wireless sensor network applications. *IEEE Communications Surveys & Tutorials* 2014; 16(4): 1860-1890.
- [5] Vukobratovic D, Stankovic L, Stankovic V. Performance analysis of node cooperation with network coding in wireless sensor networks. In: *4th IFIP International Conference on New Technologies, Mobility and Security*; Paris, France; 2011. pp. 1-4.
- [6] Stankovic L, Stankovic V. Cooperative network-coding system for wireless sensor networks. *IET communications* 2012; 6(3): 344-352.
- [7] Kumar S, Chaurasiya VK. A strategy for elimination of data redundancy in internet of things (IoT) based wireless sensor network (wsn). *IEEE Systems Journal* 2018; 13(2): 1650-1657.
- [8] Wan S, Gu Z, Ni Q. Cognitive computing and wireless communications on the edge for healthcare service robots. *Computer Communications* 2019; 149(1): 99-106.
- [9] Wan S, Zhao Y, Wang T, Gu Z, Abbasi QH, Choo KKR. Multi-dimensional data indexing and range query processing via Voronoi diagram for internet of things. *Future Generation Computer Systems* 2019; 91(1): 382-391.
- [10] Salma B, Youssef B, Abderrahim H. Software Defined Networking Based for Improved Wireless Sensor Network. In: Ezziyyani M (editors). *International Conference on Artificial Intelligence and Symbolic Computation*. Cham, Switzerland: Springer, 2019, pp. 246-258.
- [11] Wee J, Choi JG, Pak W. Wildcard fields-based partitioning for fast and scalable packet classification in vehicle-to-everything. *Sensors* 2019; 19(11): 2563.
- [12] Attar H, Alhihi M, Zhao B, Stankovic L. Network Coding Hard and Soft Decision Behavior over the Physical Payer Using PUMTC. In: *International Conference on Advances in Computing and Communication Engineering (ICACCE)*; Paris, France; 2018, pp. 471-474.
- [13] Prathap PJ. Opportunistic routing with virtual coordinates to handle communication voids in mobile ad hoc networks. In: Thampi S, Bandyopadhyay S, Krishnan S, Li KC, Mosin S, Ma M (editors). *Advances in Signal Processing and Intelligent Recognition Systems*. Cham: Springer, 2016, pp. 323-334.
- [14] Rabileh AA, Bakar KAA, Mohamed RR, Mohamad MA. Enhanced buffer management policy and packet prioritization for wireless sensor network. *International Journal on Advanced Science, Engineering and Information Technology* 2018; 8(4): 1770-1776.
- [15] Chithaluru P, Prakash R, Srivastava S. WSN Structure Based on SDN. In: Dumka A (editors). *Innovations in Software-Defined Networking and Network Functions Virtualization*. India: IGI Global, 2018, pp. 240-253.
- [16] Rao VS, Dakshayini M. SDN Based Energy Efficient Mechanism for Constrained Networks. *Procedia computer science* 2018; 143(1): 341-348.



- [17] Wan S, Li X, Xue Y, Lin W, Xu X. Efficient computation offloading for Internet of Vehicles in edge computing-assisted 5G networks. *The Journal of Supercomputing*; 2019; 1-30.
- [18] Inoue T, Mano T, Mizutani K, Minato Si, Akashi O. Fast packet classification algorithm for network-wide forwarding behaviors. *Computer Communications* 2018; 116(1): 101-117.
- [19] Abbasi M, Tahouri R, Rafiee M. Enhancing the performance of the aggregated bit vector algorithm in network packet classification using GPU. *PeerJ Computer Science* 2019; 5(1): e185.
- [20] Taylor DE. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)* 2005; 37(3): 238-275.
- [21] Rafiee M, Abbasi M. Pruned Kd-tree: a memory-efficient algorithm for multi-field packet classification. *SN Applied Sciences* 2019; 1(12): 1537.
- [22] Nayyar A, Mahapatra B. Effective Classification and Handling of Incoming Data Packets in Mobile Ad Hoc Networks (MANETs) Using Random Forest Ensemble Technique (RF/ET). In: Sharma N, Chakrabarti A, Balas V (editors). *Data Management, Analytics and Innovation*. Singapore: Springer, 2020, 1016(1): pp. 431-444.
- [23] Gao Z, Xuan HZ, Zhang H, Wan S, Choo KKR. Adaptive fusion and category-level dictionary learning model for multi-view human action recognition. *IEEE Internet of Things Journal* 2019; 6(6): 9280-9293.
- [24] Rajkamal R, Ranjan PV. Packet classification for network processors in wsn traffic using ann. In: 6th IEEE International Conference on Industrial Informatics; Daejeon, South Korea; 2008, pp. 707-710.
- [25] Xianyang F. Design and implementation of QoS routing protocol for wireless sensor network. In: Fourth International Conference on Digital Manufacturing & Automation; Qingdao, China; 2013, pp. 428-431.
- [26] Letswamotse BB, Malekian R, Chen CY, Modieginnyane KM. Software defined wireless sensor networks (SDWSN): a review on efficient resources, applications and technologies. *Journal of Internet Technology* 2018; 19(5): 1303-1313.
- [27] Farhan L, Kharel R, Kaiwartya O, Hammoudeh M, Adebisi B. Towards green computing for Internet of things: Energy oriented path and message scheduling approach. *Sustainable Cities and Society* 2018; 38(1): 195-204.
- [28] Modieginnyane KM, Letswamotse BB, Malekian R, Abu-Mahfouz AM. Software defined wireless sensor networks application opportunities for efficient network management: A survey. *Computers & Electrical Engineering* 2018; 66(1): 274-287.
- [29] Kim CG, Kim JG, Lee DH. Optimizing image processing on multi-core CPUs with Intel parallel programming technologies. *Multimedia tools and applications* 2014; 68(2): 237-251.
- [30] Reinders J. *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media Inc, 2007.
- [31] Zhou S, Qu YR, Prasanna VK. Multi-core implementation of decomposition-based packet classification algorithms. In: Malyshekin V (editors). *International Conference on Parallel Computing Technologies*. Berlin: Springer, 2013, pp. 105-119.
- [32] Wu X, Taylor V. Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore supercomputers. *Journal of Computer and System Sciences* 2013; 79(8): 1256-1268.
- [33] Memeti S, Pllana S. PAPA: A parallel programming assistant powered by IBM Watson cognitive computing technology. *Journal of computational science* 2018; 26(1): 275-284.
- [34] Abbasi M, Rafiee M, A calibrated asymptotic framework for analyzing packet classification algorithms on GPUs. *The Journal of Supercomputing* 2019; 6574-6611.
- [35] Rafiee M, Abbasi M, Nassiri M, An Efficient Method for Parallel Implementation of H-Trie Packet Classification Algorithm on GPU. *Tabriz Journal of Electrical Engineering* 2016; 46(3): 181-196.
- [36] Lim H, Lee S, Swartzlander Jr. A new hierarchical packet classification algorithm. *Computer Networks* 2012; 56(13): 3010-3022.

- [37] Batty M, Memarian K, Owens S, Sarkar S, Sewell P. Clarifying and compiling C/C++ concurrency: from C++ 11 to POWER. *ACM SIGPLAN Notices* 2012; 47(1): 509-520.
- [38] Stroustrup B. *A Tour of C++*. Addison-Wesley Professional, 2013.
- [39] Wolf F, Psaroudakis I, May N, Ailamaki A, Sattler KU. Extending database task schedulers for multi-threaded application code. In: *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*: ACM; La Jolla, California; 2015, pp. 1-12.
- [40] Kim W, Voss M. Multicore desktop programming with intel threading building blocks. *IEEE software* 2010; 28(1): 23-31.
- [41] Pong F, Tzeng NF. HaRP: rapid packet classification via hashing round-down prefixes. *IEEE Transactions on Parallel & Distributed Systems* 2010; 22(7): 1105-1119.
- [42] Qu YR, Zhang HH, Zhou S, Prasanna VK. Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU. In: *Architectures for Networking and Communications Systems (ANCS): ACM/IEEE Symposium*; Oakland, CA, USA; 2015, pp. 87-98.
- [43] Taylor DE, Turner JS. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking (ton)* 2007; 15(3): 499-511.
- [44] Varvello M, Laufer R, Zhang F, Lakshman T. Multilayer packet classification with graphics processing units. *IEEE/ACM Transactions on Networking* 2016; 24(5): 2728-2741.
- [45] Deng Y, Jiao X, Mu S, Kang K, Zhu Y. NPGPU: Network Processing on Graphics Processing Units. In: Zhou Q. (editors). *Theoretical and Mathematical Foundations of Computer Science*. Singapore: Springer, 2011, pp. 313-321.
- [46] Kang K, Deng YS. Scalable packet classification via GPU metaprogramming. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*; Grenoble, France; 2011, pp. 1-4.
- [47] Zhou S, Singapura SG, Prasanna VK. High-Performance Packet Classification on GPU. In: *High Performance Extreme Computing Conference (HPEC)*; Waltham, MA, USA; 2014, pp. 1-6.
- [48] Zheng J, Zhang D, Li Y, Li G. Accelerate Packet Classification Using GPU: A Case Study on HiCuts. In: Park J, Stojmenovic I, Jeong H, Yi G (editors). *Computer Science and its Applications*. Berlin: Springer, 2015, pp. 231-238.
- [49] Abbasi M, Fazel SV, Rafiee M. MBitCuts: optimal bit-level cutting in geometric space packet classification. *The Journal of Supercomputing* 2019; 1-24.
- [50] Khezrian N, Abbasi M, Comparison of the performance of skip lists and splay trees in classification of internet packets. *PeerJ Computer Science* 2019; 5(1): e204.