IET Computers & Digital Techniques

Research Article

IET Journals
The Institution of Engineering and Technology

Efficient parallelisation of the packet classification algorithms on multi-core central processing units using multi-threading application program interfaces

ISSN 1751-8601
Received on 26th April 2019
Revised 3rd June 2020
Accepted on 6th July 2020
E-First on 10th September 2020
doi: 10.1049/iet-cdt.2019.0118
www.ietdl.org

Mahdi Abbasi[1] ✉, *Milad Rafiee*[1]

[1]*Department of Computer Engineering, Engineering Faculty, Bu-Ali Sina University, Hamedan, Iran*

✉ *E-mail: abbasi@basu.ac.ir*

Abstract: The categorisation of network packets according to multiple parameters such as sender and receiver addresses is called packet classification. Packet classification lies at the core of Software-Defined Networking (SDN)-based network applications. Due to the increasing speed of network traffic, there is an urgent need for packet classification at higher speeds. Although it is possible to accelerate packet classification algorithms through hardware implementation, this solution imposes high costs and offers limited development capacity. On the other hand, current software methods to solve this problem are relatively slow. A practical solution to this problem is to parallelise packet classification using multi-core processors. In this study, the Thread, parallel patterns library (PPL), open multi-processing (OpenMP), and threading building blocks (TBB) libraries are examined and implemented to parallelise three packet classification algorithms, i.e. tuple space search, tuple pruning search, and hierarchical tree. According to the results, the type of algorithm and rulesets may influence the performance of parallelisation libraries. In general, the TBB-based method shows the best performance among parallelisation libraries due to using a theft mechanism and can accelerate the classification process up to 8.3 times on a system with a quad-core processor.

1 Introduction

Today, due to the increase in network users as well as in the traffic volume of communication lines, the need for more efficient networks has increased. One of the proposed methods for this purpose is to increase the speed of network communication lines. As a result, network communication has reached speeds above Terabit per second. To achieve optimum operational performance, routers and switches should also be able to route and process packets at the same speed. In addition to routing and processing, the quality of services requires special operations on packets. Thus, routers have come to need a set of new mechanisms for reserving resources, queuing and fair scheduling to better provide these services to users. One such mechanism is packet classification. The distribution of network packets into various streams in Internet routers and switches is called packet classification. Packet classification algorithms are required on all network devices that perform certain operations on packets at higher speeds. This need is particularly important, especially in processing devices such as routers that are used on Internet highways. Packet classification mechanisms are used by a wide range of packet processing devices including routers, firewalls, intrusion detection systems, user account management systems, and network administration systems [1].

One of the challenges in designing a router is to develop an optimum packet separation engine. Various algorithms have been so far proposed for this purpose. Packet classification algorithms can be generally implemented on both hardware and software frameworks. Despite their higher speeds, hardware methods have certain disadvantages. These disadvantages include high costs, excessive energy consumption, non-expandability, and inefficient utilisation of storage space. These negative aspects have shifted the focus to software-based classification methods which have great potential for customisation and expansion [2, 3]. A large number of software-based classification algorithms have been developed in recent years. Taylor classifies these algorithms into four main groups, i.e. linear search, decomposition, decision tree, and tuple space [4]. The main drawback of these algorithms, however, is their classification speed.

This paper focuses on the tuple space search (TSS) algorithm, tuple pruning search (TPS) algorithm, and hierarchical tree (H-tree) algorithm. These algorithms have greatly reduced the amount of memory consumption, but they face the problem of speed like other software-based algorithms. In this study, we try to increase the speed and efficiency of these algorithms. One way to speed up packet classification algorithms is to parallelise them via multi-core processors. On multi-core processors, tasks can be divided among the cores. Various application program interfaces such as open multi-processing (OpenMP) and threading building blocks (TBB) have so far been produced for code parallelisation in different programming languages [5, 6]. Another way to parallelise the classification is to use platforms like Compute Unified Device Architecture (CUDA) on many-core platforms like graphics processing units (GPUs) [7–15]. The complexity of GPU-based parallelisation platforms for packet classification has been fully studied [16]. In this paper, from among the various libraries developed to parallelise algorithms on multi-core systems, we investigate Thread, parallel patterns library (PPL), OpenMP, and TBB libraries for parallel implementation of packet classification algorithms. The contributions of this study are as follows:

- In this paper, the packet classification algorithms of TSS, TPS, and H-tree are examined regarding parallelisation on multi-core systems.
- The parallel versions of these algorithms were implemented on multi-core processors using Thread, PPL, OpenMP, and TBB libraries.
- Parallelisation libraries are evaluated according to the type of packet classification algorithm, set of rules, and packages in terms of different efficiency measures, including speedup and throughput, and the best of them are specified.

The rest of the paper is organised as follows. In the second section, the structure of the packet classification algorithms of TSS, TPS, and H-tree is examined. The third part introduces the parallelisation libraries Thread, PPL, OpenMP, and TBB, and the next section reviews the related works conducted in this field. The proposed method for parallelising the algorithms used in this paper

© The Institution of Engineering and Technology 2020

Table 1 Example rule set

Rule no.	Source prefix	Destination prefix	Source port	Destination port	Protocol	Tuple
R 0	1010*	01*	0–65535	25	6	(4,2)
R 1	10001*	0111*	53	443	4	(5,4)
R 2	0*	1110*	53	1024–65535	17	(1,4)
R 3	0*	1100*	53	443	4	(1,4)
R 4	111*	1110*	53	25	4	(3,4)
R 5	1110*	*	0–65535	2788	17	(4,0)
R 6	1*	10*	53	5632	6	(1,2)
R 7	*	1*	53	25	6	(0,1)
R 8	*	10*	0–65535	2788	17	(0,2)

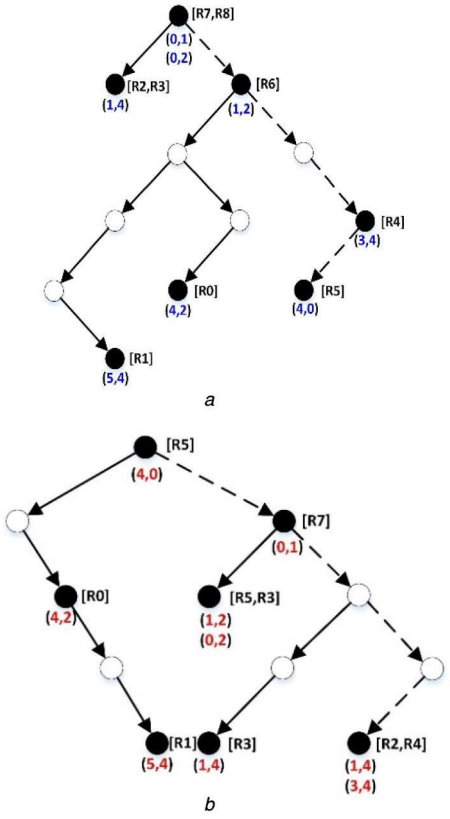

Fig. 1 *Binary trees created via the source and destination IP address fields of Table 1*
(a) Source IP address, ***(b)*** Destination IP address

as well as the evaluation of this method is addressed in Section 5. In the final section, some conclusions are made regarding the implementation of various rulesets on multi-processor systems using different libraries.

2 Relevant tools and algorithms

This section describes the structure of the classification algorithms of TSS, TPS, and H-tree.

2.1 Tuple space search algorithm

A tuple space maps each k-dimensional rule onto an ordered k-tuple so that the ruleset mapped onto an ordered k-tuple will have a constant and fixed length. TSS algorithm was introduced by Srinivasan *et al.* [17]. This method reduces the number of required searches by a factor of four to seven in comparison with the number of searches required by linear search in a ruleset. This basic technique has a relatively good performance for large rulesets given the properties of large sets of filters. The reason for developing this method was cased in which no address had more than six prefix matches in the routing table. Therefore, a technique was introduced to reduce the number of spaces that needed to be thoroughly searched [17]. The key idea in the TSS algorithm is the small number of separate tuples compared with the total number of rules. Therefore, in this algorithm, the set of rules is first divided into tuples. A tuple represents the number of bits specified in each rule field. To explain the steps of the algorithm, the rules in Table 1 are used. Each rule is defined by five fields, i.e. the prefix of source IP address, the prefix of destination IP address, source port number, destination port number, and protocol. The tuple corresponding to each rule is shown in the last column of the table.

The binary trees created via the source and destination IP address fields of Table 1 are shown in Fig. 1. The black nodes represent the rules, next to which are their corresponding tuples.

Following is how this algorithm works with an incoming packet with the tuple (11100, 11101, 53, 25, 4) whose fields correspond to the fields of the ruleset in Table 1. In this example, the search is initially done based on the fields of the source address and the destination address in the corresponding binary tree. The search path in Fig. 1 is marked with broken lines. Rules corresponding to the traversed tuples are extracted. The traversed rules are R2, R4, R5, R6, R7, and R8. As can be seen, the number of extracted rules is smaller than the entire set of rules. This method will result in a significant elimination of rules in a large-scale ruleset. Finally, the best matching rule, if any, is determined by applying a linear search on the extracted rules of the previous step. In the linear search, the rest of the fields of the packet and the rules are compared together. If a rule is found to be matching the packet, it will be considered as

© The Institution of Engineering and Technology 2020

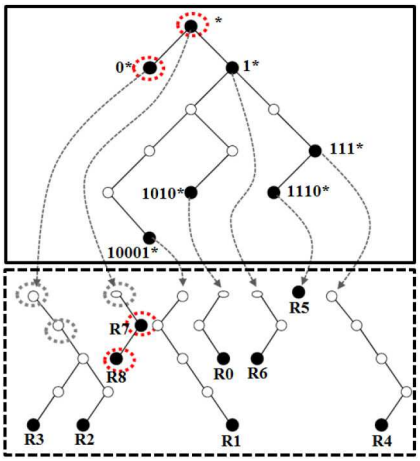

Fig. 2 *Structure of H-tree algorithm*

the best rule. For example, the best matching rule here is Rule 4. If the packet matches several rules, the rule with the highest priority will be selected as the final result. The priority of the rules in this study is based on the order in which they appear in Table 1.

2.2 Tuple pruning search algorithm

TSS algorithm performed a linear search on all the tuples derived from the traversal of the tree of source and destination addresses. However, the TPS algorithm first extracts the intersection of the derived tuples. Then it performs a linear search on the rules obtained from the shared tuples [18]. When applied to the above example, this will provide us with R4, R5, and R7. A comparison of the number of the rules obtained by these two algorithms in this example clearly shows that the classification time of the TPS algorithm is remarkably shorter than that of the TSS. The performed evaluations strongly indicate this fact.

2.3 Hierarchical tree algorithm

One of the algorithms based on decision trees is the H-tree algorithm. In the H-tree algorithm, the decision tree for classification of incoming packets is built by using the source and destination IP addresses. As an example, the tree formed in Fig. 2 can be used according to the nine rules in Table 1.

To create a tree, first, the source IP address of the rules is read bit by bit. On reading a 0 or 1, the tree is traversed on the left or right, respectively. When the wildcard sign (*) is seen, the formation of the tree based on the source IP address of that rule is completed. Therefore, a pointer is used to continue the tree using the destination IP address. Building the tree through the destination IP address exactly resembles building the tree through the source IP address [19].

When an incoming packet is received, the source and destination IP addresses, the source and destination ports, and the protocol are initially extracted from the packet. For example, in a packet with the fields (00000, 10000, 542, 2788, 6), traversal of the tree starts from the root based on the bits of the source IP address field. If a node in the traversal path has a pointer to the tree of the destination IP address, it will be placed into a queue. When the traversal of the tree of the source IP address is finished, the nodes in the queue are traversed based on the destination address fields of the packet. All the rules in the traversed path are stored and then searched linearly to select the best matches. In this example, the matching rule is R8 [19].

3 Parallelisation libraries

3.1 Thread

One of the easiest ways to implement parallelisation on multi-core processors is to use the C++ standards. C++ 11 is a shared-memory-concurrency language with explicit thread creation and implicit sharing. That is 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 parallelisation. 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 [20, 21].

3.2 Parallel patterns library

Like Thread, PPL is a library provided by Microsoft. PPL is based on resource management and concurrent scheduling. This feature increases the level of abstraction between the application code and the mechanism of thread processing by providing type-safe and general algorithms as well as containers that act on the data in a parallel manner. This library provides the following features:

- *Parallel task*: A mechanism based on which Windows Thread Pool can run multiple tasks in parallel.
- *Parallel algorithms*: General algorithms that act in parallel to operate on data sets based on runtime concurrency.
- *Parallel objects and containers*: Public containers that provide concurrent secure access to their elements.

PPL provides functions that work synchronously on a data set. For example, *concurrency*::*parallel_for* function is used for parallel loop iterations. *Parallel_for* function divides the execution of tasks according to the number of available computing resources. Many *for* loops can be converted to *parallel_for*. However, the *parallel_for* algorithm is sometimes different from the simple *for* loop. For example, *parallel_for* does not perform tasks in a predefined order. The iterations of *parallel_for* should have a forward direction [22].

3.3 OpenMP

OpenMP is a shared memory programming interface in C, C++, and Fortran on all major compilers and platforms. The library includes parallelisation patterns such as parallel blocks, loop, and tasks supported by basic concepts like data sharing and synchronisation. Parallel algorithms can be expressed in simple, compact pieces of code to improve productivity. Also, OpenMP provides a more abstract interface than many common schedulers used in applied programming. This asset is a great advantage for the implementation of scientific and analytical algorithms. The underlying concept of this library is the Fork/Join model for parallelisation which is shown in Fig. 3.

The command used to interpret parallel blocks is #*pragma opm parallel* [23].

© The Institution of Engineering and Technology 2020

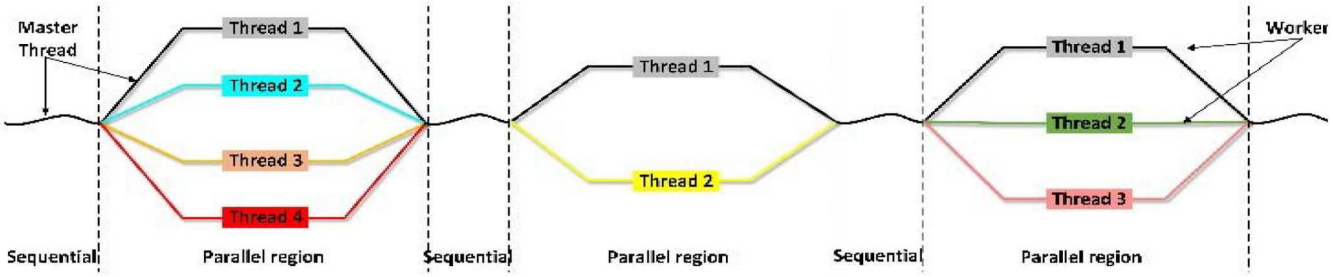

Fig. 3 *OpenMP parallelisation model*

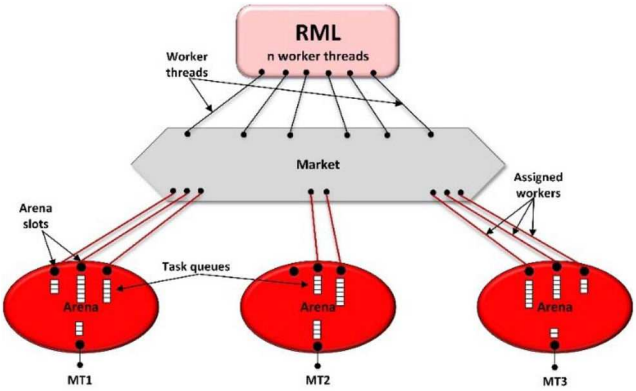

Fig. 4 *Components of TBB's task scheduler*

In this model, each parallel block has a master thread, several 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 synchronise 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 synchronise 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 the execution step 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 [23].

3.4 Threading building blocks

TBB was first introduced by Intel in 2006 as a parallel programming library. Fig. 4 illustrates the general structure of TBB and how it works to create threads and balance their workloads. This library allows parallelisation to be interpreted both explicitly and implicitly. In 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.

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 central processing unit (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. Several slots are assigned to each arena which represent the number of worker threads that 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, the 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 running 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 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 failure of a worker thread in stealing exceeds 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 [24, 25].

OpenMP and TBB as two potential tools, aid the programmer to deploy parallel programs without any manual thread organisation.

OpenMP helps to parallelise the code in the compile time. Wide-spread compilers support OpenMP specification. The chief advantage of TBB to OpenMP is its superior method for load balancing. While the latter parallelise a loop by inserting a parallel for pragma, the former requires to embrace the loop in a function object, as the task body. TBB has other clear advantages over OpenMP. First, unlike OpenMP, it provides a higher level of abstraction by working with any thread-safe data structure. Secondly, more complex parallelism is allowed with TBB as compared to OpenMP. While the programmer can freely use the TBB task scheduler directly to generate its parallel forms, the OpenMP is restricted to the built-in paradigms.

OpenMP provides three complex scheduling techniques to a programmer including static, guided, and dynamic. TBB replaces those with a superior divide-and-conquer method. OpenMP exploits task schedulers based on the work-first and breadth-first strategies. With the former, tasks are executed when they are formed, while with the latter, all tasks are first formed [26].

In C++11, tasks and threads are generated by using std: async and std::thread, respectively. Note that, while in the task-level parallelism, runtime library orchestrates tasks and balances the loads, in thread-level parallelism the programmer should control the load balancing. The runtime system for low-level programming model of C++11 is simpler than that of OpenMP and TBB. The

© The Institution of Engineering and Technology 2020

```
Input: Rules R, headers H
Output: RulesIndexArray
1: construct tree based on R
2: function PClassification_ThreadLib
        (tree, RulesIndexArray, H, R, index)
3:     for all i ∈ ⌈|H| / n_threads⌉ do
4:         RulesIndexArray ← Classify(tree, R, H, i, index)
5:     end for
6: end function

7: n_threads ← number of defined threads
8: vector < thread > threads
9: for all index ∈ n_threads do
10:    threads.push_back(thread(PClassification_ThreadLib,
                tree, RulesIndexArray, H, R, index))
11: end for
```

Fig. 5 *Algorithm 1: implementation of parallel packet classification based on Thread library*

```
Input: Rules R, headers H
Output: RulesIndexArray
1: construct tree based on R
2: parallel_for (0, |H|, [&](int index)
3:     RulesIndexArray ← Classify(tree, R, H, index)
4: end parallel_for
```

Fig. 6 *Algorithm 2: implementation of parallel packet classification based on PPL*

```
Input: Rules R, headers H
Output: RulesIndexArray
1: construct tree based on R
2: #pragma omp parallel for
3: for all index ∈ |H| do
4:     RulesIndexArray ← Classify(tree, R, H, index)
5: end for
```

Fig. 7 *Algorithm 3: implementation of parallel packet classification based on OpenMP library*

programmer can efficiently exploit the available hardware resources directly [27].

The PPL library is simpler than C++11. It provides C++ templates for writing parallel algorithms. This makes it more communicative and flexible [28].

4 Related works

In this section, we shall review studies that have parallelised packet classification algorithms on multi-core processors. In 2011, Pong and Tzeng [29] parallelised the algorithm based on Hashing Round-Down Prefixes (HaRP) and HyperCuts tree-based algorithm by using multi-core processors. Although the parallelisation library used in their research is not specified, their results at best represent a rate of 11.74 and 4.01 million packets per second for HaRP and HyperCuts algorithms, respectively. Zhou and colleagues [30] implemented in 2013 two packet classification algorithms called linear search algorithm and domain tree search algorithm on a multi-core processor. In their study, the algorithms were implemented using the Pthread parallelisation library on a 16-core processor. The maximum throughput obtained in this study was about 11.5 Gbps.

In 2015, Qu *et al.* [31] implemented the bit-vector algorithm, which is a decomposition-based algorithm, in a parallel manner on a multi-core processor. Decomposition-based algorithms have two phases of search and a combination of results. They used the OpenMP library to parallelise the algorithm. The maximum throughput obtained in this study was about 15 million packets per second.

A review of previous studies shows that the parallel version of TSS, TPS, and H-tree algorithms has not been so far implemented on multi-core processors. Also, despite the extensive capabilities of the TBB library compared to other parallelisation libraries, none of the studies have made use of the TBB library to parallelise classification algorithms on multi-core processors. Therefore, the present study for the first time implements and compares the performance of packet classification algorithms using C++ parallelisation libraries thread, OpenMP, and TBB. In the next section, we examine the parallelisation of the TSS, TPS, and H-tree algorithms using the parallelisation libraries which were mentioned in Section 3.

5 Implementation and evaluation

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 an increase in the number of packets increases the time of classification linearly. Therefore, 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 multi-core 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 control. Needless to say, this dependency would reduce the efficiency of parallelisation. To solve this problem, we parallelise 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. In the proposed parallelisation method, therefore, packets are distributed among cores that share a copy of the classifying tree and perform their tasks independently.

Algorithm 1 (see Fig. 5) shows the pseudo-code of parallel packet classification using the Thread library. The inputs of this algorithm are rules, R and the header of the packets, H. The output of the algorithm is the array *RulesIndexArray*. Each index of this array is used to store the result of classifying a packet. First, the algorithm reads the ruleset and creates a tree according to one of the classification algorithms naming TSS, TPS, and H-tree (line 1). Lines 2 to 6 represent a function that receives parameters including tree structure, array to store the result, packets, rules, and the thread index, and then classifies the specified packets by that thread. In the main body of the program, the number of threads is first specified (line 7), then a vector of threads (line 8) is created. Finally, for all threads, the packet classification function is called (lines 9 to 11).

Algorithm 2 (see Fig. 6) represents the pseudo-code for parallel implementation using the PPL library. In this algorithm, the input, output, and tree layout are similar to Algorithm 1. To parallelise the classification, once the tree is constructed, the packets are divided between threads and each thread classifies a certain number of packets. In this library, the *Parallel_for* is used to parallelise the packet classification operation (lines 2 to 4). The *Parallel_for* schedules the tasks optimally for parallel processing.

Algorithm 3 (see Fig. 7) shows the pseudo-code for parallel packet classification using the OpenMP library. The input, output, and tree construction in this algorithm are similar to Algorithm 1. In this algorithm, as in the previous algorithms, the packets are divided between threads for parallel classification. The command $\#pragmaompparallelfor$ schedules the classification operation (lines 3 to 5) between multiple threads running concurrently. Each thread classifies a number of packets.

Algorithm 4 (see Fig. 8) shows the pseudo-code of using the TBB library for parallel packet classification. The input of this algorithm is the rules, packets, and tree structure and its output is an array for saving results. The first line of Algorithm 4 shows the construction of the tree.

The $PClassification\_TBBLib$ is actually a class that allows the execution of the $parallel\_for$ on input tasks (lines 2 to 13). The $parallel\_for$ is executed using the *auto_partitioner* algorithm provided by the TBB library; this way, the correct splitting of the *Task* array in chunks of basic tasks, which are assigned to the

© The Institution of Engineering and Technology 2020

executor threads, is left to the run-time support. In line 14, the algorithm calls the packet classification function to concurrently classify the packet.

5.1 Implementation

To implement the aforementioned algorithms, eight thread processes were executed on a multi-core processor. Each process, according to its index, selects and classifies a number of incoming packets.

The proposed methods were implemented on a multi-core system with Intel Core i7-3770K processor, 3500 MHz clock speed, 32 GB DDR3 RAM, four cores, and eight threads. The Classbench tool was used to generate a ruleset and experimental packets [32]. This tool generates the dummy rules required for the classifier as well as the corresponding dummy packets that are required as test input packets. This tool meets the needs of the developers of packet classification algorithms to factual and heterogeneous rules used in firewalls, IP chains, and access control lists. In this study, we produced a set of rules corresponding to the ACL2, FW2, and IPC2 parameters containing 1k rules with 1k, 4k, 8k, and 16k packets to assess the proposed method.

5.2 Results

The first evaluation criterion examined in this paper is the time of packet classification by parallel algorithms with different parallelisation libraries and different rulesets. Table 2 shows the results of packet classification time using Thread, PPL, OpenMP, and TBB libraries as well as the serial version. Classification time is in milliseconds.

Input : Rules R, headers H
Output : $RulesIndexArray$

```
1: construct tree based on R
2: class PClassification_TBBLib {
3:     TreeStructure *tree
4:     OutputStructure RulesIndexArray
5:     HeaderStructure H
6:     RuleStructure R
7:     public:
8:     void operator() ( const blocked_range < int > & r ) const
9:         for all index ϵ r do // iterates over the entire chunk
10:            RulesIndexArray ← Classify(tree, R, H, index)
11:        end for
12:    end void
13: };
14: parallel_for(blocked_range < int
    > (0, |H|), PClassification_TBBLib(tree, RulesIndexArray, H, R)
```

Fig. 8 *Algorithm 4: implementation of parallel packet classification based on TBB library*

As can be seen from the results, parallelisation libraries in most cases have reduced the classification time compared to the serial version. In some instances of the parallelisation of the H-tree algorithm, the Thread library has increased packet classification time in comparison with the serial version.

In fact, in cases where the number of available packets 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. In FW and ACL rulesets, this library has spent more time than the serial version on the classification of 1k and 4k packets. For different numbers of incoming packets, packet classification based on the IPC ruleset required more time than other rules. As the H-tree created for this ruleset is larger than the trees corresponding to ACL and FW rulesets, the implementation of packet classification with this ruleset is less complicated than with other rulesets.

Another evaluation criterion used in this paper is throughput. Throughput refers to the number of packets classified in the unit of time (seconds). Fig. 9 shows the throughput obtained by the implementation of parallelised TSS, TPS, and H-tree algorithms on ACL, FW, and IPC rulesets. In this figure, the unit of throughput is kilo-packets per second. The final criterion for the evaluation of parallelisation libraries is speedup. The speedup refers to the ratio of the time needed for parallel classification of packets to the time needed for sequential classification of packets. Speedup is denoted by S and is calculated by

$$S = \frac{\text{Computation time in the serial version}}{\text{Computation time in the parallel version}} \quad (1)$$

The speedup of TBB, OpenMP, PPL, and Thread libraries on different rulesets is shown in Fig. 10. The highest speedup rate belonged to TBB, which can be attributed to CPU cores not idling during the functioning of TBB. In TBB, if a core remains idle while other cores still have work to do, the scheduler balances the workload by stealing work from the busy core and assigning it to idle cores. In other parallelisation libraries, however, it is possible for a processor to remain idle. At the beginning of the experiment, OpenMP was expected to have the best results after TBB. However, this expectation was not fulfilled in all cases and, as the results show, the performance of OpenMP was weaker than Thread and PPL in some parallelisation tasks. These situations could be primarily explained by the fact that OpenMP detects by itself the parallel points of the program and operates based on *Fork-Join*. Thus, the main thread does not take control of the program until all the threads assigned to parallel tasks have finished their work. If

Table 2 Classification time (ms)

Ruleset	Library	TSS				TPS				H-tree			
		Number of packets											
		1k	4k	8k	16k	1k	4k	8k	16k	1k	4k	8k	16k
ACL	seq	17.6	64.2	134.4	261.7	4.7	18.6	39	76.9	2.92	7.48	23.71	34.47
	Thread	16.3	32.7	47.6	94.2	1.5	7.3	16.8	31.2	7.49	8.02	10.46	13.73
	PPL	15.2	31.7	47.8	78.3	1.2	2.7	9.9	16.8	1.4	5.18	7.53	11.85
	OpenMP	16	32.1	49.6	78.9	1.7	2.8	7.7	15.9	0.98	2.63	4.95	8.27
	TBB	14.7	31	47.2	63.5	0.9	2.2	7.4	15.7	0.8	2.18	4.3	8.01
FW	seq	24.2	101.1	185.7	390.6	9.1	34.3	65.2	132.8	0.78	3.27	6.24	10.14
	Thread	16.6	52	65.7	139.4	1.8	15.2	32.3	47.8	4.06	5.06	5.94	6.71
	PPL	15.9	47.2	63.4	125.5	1.3	7.8	16.4	32	0.94	2.34	3.75	5.94
	OpenMP	16.1	32	53.7	104.4	1.2	7.7	16.2	31.9	0.81	2.09	3.53	5.65
	TBB	15.8	31.9	46.1	93.3	1.1	7.2	15.9	31.4	0.62	1.56	2.96	4.17
IPC	seq	16.2	78.7	109.3	234.2	1.9	4.8	7.9	32.8	0.78	2.34	4.21	9.52
	Thread	15.9	31.8	47.9	79	1	2.7	7.1	13.4	4.37	5.77	8.12	10.53
	PPL	16	32.1	47.8	78.9	0.7	2.2	8.9	19.2	0.63	2.5	3.71	5.83
	OpenMP	15.2	31.7	47.6	78.8	0.61	2	6.4	12.7	0.57	1.34	3.14	3.93
	TBB	9.7	20.2	32.3	58.7	0.44	1.4	4.6	9.4	0.4	1.21	2.98	3.72

© The Institution of Engineering and Technology 2020

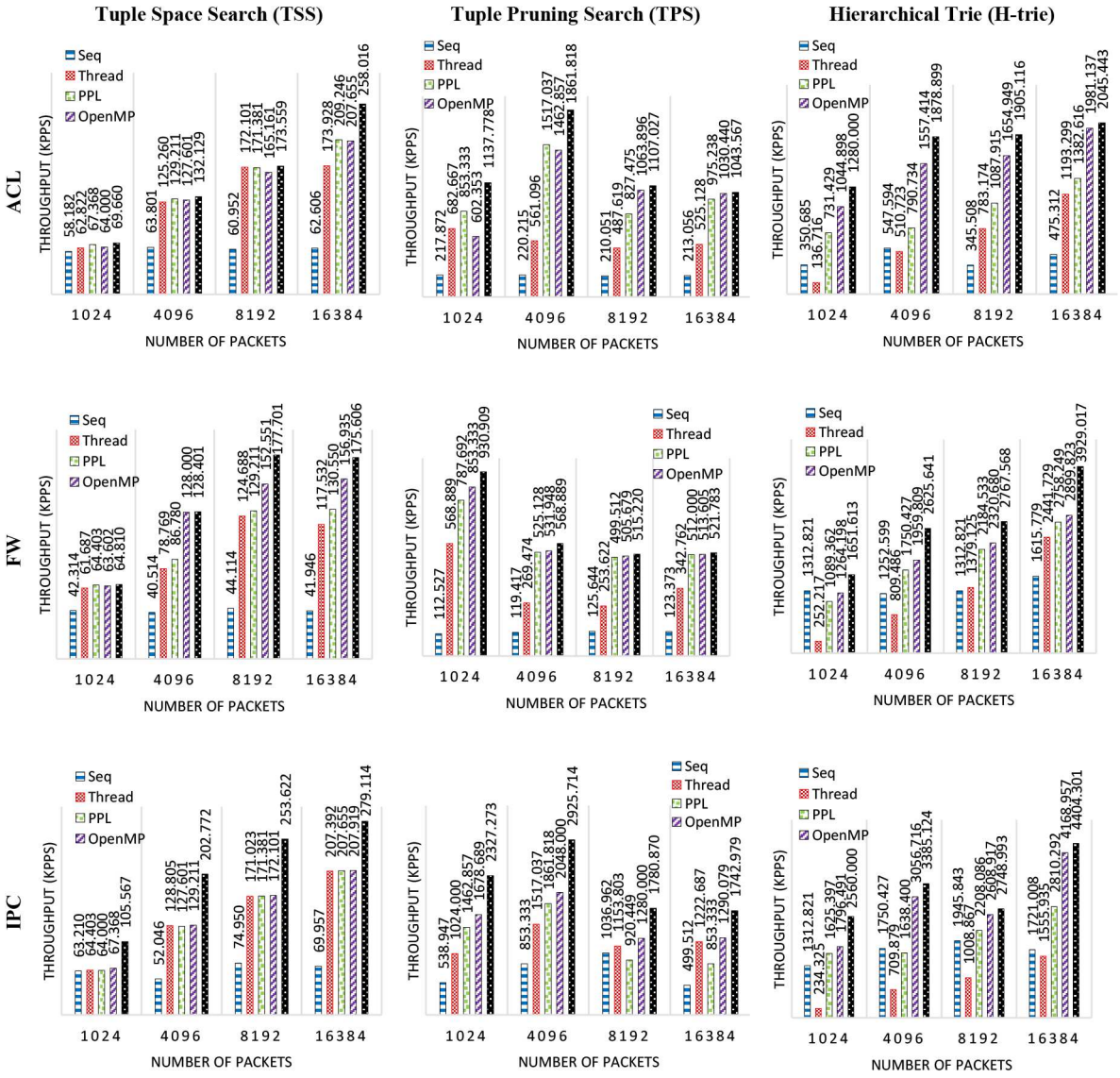

Fig. 9 *Result of implementation (throughput)*

only a thread does not finish its task, the main thread will not continue the program. Thread and PPL also work based on the threads and face the same problem. Therefore, as can be seen from the results, none of these three scenarios can be considered to be superior and, depending on the conditions of a program, one may outperform the others.

As can be seen from Figs. 9 and 10, the throughput and speedup of parallelisation for both algorithms using TBB, OpenMP, PPL, and Thread libraries have decreased 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. Decreased and increased throughput and speedup in all libraries for a specified number of packages confirm this explanation.

From among ACL, FW, and IPC rulesets, the highest throughput belongs to packet classification with IPC ruleset. The reason is that the tree structure created by this ruleset is smaller than those created by other rulesets. Compared to other parallel libraries, TBB library has obtained the highest amount of throughput and speedup for all rulesets, various numbers of packets, and different classification algorithms. The reason is that TBB automatically defines the execution of the loops through defined threads by using a *divide-and-conquer* method. Also, TBB becomes more efficient than other parallelisation libraries by using the technique of stealing tasks during the execution of threads. The maximum throughput obtained from parallelisation of IPC ruleset using TBB for TSS, TPS, and H-tree algorithms is 279.114, 2925.714, and 4404.301 kilo-packets per second, respectively. The highest speedup rate in our experiments was 8.273 which belonged to TBB.

5.3 Efficiency evaluation and comparison

Table 3 compares the efficiency of the TBB-based parallel packet classification algorithm with state-of-the-art multi-core methods. The efficiency of each method is computed according to the following equation:

$$\text{Efficiency} = \frac{\text{Maximum throughput}}{\text{Number of cores}} \tag{2}$$

The proposed method uses four cores, and the competitor methods use 16 cores. Nevertheless, the efficiency of the proposed method is the highest. Therefore, the proposed method can better use the resources of the multi-core CPUs for parallel packet classification.

6 Conclusion

Packet classification is a basic process in network processors. The most important issue in this domain of research is using an algorithm that can classify packets at speeds close to the speed of

© The Institution of Engineering and Technology 2020

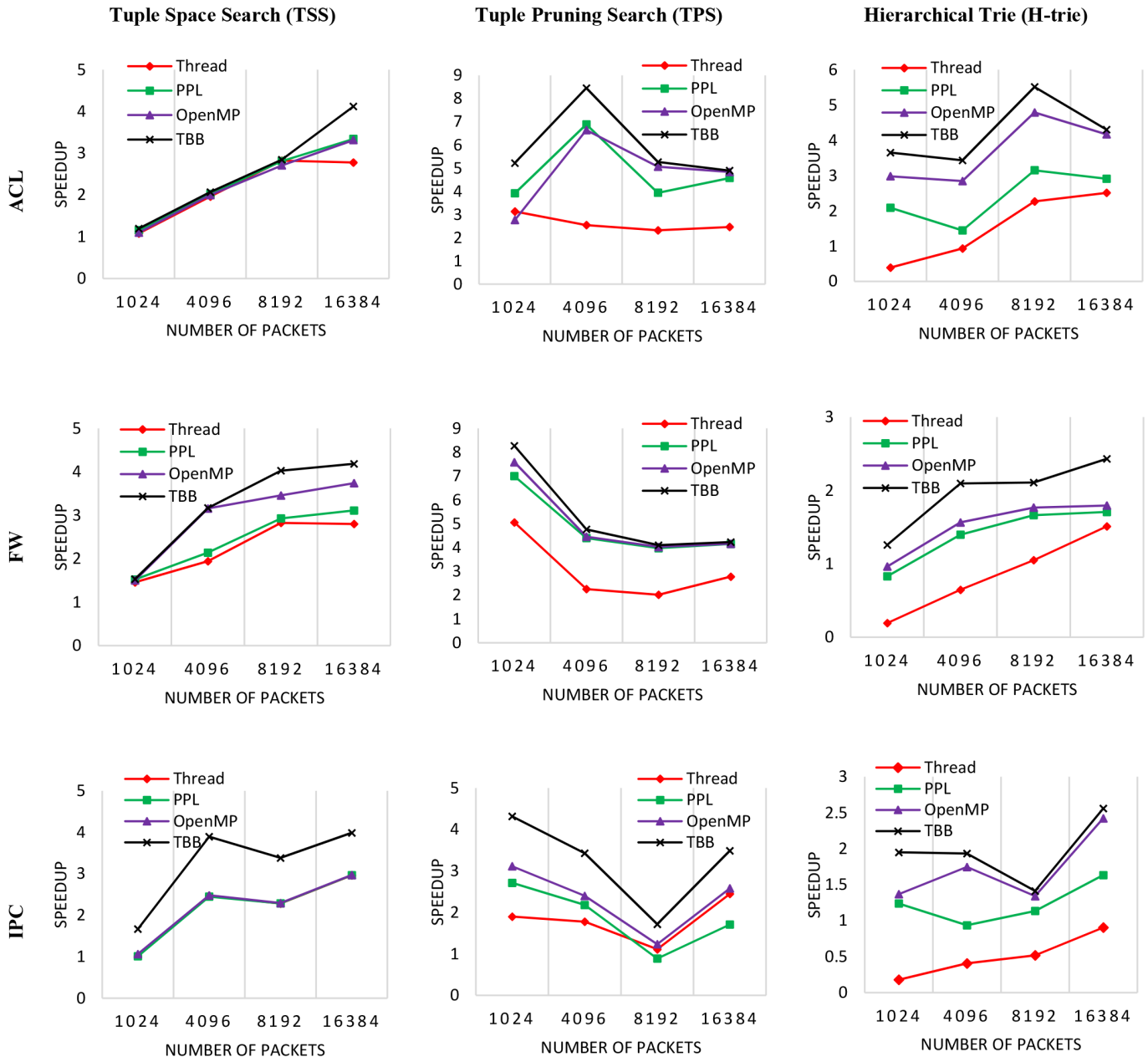

Fig. 10 *Results of implementation (speedup)*

Table 3 Comparing the efficiency of multi-core packet classification methods

Algorithm	Number of cores	Maximum throughput (MPPS)	Efficiency
HaRP [29]	16	11.74	0.734
HyperCuts [29]	16	4.01	0.251
decomposition-based [31]	16	15	0.937
our Method	4	4.301	1.075

the network. These algorithms should also optimise memory consumption. Existing methods have not been able to compromise between time and memory consumption. The focus of this study was on the algorithms of TSS, TPS, and H-tree. These algorithms can greatly reduce memory consumption, but their speed is relatively low. This study was an attempt to reduce packet classification time through parallel implementation of these three algorithms. For this purpose, we made use of Thread, PPL, OpenMP, and TBB parallelisation libraries. In this paper, various numbers of experimental packets were classified using these four libraries and the abovementioned algorithms. For our evaluations, we used the rulesets and experimental packets generated by the Classbench tool. The results of this study show that the parallel implementation of these algorithms significantly increases the speed of packet classification. Also, TBB showed better performance than other methods, and its speedup rate on a quad-core processor reached 8.3. To continue this line of research further, we can use distributed and clustered multiprocessor systems [33]. In this case, each of the distributed multiprocessors classifies a portion of the packets.

7 References

[1] Mythrei, S., Dharmaraj, R.: 'Packet classification based on standard access control list', *Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET)*, 2014, **3**, (1), pp. 172–175

[2] Dobrescu, M., Egi, N., Argyraki, K., *et al.*: 'Routebricks: exploiting parallelism to scale software routers'. Proc. of the ACM SIGOPS 22nd symp. on Operating Systems Principles, MT, USA, 2009

[3] Pao, D., Zhou, P., Liu, B., *et al.*: 'Enhanced prefix inclusion coding filter-encoding algorithm for packet classification with ternary content addressable memory', *IET Comput. Digit. Tech.*, 2007, **1**, (5), pp. 572–580

[4] Taylor, D.E.: 'Survey and taxonomy of packet classification techniques', *ACM Comput. Surveys (CSUR)*, 2005, **37**, (3), pp. 238–275

[5] Machado, R.S., Almeida, R.B., Jardim, A.D., *et al.*: 'Comparing performance of C compilers optimizations on different multicore architectures'. 2017 Int.

© The Institution of Engineering and Technology 2020

Symp. on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Campinas, Brazil, 2017
[6] Paulino, H., Marques, E.: 'Heterogeneous programming with single operation multiple data', *J. Comput. Syst. Sci.*, 2015, **81**, (1), pp. 16–37
[7] Zhou, S., Singapura, S.G., Prasanna, V.K.: 'High-performance packet classification on Gpu'. High Performance Extreme Computing Conf. (HPEC), MA, USA, 2014
[8] Zheng, J., Zhang, D., Li, Y., *et al.*: 'Accelerate packet classification using Gpu: a case study on hicuts', in '*Computer science and its applications*' (Springer, Germany, 2015), pp. 231–238
[9] Zhao, Y., Chen, L., Xie, G., *et al.*: 'Gpu implementation of a cellular genetic algorithm for scheduling dependent tasks of physical system simulation programs', *J. Comb. Optim.*, 2018, **35**, (1), pp. 293–317
[10] Nottingham, A., Irwin, B.: 'Gpu packet classification using opencl: a consideration of viable classification methods'. Proc. of the 2009 Annual Research Conf. of the South African Institute of Computer Scientists and Information Technologists, Vanderbijlpark, South Africa, 2009
[11] Maghazeh, A., Bordoloi, U.D., Dastgeer, U., *et al.*: 'Latency-aware packet processing on Cpu-Gpu heterogeneous systems'. 2017 54th ACM/EDAC/IEEE Design Automation Conf. (DAC), TX, USA, 2017
[12] Li, T.-H., Chu, H.-M., Wang, P.-C.: 'Ip address lookup using Gpu'. IEEE 14th Int. Conf. on High Performance Switching and Routing (HPSR), Taipei, Taiwan, 2013
[13] Kang, K., Deng, Y.S.: 'Scalable packet classification via Gpu metaprogramming'. Design, Automation & Test in Europe Conf. & Exhibition (DATE), Grenoble, France, 2011
[14] Deng, Y., Jiao, X., Mu, S., *et al.*: 'Npgpu: network processing on graphics processing units', in '*Theoretical and mathematical foundations of computer science*' (Springer, Germany, 2011), pp. 313–321
[15] Beyeler, M., Oros, N., Dutt, N., *et al.*: 'A Gpu-accelerated cortical neural network model for visually guided robot navigation', *Neural Netw.*, 2015, **22**, pp. 75–87
[16] Abbasi, M., Rafiee, M.: 'A calibrated asymptotic framework for analyzing packet classification algorithms on Gpus', *J. Supercomput.*, 2019, **75**, (10), pp. 6574–6611
[17] Srinivasan, V., Suri, S., Varghese, G.: 'Packet classification using tuple space search'. ACM SIGCOMM Computer Communication Review, NY, USA, 1999
[18] Wang, P.-C., Chan, C.-T., Tseng, W.-C., *et al.*: 'A fast packet classification by using enhanced tuple pruning', in '*Protocols for high speed networks*' (Springer, Germany, 2002), pp. 180–191
[19] Lim, H., Lee, S., Swartzlander, JrE.E.: 'A new hierarchical packet classification algorithm', *Comput. Netw.*, 2012, **56**, (13), pp. 3010–3022
[20] Batty, M., Memarian, K., Owens, S., *et al.*: 'Clarifying and compiling C/C++ concurrency: from C++11 to power', *ACM SIGPLAN Notices*, 2012, **47**, (1), pp. 509–520
[21] Stroustrup, B.: '*A tour of C++*' (Addison-Wesley Professional, England, 2013)
[22] Available at http://msdn.microsoft.com/, Date Accessed 2019
[23] Wolf, F., Psaroudakis, I., May, N., *et al.*: 'Extending database task schedulers for multi-threaded application code'. Proc. of the 27th Int. Conf. on Scientific and Statistical Database Management, CA, USA, 2015
[24] Iordan, A.C., Jahre, M., Natvig, L.: 'Tuning the victim selection policy of intel Tbb', *J. Syst. Archit.*, 2015, **61**, (10), pp. 584–591
[25] Kim, C.G., Kim, J.G., Lee, D.H.: 'Optimizing image processing on multi-core Cpus with intel parallel programming technologies', *Multimedia Tools Appl.*, 2014, **68**, (2), pp. 237–251
[26] Refsnes, P.R.: 'Comparison of Openmp and threading building blocks for expressing parallelism on shared-memory systems'. The University of Bergen, 2011
[27] Salehian, S., Liu, J., Yan, Y.: 'Comparison of threading programming models'. 2017 IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW), FL, USA, 2017
[28] Zhang, Y., Wang, H., Zhang, Y., *et al.*: 'An improved ant system algorithm based on PPL'. 2010 2nd Int. Conf. on Information Engineering and Computer Science, Hangzhou, China, 2010
[29] Pong, F., Tzeng, N.-F.: 'Harp: rapid packet classification via hashing round-down prefixes', *IEEE Trans. Parallel Distrib. Syst.*, 2011, **22**, (7), pp. 1105–1119
[30] Zhou, S., Qu, Y.R., Prasanna, V.K.: 'Multi-core implementation of decomposition-based packet classification algorithms'. Int. Conf. on Parallel Computing Technologies, St. Petersburg, Russia, 2013
[31] Qu, Y.R., Zhang, H.H., Zhou, S., *et al.*: 'Optimizing many-field packet classification on Fpga, multi-core general purpose processor, and Gpu'. 2015 ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS), Oakland, CA, USA, 2015
[32] Taylor, D.E., Turner, J.S.: 'Classbench: a packet classification benchmark', *IEEE/ACM Trans. Netw. (ton)*, 2007, **15**, (3), pp. 499–511
[33] Azarkhish, E., Loi, I., Benini, L.: 'A case for three-dimensional stacking of tightly coupled data memories over multi-core clusters using low-latency interconnects', *IET Comput. Digit. Tech.*, 2013, **7**, (5), pp. 191–199

© The Institution of Engineering and Technology 2020