Cluster Computing (2023) 26:495–512
https://doi.org/10.1007/s10586-021-03526-7

High-performance pseudo-anonymization of virtual power plant data on a CPU cluster

Mahdi Abbasi[1,2] · Azam Fazel Najafabadi[1] · Seifeddine Ben Elghali[2] · Mohamed Zerrougui[2] · Mohammad R. Khosravi[1] · Habib Nasser[3]

Received: 12 April 2021 / Revised: 21 December 2021 / Accepted: 22 December 2021 / Published online: 15 January 2022

Abstract

The considerable move towards the use of renewable energy resources has been provided by the digitization of energy systems with the help of virtual power plants (VPPs). However, due to the coincidence of this move with the introduction of new technologies in information and communications, joining these systems raises concerns about the privacy of personal data. The only real-world approach widely used in this case is to anonymize or pseudonymize the information associated with individuals in data received from distributed measurement devices. In this paper, we propose the method of classifying received data packets into different flows and assigning different access levels for each flow. This method makes data pseudonymous. Before this step, the received data, which has a different format, is unionized. To implement this idea, a tuple space flow classification algorithm is parallelized on a CPU cluster using MPI and OpenMP according to different scenarios. The CPU cluster consists of one head node and two computational nodes for packet classification operations. In this research, two scenarios have been used to run the CPU algorithm in parallel. The first scenario uses MPI and the second scenario uses a combination of MPI and OpenMP libraries. Also, the Tuple Space algorithm has been implemented on the computing systems using the mentioned libraries in the form of two scenarios using OpenMP and MPI. According to our results, the increase in the number of processor cores is linearly correlated with the increase in the speed of classification. Furthermore, while MPI uses more memory than OpenMP, it helps to achieve a higher speed of classification. In the combined method, the maximum speed of flow classification can be achieved if the number of processes and threads is equal to the number of processor cores. In other words, when the sum of processes and threads does not outnumber CPU cores, the least classification time and memory usage can be achieved.

Keywords Virtual power plant (VPP) · Anonymization · Flow classification · Tuple space algorithm · CPU cluster · MPI · OpenMP

1 Introduction

The energy crisis is on the rise. The rapid growth of electricity demand, especially in industrial domains with heavy energy consumption, has necessitated the use of distributed renewable energy [1, 2]. Simultaneously, with the introduction of controllable systems for energy storage, new energies that are more than needed are imported into electrical storage systems to compensate for uncertainty in the power generation grid when necessary. Accordingly, the concept of virtual power plant (VPP) as a successful method has been proposed to meet the needs in the aggregation of renewable and distributed energies, respond to demands for electricity consumption and control electrical energy storage systems to maximize daily income from the electricity market [3].

The VPP is a set of physical devices producing or consuming energy [4–6]. In this complex, generators, storage units, and manageable or flexible loads, each of which forms a virtual entity and interacts together. Along with the centralized models for VPP, hierarchical models

✉ Mahdi Abbasi
abbasi@basu.ac.ir; mahdi-abbasi@lis-lab.fr

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

[2] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

[3] RDI'UP (Innovative Research), 2 Rue Louis Blériot, 78130 Les Mureaux, France

of VPP are also presented. In a typical hierarchical model, as illustrated in Fig. 1, power generators, storage units, and low-level flexible loads are used that can be controlled by a remote terminal unit (RTU).

At the middle level is the VPP node. This node is the main operator and decision-making and can be an independent operator or distributed system operator. Such a node is responsible for managing supervised RTU and optimizing the performance of distributed resources as well as coordinating with the market. The knowledge base needed to virtualize resources controlled by RTUs is at the disposal of the VPP node. The required data are collected heterogeneously at specified time intervals from smart electronic devices installed in RTUs and transferred to the VPP node to control and optimize the VPP performance through standard communication mechanisms [6, 7]. Since the measurement data are collected from different installations with inhomogeneous scales and units, a unification is performed on them to represent them with the same units, labels, and scales [8]. These data are used for strategic management of customer energy assets, optimal planning of controllable and flexible loads, control of renewable energy producers and storage units, as well as strategic management of service quality provided to end-users.

According to Global Data Protection Regulation (GDPR), any data containing personal information must be anonymized before going through any of the abovementioned interrelated processes in the VPP node [7, 9–12]. That is, to establish the required degree of trust in real persons who join the VPP, a certain level of security and privacy must be provided. Two common methods for this objective are anonymization and pseudo-anonymization [13]. The former is the changing of personal information so that the individual information about personal or material relationships can no longer be assigned to a certain person or determinable natural person or only with an unreasonably great expense of time, costs, and effort. The latter is the processing of personal data in such a way that the personal data or enlistment of additional information can no longer be traced to a specific person if this additional information is to be stored separately and is subject to technical and organizational measures which ensure that the personal data cannot be assigned to an identified or identifiable natural person. In pseudonymization, the data that would allow for identification are replaced with a code, for example. However, there is a separate key (e.g., in the form of a table) between the subject and the pseudonym, so that it is ultimately still possible to re-identify the subject if one knows this key. In anonymization, however, all identifying characteristics are deleted. Due to the re-identification possibility provided by pseudo-anonymization, it is the more commonly accepted approach [9, 10].

Traffic flow classification is essential to a broad range of network processes and management, e.g., privacy- preserving, quality-of-service (QoS) provisioning, and intrusion detection [14, 15]. In this paper, we propose a novel pseudo-anonymization method that can be parallelly exploited for VPP data. In this method, the unionized data are first classified into specific flows regarding their source and destination addresses. At this step, a unique flow number is associated with each installation. The corresponding part of data that includes personal information is

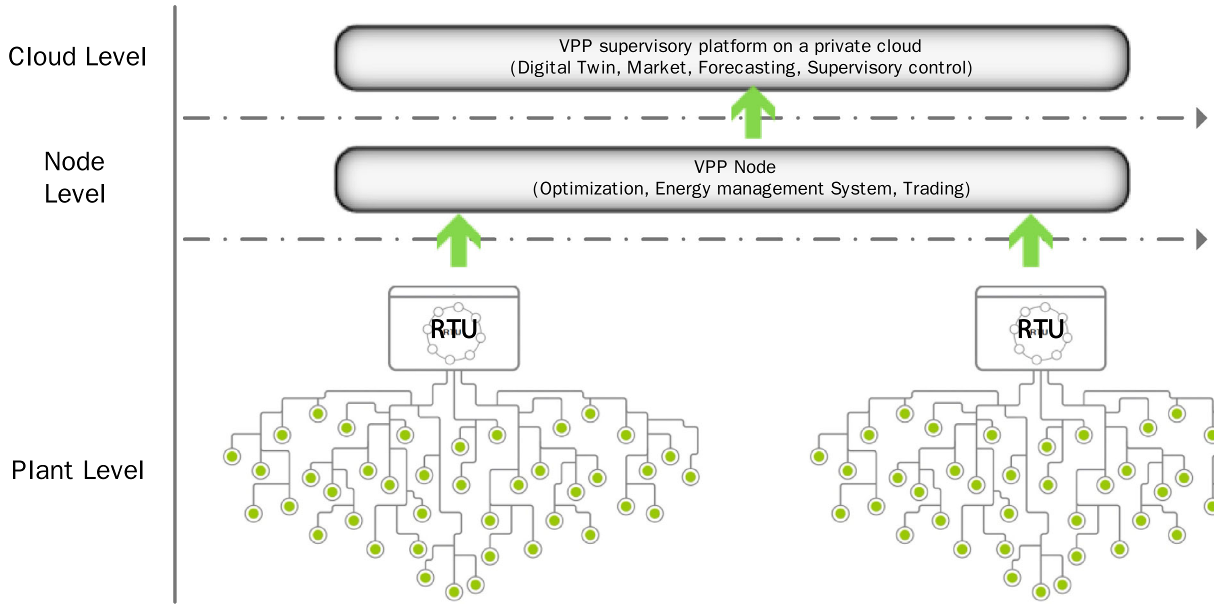

Cloud Level
Node Level
Plant Level
VPP supervisory platform on a private cloud
(Digital Twin, Market, Forecasting, Supervisory control)
VPP Node
(Optimization, Energy management System, Trading)
RTU
RTU

Fig. 1 The architecture of Information and Communication Technology (ICT) infrastructure of hierarchical VPP

encrypted and stored in a secure database with the flow number as the key.

Our parallel flow classification method classifies the Internet Packets (IPs) into certain flows according to the address characteristics of their sender and receiver processes. While normalizing them for each stream, providing a certain level of access and confidentiality controls in the knowledge base of the VPP node. Figure 2, illustrates this idea. Our method is based on the classification of internet flows. Also, we explain a method to accelerate our flow classification on the cluster of CPUs and explain different scenarios in implementing that method.

Methods of classification are either hardware-based or software-based. The main disadvantage of hardware-based methods is their restriction in using memories that can be used in parallel searching. Moreover, they have high cost and power consumption [6–8]. As a result, they are only appropriate for classifications that have a small number of filters. On the other hand, software-based methods have recently come to the fore [7, 9, 10]. These methods are highly flexible and more cost-effective. Software-based methods have been widely researched. Many studies have sought to reduce the time and frequency of memory access by making use of certain techniques for developing algorithms and data structures [16, 17].

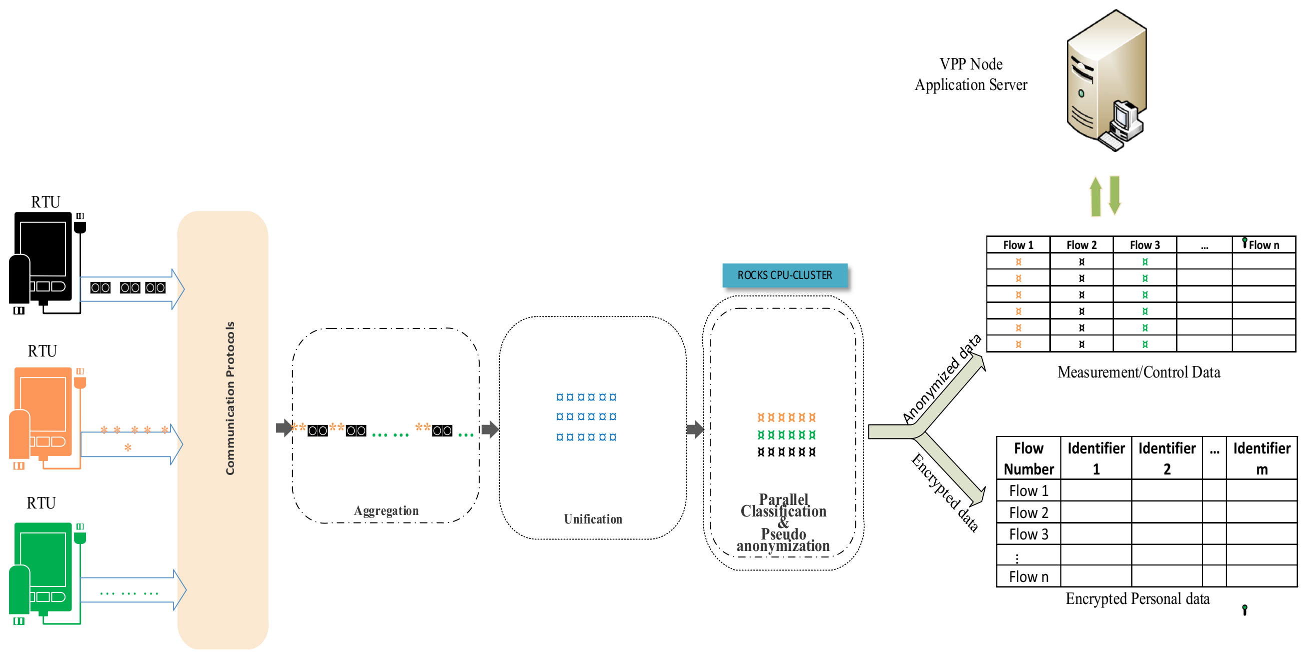
Taylor has categorized classification methods. He mentions four general categories including linear search, decomposition, tuple space, and decision tree [18]. Each algorithm is briefly described below.

Linear search In this algorithm, filters are arranged according to their priority. Each incoming packet is compared with all the filters. The algorithm performs quite efficiently in terms of memory usage; however, the search time increases linearly as the number of filters increases.

Decomposition The problem of multidimensional packet classification can be decomposed into several one-dimensional linear search problems on a single field to apply proven techniques that could search according to a single field. In these methods, the speed of searching the routing table is relatively higher due to their hardware-based implementation along with the parallel execution of the algorithm. Their disadvantage, however, is that their memory usage is remarkably high.

Tuple space This method quickly narrows down the scope of the search by breaking the filter set into tuples. A tuple shows the number of specified bits in the fields of the filter. This group of algorithms is based on the fact that unique tuples are significantly smaller in number than the total number of filters in a filter set.

Decision tree It is a binary tree that is made of filter prefixes. Its nodes hold filters or a subset of filters. Necessary operations are conducted on encountering a single node depending on the algorithm that uses a decision tree. The advantage of this method lies in its parallel functioning. In decision tree-based algorithms, a decision tree is produced based on several fields. In such a tree, the leaves hold filters or subsets of filters. To search by using decision trees, a search key is built from the fields of the packet



Flow 1	Flow 2	Flow 3	...	Flow n
¤	¤	¤		
¤	¤	¤		
¤	¤	¤		
¤	¤	¤		
¤	¤	¤		
¤	¤	¤		

Flow Number	Identifier 1	Identifier 2	...	Identifier m
Flow 1				
Flow 2				
Flow 3				
⋮				
Flow n				

Fig. 2 The parallel classification and anonymization of data received from different RTUs on a CPU-cluster

header. Afterward, the bits or subsets of the bits of the search key are used to traverse the tree.

From among the four methods described above, the tuple space algorithm will be investigated in the present paper. The classification time can be significantly reduced by parallelization of the algorithm, thus providing a compromise between classification time and memory usage.

There are multiple methods for the parallel implementation of algorithms [19–24]. One common method is to use a CPU cluster. In the present study, the hierarchical trie algorithm is parallelized for the first time on a CPU cluster.

The paper is organized as follows. Section 2 discusses two tuple space algorithms as well as the classification method in this kind of algorithm. Next, cluster systems and their programming will be described. Then we will have a brief review of the literature on computation clusters and parallelization of packet classification. In Sect. 4, the implementation of the proposed scenarios for parallelization of the tuple space algorithm on a CPU cluster will be described. The results of the implementations will also be analyzed and evaluated in this section. The final section offers some suggestions for further research and practice in the field.

2 Related work

2.1 Algorithms and tools

This section describes the structure as well as the classification method of the tuple space algorithm. Afterwards, we shall explain cluster computation and the parallelization tools used for this purpose.

2.1.1 Tuple space algorithm

The tuple space algorithm maps each k-dimensional rule to a tuple with k elements. In such a way that the set of rules mapped to regular tuples with k elements have fixed and definite lengths.

The tuple space algorithm was first proposed by Srivinasan et al. [25]. This method quickly narrows down the search scope of the fields. The main reason for developing a tuple space algorithm is the smaller number of separate tuples compared with the total number of filters. This will be done by breaking the filter sets into tuples. A tuple shows the number of bits specified in each field of the filter. For example, considering 2-dimensional filters, both F1(01*,111*) and F2(10*,010*) map to the tuple [2, 3]. An example of a tuple is presented in Table 1.

The execution of this algorithm on the input (11100,11101,53,25,4) with the filter set of Table 2 is described in the following. The input (11100,11101,53,25,4) contains five fields of a packet header including source IP, Destination IP, source port, destination port, and protocol fields. These fields are shown in order from left to right in Table 2

Table 1 Method of creating a tuple

Rule	(Source address, destination address)	Tuple
R1	(00*,00*)	(2,2)
R2	(0*, 01*)	(1,2)
R3	(1*, 0*)	(1,1)

The binary trees produced based on the source and destination address fields are shown in Fig. 3. The black nodes represent the filters. The corresponding tuples are shown next to these nodes.

In this example, the search is first performed on the corresponding binary tree based on the source and destination address fields. The search paths are shown in the dashed line in Fig. 3, respectively from left o right. The filters corresponding to the tuples encountered on the traversed path are extracted.

In the end, the best matching filter, if any, is specified by a linear search on the filters extracted in the previous step. In the above example, the best matching filter is R4.

The filters corresponding to the tuples encountered on the traversed path are shown in Table 3. As can be seen, the number of filters obtained is smaller than the total number of initial filters. This technique can help to remove a significant number of filters in large filter sets.

2.1.2 Cluster computing

As clusters are intended to increase the processing power or the physical security of information and services, they are more reliable and cost-effective than a single server. In a cluster, the computers are not required to have the same performance; they only have to have identical architectures. The difference in architecture will cause problems in the process of clustering [26, 27].

Some applications of clusters include High-Performance Computing (HPC), High Throughput Computing (HTC), High Availability (HA), and High-Performance Systems (HPS). One of the most important of these is HPC clustering services. In HPC, both the software and hardware power of a computer is used in parallel to perform a large amount of processing in a shorter time [28].

Cluster processing is counted as a kind of parallel or distributed processing. The cluster refers to a set of independent computers that are closely interrelated with the whole system and act together as an integrated resource.

Table 2 Example of a classification rule set

Filter	Source IP	Destination IP	Source Port	Destination Port	Protocol
R0	1010*	01*	0,65,536	25,25	6
R1	10,001*	0111*	53,53	443,443	4
R2	0*	1110*	53,53	1024,65,535	17
R3	0*	1100*	53,53	443,443	4
R4	111*	1110*	53,53	25,25	4
R5	1110*	*	0,65,535	2788,2788	17
R6	1*	10*	153,53	5632,5632	6
R7	*	1*	53,53	25,25	6
R8	*	10*	0,65,536	2788,2788	17

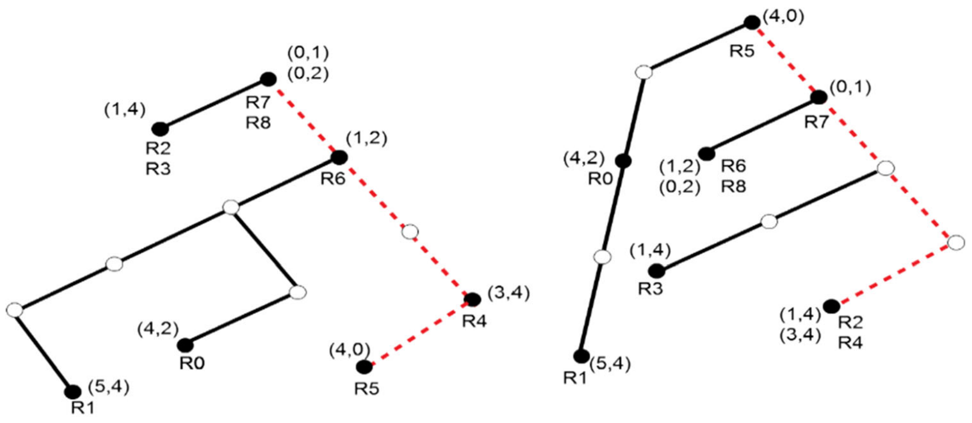


Fig. 3 Binary trees of source and destination addresses

Table 3 The selectional rules of tuple space search based on the rules in Table 2

Filter	Source IP	Destination IP	Source port	Destination port	Protocol
R2	0*	1110*	53,53	1024,65,535	17
R4	111*	1110*	53,53	25,25	4
R5	1110*	*	0,65,535	2788,2788	17
R6	1*	10*	53,53	5632,5632	6
R7	*	1*	53,53	25,25	6
R8	*	10*	0,65,535	2788,2788	17

They can be indeed regarded as a single system. In most cases, the components of a cluster connect through a local network. There should be at least two components at work. In such a structure, the systems need a mediator for transferring messages and a scheduler for allocating resources [26, 28–30].

One of the major services provided by cluster systems is high-performance computing (HPC). As mentioned above, this method of clustering is also known as parallel processing [31]. What follows is a brief description of the parallel programming models used in this study.

2.1.2.1 OpenMP OpenMP is an API that is widely used for parallel programming with C, C++, or Fortran in shared memory systems. A variety of architectures are supported in this interface including Windows and Unix platforms. This tool provides threads in a cluster system with shared memory. The strength of OpenMP lies in the fact that the parallel and serial versions of a piece of code are remarkably similar. Thus, a serial program in OpenMP can be converted into a parallel program simply by adding several instructions. Other merits of OpenMP include its simple and highly standard conventions, cross-platform structure, and popularity among programmers [31–34].

2.1.2.2 MPI Message passing interface (MPI) is the most widely used method of parallel programming. MPI determines the features of a general API to be used on shared-memory systems such as clusters. MPI is not a tool as such; it is a communication protocol that, as its name suggests, specifies how parallel systems can exchange messages [32–34]. Its major advantage over other methods of message transfer is its portability and high speed. Its high speed is explained by the fact that it can be optimized during execution on any hardware configuration. Moreover, its functions can be called in C, C++, Fortran, Java, C#, and Python [35–39].

MPI has various implementations for different operating systems and hardware configurations. One of the implementations is MPICH that is an open-source Linux implementation. The power of MPICH programs lies in their ability to be executed on the majority of important cluster architectures in the world. MPICH includes the C, C++, and Fortran libraries required to use MPI-2. For these reasons, we used MPICH as the message transfer interface in our experiments.

2.1.2.3 Hybrid method The hybrid method makes simultaneous use of OpenMP (for cores with shared memory) and MPI (for those with separate memories). However, MPI can also be used for the cores within a single system. Figure 4 illustrates the structure of hybrid architecture. In this study, we use all of the above three methods for parallelization of the tuple space algorithm on a CPU cluster.

2.1.2.4 Parallel packet classification This section first addresses the algorithms so far parallelized on GPUs and multi-processor systems and then reviews previous studies of parallelization on GPU clusters.

Among the first studies of parallelization of packet classification algorithms on GPUs is the research conducted by Nottingham et al. in 2009. They theoretically investigated the possibility of parallel implementation of classification algorithms on GPUs [35].

Hung et al. conducted the first applied study in the field in 2011 and investigated the parallelization of two classification algorithms, i.e., BitMap-RFC and BPF, on GPUs in the framework of CUDA. They used three filters for implementation, and evaluation of the classification of 65 random packets [22]. Simultaneously, Deng et al. parallelized linear search algorithm with a combined method that used both GPU and CPU. They confirmed that this combination could increase processing power and decrease the delay in comparison with GPU-based methods. Another study was done in the same year by Pong et al. parallelizing HaRP and HyperCuts algorithms on multi-core processors. In their study, the maximum throughput rates obtained for HaRP and HyperCuts were 30.14 and 4.07 packets per second, respectively [36]. In 2011, Kang et al. implemented the DBS algorithm (an algorithm based on hash tables) along with linear search on both GPU and CPU. The speedup rate of DBS was 11 [37]. In 2014, Varvello et al. parallelized three algorithms on GPU, i.e., linear search, tuple space search, and Bloom search. The maximum speedup rate was 7 [38]. In 2016, Rafiee et al. implemented the hierarchical trie algorithm on a multi-core processor and GPU. For their implementation they used a GeForce GTX 750 graphics card which has four SMs. They used an Intel Core i7-3770L CPU and utilized CUDA and OpenMP for parallelization on GPU and CPU, respectively. Among their five scenarios for GPU and one scenario for CPU, the scenario which used shared memory on GPU had the best performance [39].

A major study of parallelization of packet classification algorithms on multi-core systems is the Zhou et al. [40]. They utilized Pthread for parallel implementation of linear search algorithm and area-based tree search algorithm on a 16-core processor. The maximum throughput achieved was 11.5 Gbps. Qu et al. conducted an influential study in 2015 and implemented a bit-vector decomposition algorithm on multi-core processors. Their parallelization was based on the OpenMP library. Their maximum throughput rate was 14/7 mega packets per second [41]. Tung et al. recently implemented a new parallel packet classification algorithm on an eight-core processor. According to their results, the productivity of the algorithm increased 40 percent on average. Also, the delay in addition and omission of filters had an average reduction of 43 percent. Furthermore, they improved the productivity of cache memory by setting the parameter of the dependence of CPU on threads and [41].

Recent developments subject to change in the Internet of Things (IoT), have motivated many researchers to look solutions for subsequent challenges in energy management,

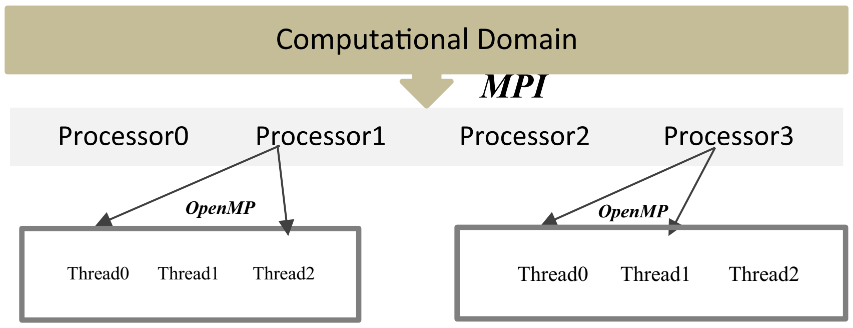


Fig. 4 Combination of MPI and OpenMP programming models

load balancing, security provisioning, and edge computing [42]. Software Defined Networking (SDN) has provided a suitable framework for secure processing of the network flows with reasonable speeds [43, 44]. Recently, clusters of network processors have been exploited to enhance the speed of SDN networks. Jafarian et al., have fully analyzed SDN anomaly detection mechanisms on clusters of computing nodes [45]. Also, Mei-ling et al., have proposed a novel load balancing algorithm that considers the real-time processing of SDN loads on the cluster of SDN servers [46]. It still has limitations concerning the number of its cores. This is why there is a widespread tendency to cluster systems. CPU clusters have great potential for development. This means that a new computational node can be simply added to it to use more cores simultaneously. Thus, in this study, we use a CPU cluster and apply several different scenarios to classify network packets. In the following, we shall review some parallelized implementations on CPU and GPU to explain the achievements of this type of parallelization.

Henty et al. used MPI and OpenMP on a CPU cluster for modeling purposes [19]. They concluded that the performance of this combined method was lower than the MPI-alone method due to its overload. They also indicated that the OpenMP-alone method performed better than the combined method in small-scale problems. As their results suggest, this conclusion could not be generalized to all parallelization problems, the outcome depending on the structure of the code as well as on other circumstances. J. Hutter et al. combined MPI with the shared-memory method by using a 1024-core cluster. Their results showed that a major cause of inefficient parallelization in the combined method was connection delay [47]. Cappello et al. applied two methods to numerical simulation problems of aerodynamics, namely, a combined method and an MPI-alone method. According to their findings, the computation efficiency of a cluster depends several parameters such as memory access pattern and hardware performance [48]. In 2018, M. Ferretti et al. implemented Cross Motif search in a parallel form on a CPU cluster. Their study indicated that MPI alone performed better than the combination of MPI and OpenMP [49]. Q. Zhao et al. indicated in 2019 that large-scale numerical simulation for analyzing discrete spherical forms requires large amounts of time. A major factor affecting the efficiency of this simulation is the solving of linear equations in this problem. They used OpenMP, MPI, and combined method for parallelized implementation of this problem on the cluster Sugon TC4600. Their results suggested that the combined method performed better than the other two methods. According to their speedup graph, an increase in the number of processes and threads will increase efficiency as long as this number is less than processor cores [50].

As this review shows, clusters could provide higher parallelization capability for algorithms in different combinations of programming models. It also discloses that the efficiency of a programming model is dependent both on the type of the problem and on the implemented system. In this line, the present study intends to parallelize the tuple space algorithm for the first time on a CPU cluster. Below, we will show that by using different scenarios we can significantly increase packet processing speed and throughput rate.

3 Implementation of tuple space algorithm

Here we first take a look at the specifications of the implemented cluster and then describe the parallelization of the tuple space algorithm on this cluster by using a combined model.

3.1 Specifications of the implemented cluster

Our cluster was implemented on the distributed operating system Rocks which is based on Centos. The typical topology of the Rocks cluster is illustrated by Fig. 5.

Being an open-source operating system, Rocks has been designed to simplify processing, development, and management as well as to enhance performance in parallel cluster systems. Installation of Rocks on a master node requires two network adapters. One adapter should be used for internal communications (eth0) and the other for external communications (eth1). The operating system should be first installed on the master node and then on other computers. Most packages needed for clusters such as MPICH and OpenMPI are installed as default in Rocks [51].

In a cluster, communication between systems is performed through switches. We used a D-Link 100/1000 switch. Also, we used two homogeneous systems with quad-core CPUs for performing computations. Figure 6 depicts the architecture of the systems as well as how they are connected.

As shown in the figure, each system had a separate CPU and memory (i.e., symmetric multiprocessing architecture). The CPU specifications are listed in Table 4. The factors affecting CPU speed include the number of cores, cache memory, and bus speed, the most important one being the number of cores. Increasing the cores allows the CPU to perform more instructions simultaneously.

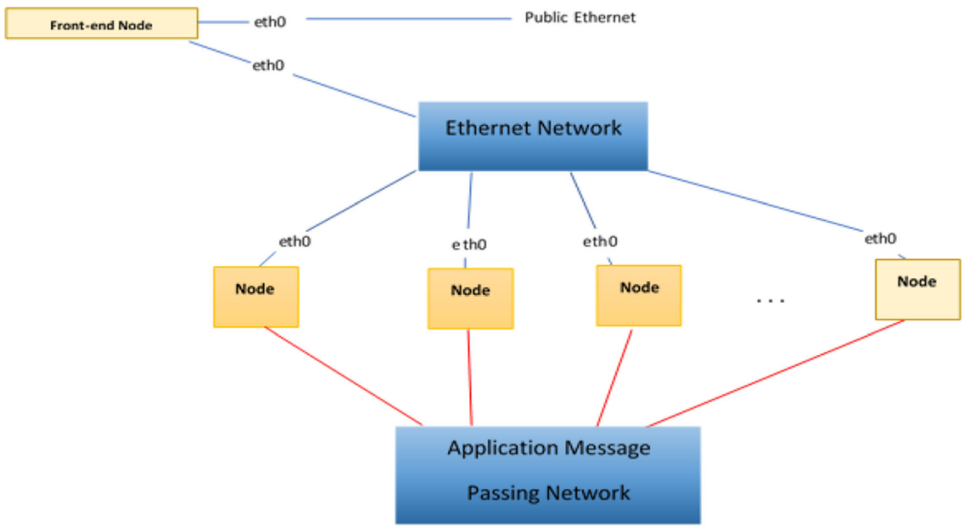


Fig. 5 Clustering in Rocks

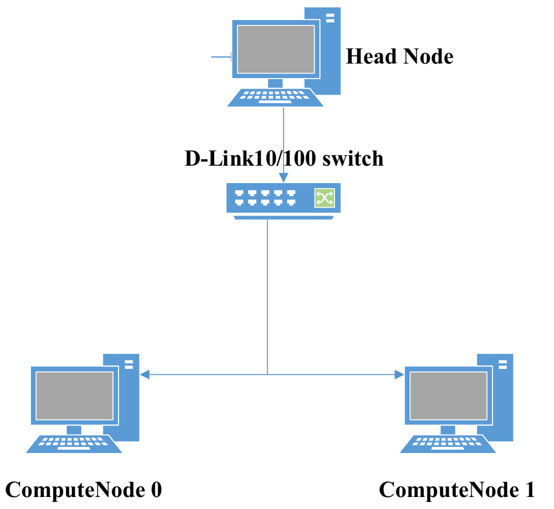


Fig. 6 The topology and configuration of the installed CPU Cluster

Table 4 Specifications of the CPU

Processor family	Intel Core 2QuadQ6600
Status	56End of Interactive Support
Launch Date	Q1′07
Lithography	65 nm
# of Cores	4
Processor Base Frequency	2.40 GHz
Cache	8 MB L2
Bus Speed	1066 MHz FSB
FSB Parity	NO
Cache size	4096 KB
TDP	105 W
VID Voltage Range	0.8500 V-1.500 V

3.2 Implementation of hierarchical trie algorithm on a CPU cluster

3.2.1 Using one system

The first scenario As described earlier, OpenMP uses threads for parallelization. All of the systems used have quad-core CPUs. Therefore, the number of required threads in the program varies from 1 to 4.

The algorithm of this scenario is presented below. The inputs of the algorithm are the filter set R, the tree structure T, the packets H, and the number of threads N. First, by using the parallelization commands of OpenMP, the for-loop is split statically among the four input cores which simultaneously process incoming packets in a parallel manner (Lines 1 and 2). In the end, the index corresponding to the best matching filter is stored in ruleIndexArray and returned as output. The length of this array is equal to the number of packets.

```
Input: rules R, T
− space T, Headers H, Numthread N
Output: ruleIndexArray
1:  #pragma omp parallel for num_threads(N)
2:  for all i ∈ [0, H] do
3:      P ← ReadPacket (i)
4:      rIdx ← Classify (P, T, R)
5:      if rIdx ≠ Null then
6:          ruleIndexArray (i, rIdx)
7:      end if
8:  [illegible]
9:  end for
```

Algorithm 1 Splitting the packets in the system using OpenMP

The second scenario As in the previous scenario, the criterion is the number of CPU cores; however, processes are used instead of threads. The MPI generates some processes each of which is assigned a file to execute. MPI runs the executable file simultaneously in all the cores. The rank of each process which is a unique id can be used to separate the packets corresponding to that process. This feature is used in this study to divide the packets among different processes. Figure 2 presents the pseudocode for the algorithm which distributes packets among processes. The inputs to the algorithm are the total number of processes (S), the packets (H), and the rank of each process. By use of the unique rank of each process, the function returns the index of the first ($start_i$) and the final ((end_i)) packet which are to be classified by the i-th process. In this pseudocode, H_i specifies the number of packets assigned to the i-th process.

```
Input: Mpi size $S$, Headers $H$, rank $R$
Output: Headers $H_i$, $start_i$, $end_i$
1:   if $(|H| \% S) == 0$ then
2:       $|H_i| \leftarrow |H| / S$
3:   else
4:       $|H_i| \leftarrow (|H| / S) + 1$
5:       $start_i \leftarrow R * |H_i|$
6:       $end_i \leftarrow ((R + 1) * |H_i|) - 1$
7:   end if
8:   if $(R == (S - 1))$ && $(end_i > H)$ then
9:       $end_i \leftarrow |H|$
10:  end if
```

Algorithm 2 Distribution of packets in a system using MPI

After distributing the packets among the processes and finding the indexes (end_i, $start_i$) as well as the number of the packets for each process, these values along with the filter set R and the tree structure T are given as arguments to the classification algorithm. As can be seen in Fig. 7, for all the processes, the classification algorithm stores the best matching filter for the i-th process in $ruleIndexArray_i$, which corresponds to $ruleIndexArray$ in the range [$start_i$,end_i], and returns it as output.

```
Input: rules $R$, $T-$space $T$, Headers $H_i$, $start_i$, $end_i$
Output: $ruleIndexArray_i$
1:  for all $i \in [start_i, end_i]$ do
2:      $P \leftarrow ReadPacket(i)$
3:      $rIdx \leftarrow Classify(P, T, R)$
4:      if $rIdx \neq Null$ then
5:          $ruleIndexArray(i, rIdx)$
6:      end if
7:      $i \leftarrow i + 1$
8:  end for
```

Algorithm 3 Packet classification in the CPU cluster

3.2.2 Using two systems

The first scenario This scenario differs from the second one-system scenario only in that it uses MPI processes without making use of any OpenMP thread. In this scenario, the algorithm is executed by a number of different processes. As mentioned earlier, this CPU cluster consists of two systems, each with four cores. Therefore, the maximum number of processes that can be defined is 8.

The second scenario In this scenario, the algorithm is executed by several processes and threads on the systems of the CPU cluster. In fact, this scenario uses a combined MPI-OpenMP method. As this cluster consists of two 4-core systems, for each packet volume the algorithm can

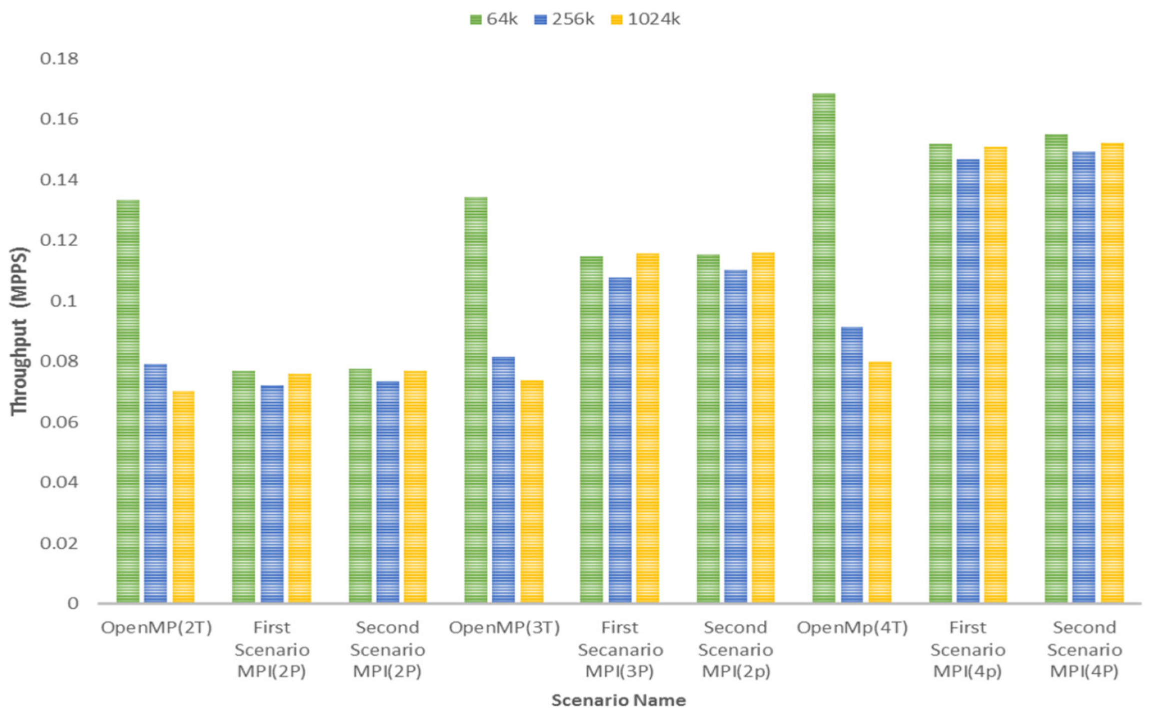


Fig. 7 The throughput of OpenMP and MPI scenarios in a system

be executed with 1–8 processes and 1–4 threads, giving a total number of 32 threads. Due to a large number of results, we have sufficed to only two states, i.e., 8 processes with 2 and 4 threads.

4 Implementation and evaluation

In this section, first, a software suite is described for generating the filter sets of experimental headers. We will then explain our criteria and evaluate the results of the scenarios presented in Sect. 4.

4.1 ClassBench suite

ClassBench which is a simulator based on the C language and Linux platform can generate the filters of classifiers as well as their corresponding headers. This suite produces filter sets that are similar to actual filter sets. It utilizes two modules. The first module generates an arbitrary number of filter sets while the second one generates a set of random packets based on the statistical features of the filters produced by the first module [52]. ClassBench can generate three types of filter sets: Firewall (FW), Access Control (ACL), and Chain (IPC). This suite has been used in many studies [53, 54] to generate filters in their evaluation of packet classification methods.

We also used this suite to produce filter sets corresponding to IPC2 with 1 k filters as well as 32 k, 64 k, 128 k, 256 k, 512 k, and 1024 k incoming packets in order to evaluate our scenarios. The C++ programs written for each scenario were executed with different numbers of packets and the average results were recorded as the final results of classification.

4.2 Evluation metrics

Various criteria exist for evaluating the efficiency of network packet classification methods. These criteria are briefly described below.

Throughput: One of the criteria is throughput. It is defined as the number of packets classified in the unit of time. It is measured in packets per second (PPS).

$$Throughput = \frac{number\ of\ headers \times 1024}{T_{Classification}/1000} \tag{1}$$

Classification time Classification time is the elapsed time during which the classification of packets is performed by the classifier system. It is denoted by $T_{\text{cllasification}}$ and calculated in two ways for different processes. The first method is to take the maximum time from among the times of all processes because it takes this amount of time to perform all classifications completely. Equation (1) shows how the time is calculated in this method.

$$T\_max_{cllasification} = max\{t_{cllasification_1}, t_{cllasification_2}, \ldots t_{cllasification_{np}} \tag{2}$$

As with the second method, the average time is considered instead of the maximum time. This is calculated by dividing the sum of the times by the total number of processes. Equation (2) represents this method.

$$\text{T\_avg}_{cllasification} = \frac{\sum_{i=0}^{np-1} t_{\text{cllasification\_i}}}{np} \tag{3}$$

Speedup Speedup is the result of the division of classification time in the serial mode by classification time in the parallel mode.

$$\text{S (s, p)} = \frac{\text{T(s,1)}}{\text{T(s,p)}} \tag{4}$$

Here, T (s, p) is the time for parallel execution of the algorithm, and T(s,1) is the time for serial execution of the algorithm.

Transfer time Transfer time refers to the time needed for copying the required data structures from the CPU memory to the memory of classifier systems.

Processing delay of packets This criterion refers to how long it takes on average to classify a packet.

Memory usage In this study, an IPC filter set was used. The number of these filters is 634, for each of which there exists a source and destination tree with 1648 and 861 nodes, respectively. In Linux, each of these nodes requires 40 bytes of memory, making up a total memory of 40*(1648 + 861) bytes.

Moreover, a certain amount of memory should also be allocated to packets, filters, and an output array used for displaying the results. Given this, the total memory needed is represented by $Memory_i$. It should be noted that processes have a separate address space. Therefore, to obtain the memory usage of np processes, this amount of memory should be multiplied by np.

$$\text{Total\_Memory} = \text{np} * Memory_i \tag{5}$$

4.3 Sequential implementation

Time Table 5 shows the classification time for different packet volumes in the two scenarios. The table has three columns that represent classification time in the sequential mode as well as using MPI and OpenMP with different numbers of cores. The results show that, in the sequential mode, by doubling the number of packets the time will approximately double. The reason is that the classification time increases as the number of packets increase.

Table 5 Packet classification time in the sequential mode, in OpenMP, and in MPI on one of the systems

Number of packets (K)	Sequential		Cores	OpenMP		MPI	
	Classification time (ms)	Process delay (µs)		Classification time (ms)	Process delay (µs)	Classification time (ms)	Process delay (µs)
64	884.56	13.49	1	979.892	14.95	1694.67	25.85
			2	491.732	7.50	850.417	12.97
			3	487.796	7.44	570.429	8.70
			4	388.963	5.93	430.919	6.57
128	1751.36	13.36	1	1944.89	14.83	3326.84	2.53
			2	1039.64	7.93	1685.23	1.28
			3	1067.81	8.14	1133.44	0.86
			4	933.921	7.12	849.5	0.64
256	3741.95	14.27	1	4747.93	18.11	7054.79	26.91
			2	3306.48	12.61	3630.25	13.84
			3	3220.28	12.28	2428.54	9.26
			4	2864.19	10.92	1785.08	6.80
512	7990.66	15.24	1	10,182.3	19.42	13,391.9	25.54
			2	7100.45	13.54	6825.7	13.01
			3	6776.84	12.92	4543.04	8.66
			4	6223.98	11.87	343.72	6.54
1024	17,507.4	16.69	1	21,023.90	20.04	26,949	25.70
			2	14,921.1	14.22	13,760.1	13.12
			3	14,188.6	13.53	9066.17	8.64
			4	13,115.3	12.5	6942.46	6.62

As the results indicate, in the OpenMP scenario, the algorithm execution time decreases as the number of threads increases. The reason is that the incoming packets are processed by a larger number of threads and the processing load of every single thread is reduced; therefore, the time required for classification decreases. This finding also holds for the MPI scenario because the incoming packets are distributed among different processes and each process performs independently and in parallel with other processes. By comparing the execution time of the algorithm in OpenMP and MPI it can be understood that MPI has a shorter classification time. The packets are processed with higher speed because MPI does the work through several processes whereas OpenMP deals with one process and several threads.

Throughput We defined throughput as the number of classified packets in the unit of time. Figure 7 shows throughput for volumes of 64 K, 256 K, and 1024 K as well as for 2, 3, and 4 processes and threads in the first scenario (using MPI) and the second scenario (using both maximum and mean times).

The results show that in the classification of packets with large volumes, the second mode of the MPI scenario can classify more packets in the unit of time in comparison with OpenMP. Also, in both methods, the maximum number of packets are observed with 4 processes and threads because all processor cores are involved. The numbers below the graph denote the number of threads and processes.

The throughput for 1024 K packets with 4 processes and 4 cores in the OpenMP scenario, the first mode of MPI scenario, and the second mode of the MPI scenario is 0.079, 0.151, and 0.152. The second mode of the MPI scenario classifies 0.07 million packets more than OpenMP.

Memory usage Calculation of the required memory was explained under 5–2. The memory used by the two proposed scenarios is shown in Fig. 8.

Like the previous graph, this figure covers packet volumes of 64 K, 256 K, and 1024 K as well as 2, 3, and 4 processes and threads. It should be noted in this figure that memory usage in MPI increases with the number of processes because each process has an independent address space, but it does not increase with the number of threads in OpenMP.

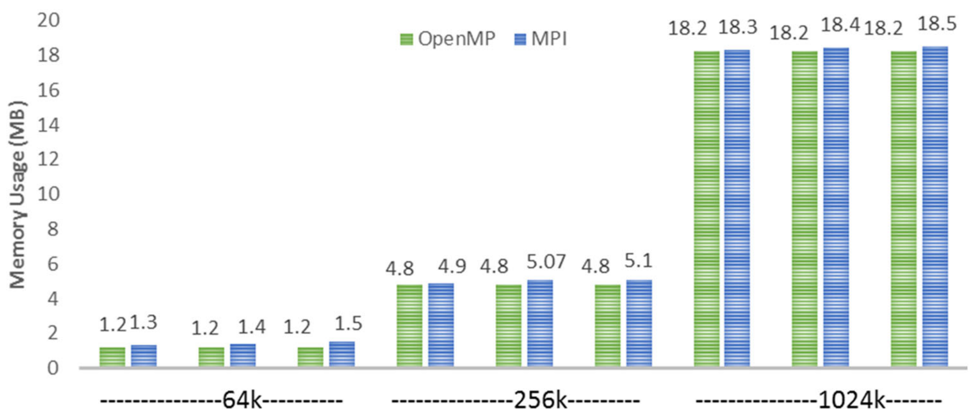
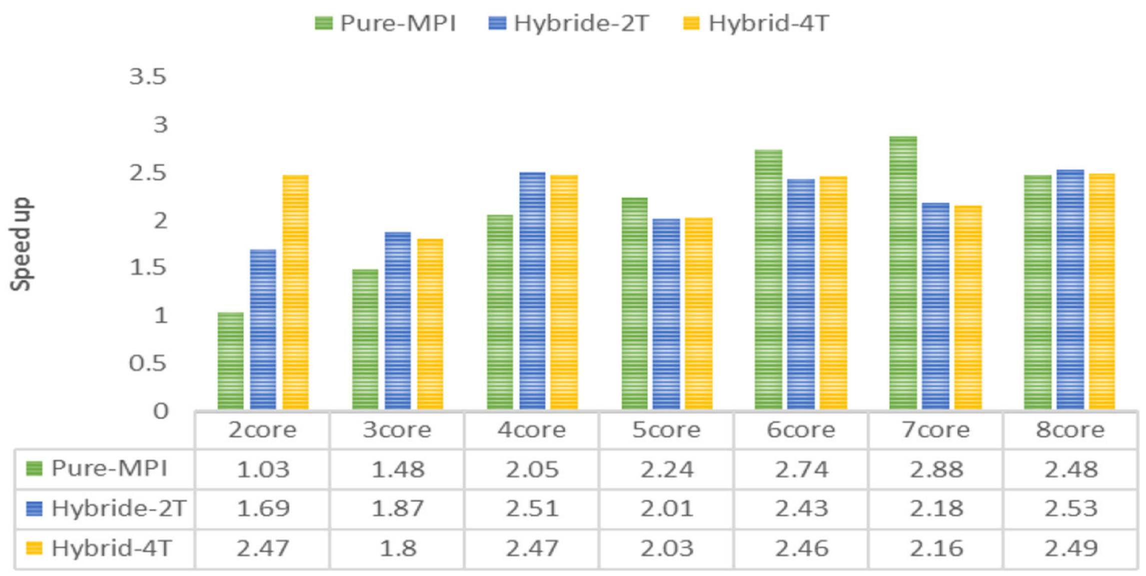


Fig. 8 The memory usage of OpenMP and MPI scenarios in a system



	2core	3core	4core	5core	6core	7core	8core
🟩 Pure-MPI	1.03	1.48	2.05	2.24	2.74	2.88	2.48
🟦 Hybride-2T	1.69	1.87	2.51	2.01	2.43	2.18	2.53
🟨 Hybrid-4T	2.47	1.8	2.47	2.03	2.46	2.16	2.49

Fig. 9 The speedup of 256 K packets using MPI and the hybrid method (OpenMP-MPI) in the CPU cluster

4.4 Parallelization on the CPU cluster

Speedup Fig. 9 shows the speedup of different scenarios run on the CPU cluster concerning to the sequential execution of the algorithm for the size of 256 K. In this figure, the combined MPI-OpenMP is illustrated with 8 processes and 2 and 4 threads.

The figure shows that the speedup parameter has a rising trend if MPI alone is used. The reason is that with an increase in the number of cores, more processes are used for classification. Therefore, the packets are distributed among more cores, which results in increased speed of execution and speedup. In the hybrid method, speedup has an increasing trend, but this increase is not observed with more than 4 cores. The reason is that the number of cores in this cluster is 8 and, when using 4 and 2 threads, 8/2 = 4 and 8/4 = 2 processes can bring about desirable results. With an increase in the number of cores, the number of processes also increases. This will lead to interference among the threads of these processes, which could increase execution time and decrease speedup.

Throughput In Fig. 10, the throughput of the different scenarios of the CPU cluster is shown for 256 K with 8 processes and 2 and 4 threads. As in the speedup figure, the hybrid method with 4 more processes has a weaker performance than the MPI-alone method and classifies fewer packets. The reason was explained above when discussing the speedup graph.

Memory usage Fig. 11 shows memory usage for the classification of 1024 K packets in the scenarios run on the CPU cluster. The results show that, as both methods use processes, memory usage in both of them is similar and increases with increasing the number of cores.

The results of the execution of the tuple space algorithm using OpenMP and MPI scenarios on a quad-core system show that requires more memory and performs better than OpenMP. Also, the results from the execution of the algorithm on the CPU cluster show that MPI, as in the one-

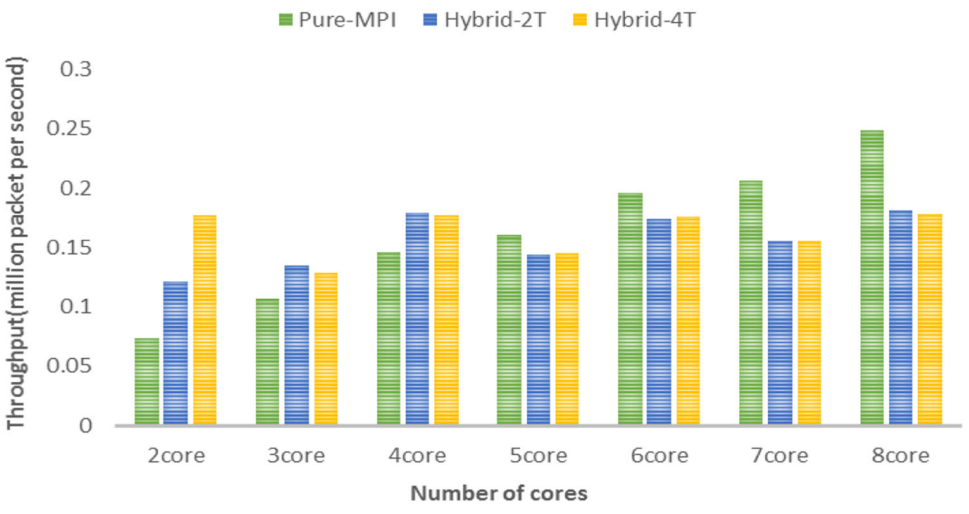


Fig. 10 The throughput of 256 K packets using MPI and the hybrid method (OpenMP-MPI) in the CPU cluster

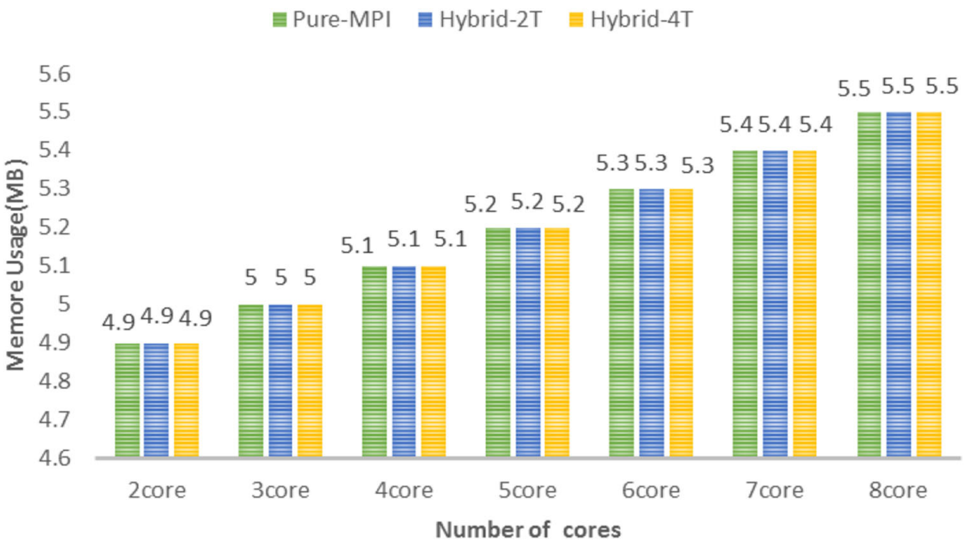


Fig. 11 Throughput of 256 K packets using MPI and the hybrid method (OpenMP-MPI) in the CPU cluster

system mode, requires more memory and has a higher performance. However, as using more processes means more memory usage, the highest throughput rate possible can be achieved by defining the number of processes equal to the number of systems in MPI and defining the number of threads equal to the number of cores in each system.

4.5 Paralleization on the GPU

The graphics card used is the Nvidia GeForce GTX 960. The graphics card has 8 SMs, and each SM has 128 processing cores, so a total of 1024 cores are available.

Pre-processing time This includes the time of construction of TupleSpace trees and transferring them and synthetic packets to the global memory of GPU. In Table 6, the preprocessing time for the different numbers of packets has been shown. The preprocessing time doubles as the number of packets doubles; for example, for 128 K of packets, this time is 0.33 s, while for 256 k of packets, it has increased to 0.69 s.

Classification time Table 7 shows the implementation results of classifying the different number of input packets with 1024 filters of IPC2 using Tuple Space. The second column is the computation time of the kernel or the packet classification time. From the table, it is evident that this

Table 6 Pre-processing time of the first scenario of a system in a GPU for different test packages

Number of packets	32 k	64 k	128 k	256 k	512 k	1024 k
Preprocessing time (s)	0.11	0.23	0.33	0.69	1.21	2.05

Table 7 Implementation results of classifying packets with the parallel version of Tuple Space algorithm on GPU

Number of packets (K)	Kernel time (ms)	Transfer time (ms)	Total time (ms)	Processing delay of each packet (μs)
32	2.203	0.566	2.769	84
64	5.321	1.23	6.551	99
128	9.683	1.522	11.205	85
256	19.332	3.442	22.774	86
512	37.474	7.011	44.485	84
1024	78.631	12.348	90.979	86

time doubles when the number of incoming packets doubles. The transfer time also doubles. Transfer time includes the time for transmission of the filters, the H-trie structure, the test packets, and the array of results from the system memory to the GPU memory and also the time of the transmission of the results from the GPU memory to the CPU.

4.6 Comparing results

In this section, the performance of all three implementations is compared. Figure 12 compares the performance of the parallelization scenarios in classifying 1024 K of packets using IPC2 filters. As it is clear when using only one CPU (quad-core CPU), the throughput is 2.3 million packs per second. The throughput of the CPUclusterb is 4.72 million packs per second. Due to the larger number of cores and, therefore, faster parallelization, the GPU can handle 13.23 million more packets in unit time. According to the results, the throughput of the CPU cluster in classifying the packets with the Tuple Space algorithm is about two times the throughput of the single system running Tuple Space. Although the throughput of the classifying packets with GPU is about four times of the throughput of the CPU cluster system, comparing 1024 cores of GPU and only eight cores of CPU cluster, the average throughput of the CPU cluster is 0.525 MPPS which as compared to 0.004 MPPS throughput of GPU, is considerably higher. This result confirms that by using appropriate parallelization APIs, the efficiency of a CPU cluster with a limited number of processing cores would be considerably higher than a GPU with many cores.

5 Conclusion

According to GDPR rules, the provision of an expected level of privacy and security in granting access privilege to the collected data from distributed subscribers of any virtual power plant is inevitable. Though this objective is possible by anonymization and pseudo-anonymization, due to its re-identification possibility, the latter is the more common approach in the ICT community.

In this paper, we presented a novel pseudo-anonymization method that is based on packet classification. In this method, the unionized data are first classified into specific flows regarding their source and destination addresses. At this step, a unique flow number is associated with each installation. The corresponding part of data that includes personal information is encrypted and stored in a secure database with the flow number as the key.

Among different packet classification algorithms, tuple space algorithms were selected and its parallel version on a CPU cluster was constructed. A review of the works conducted in this field showed that implementations of these algorithms on single-processor systems have not yet

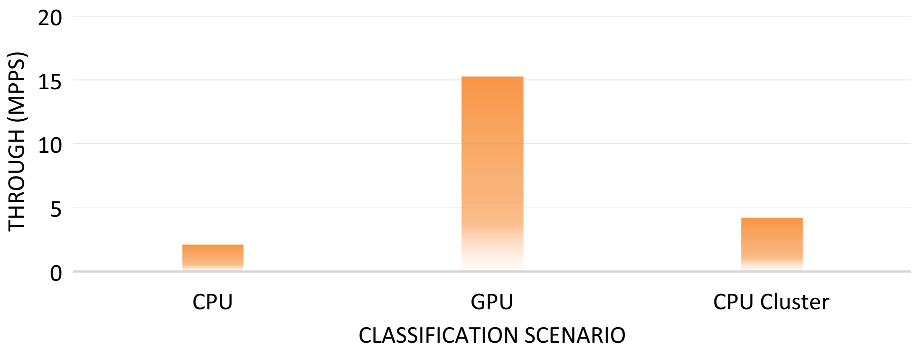


Fig. 12 Comparison of the throughput of three scenarios for classifying 1024 packets using Tuple Space

achieved desirable throughput and speedup rates and the low processing capability of these systems tends to decrease performance. To overcome this issue, the present paper implemented the tuple space algorithm on a CPU cluster. The achievements of this study are as follows:

The tuple space algorithm was first implemented and executed in two scenarios on a quad-core system in a parallel mode. Parallelization in the first scenario was performed using OpenMP and, in the second scenario, MPI was used for distribution of packets and parallelization of the algorithm. The results show that MPI uses more memory but performs better than OpenMP. Next, the algorithm was implemented in two scenarios on a CPU cluster consisting of two quad-core systems. The evaluation results suggest that the first scenario (which used MPI alone) had better outcomes than the hybrid method (MPI-OpenMP). However, it should be noted that MPI uses more memory than OpenMP.

Overall, the findings of this study suggest that the best method for implementing the tuple space algorithm on a CPU cluster is to define the number of processes as equal to the number of systems in MPI and define the number of threads equal to the number of cores in each system. Thus, the highest possible throughput rate can be achieved, which means the largest volume of packets in the unit of time.

In future work, GPU Clusters or a combination of GPU and CPU clusters can be used to increase the speed of packet classification. Given the larger number of computational cores in GPUs, it is expected that packet classification speed will be significantly increased through the optimized parallelization of classification algorithms on GPU clusters.

Author contributions All authors have participated in the design of the proposed method and practical implementation. MA and AFN have coded the method. MA, SBE, MZ, MRK, and HN have completed the first draft of this paper. All authors have read and approved the manuscript.

Funding This work has been carried out in the framework of the European Union's Horizon 2020 research and innovation program under grant agreement No 957852 (Virtual Power Plant for Interoperable and Smart isLANDS'- VPP4ISLANDS).

Data availability Not applicable.

Code availability The code will be available upon request sent to the corresponding author.

Declarations

Conflict of interest The authors declare that they have no competing interests.

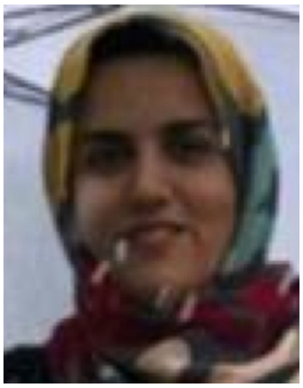
References

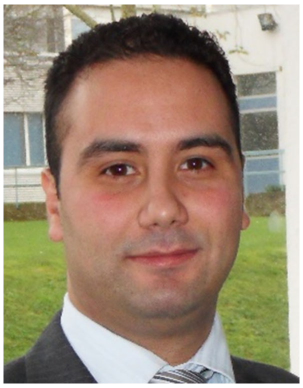
1. Yazdanie, M., Orehounig, K.: Advancing urban energy system planning and modeling approaches: gaps and solutions in perspective. Renew. Sustain. Energy Rev. **137**, 110607 (2021)
2. Abbasi, M., Yaghoobikia, M., Rafiee, M., Jolfaei, A., Khosravi, M.R.: Energy-efficient workload allocation in fog-cloud based services of intelligent transportation systems using a learning classifier system. IET Intel. Transport Syst. **14**, 1484–1490 (2020)
3. Pudjianto, D., Ramsay, C., Strbac, G.: Virtual power plant and system integration of distributed energy resources. IET Renew. Power Gener. **1**, 10–16 (2007)
4. Saboori, H., Mohammadi, M., Taghe, R.: Virtual power plant (VPP), definition, concept, components and types. In: 2011 Asia-Pacific Power and Energy Engineering Conference, pp. 1–4 (2011)
5. Chen, Y., Li, T., Zhao, C., Wei, W.: Decentralized provision of renewable predictions within a virtual power plant. IEEE Transactions on Power Systems (2020)
6. Yu, S., Fang, F., Liu, Y., Liu, J.: Uncertainties of virtual power plant: problems and countermeasures. Appl. Energy **239**, 454–470 (2019)
7. Venkatachary, S.K., Prasad, J., Samikannu, R., Alagappan, A., Andrews, L.J.B.: Cybersecurity infrastructure challenges in IoT based virtual power plants. J. Stat. Manag. Syst. **23**, 263–276 (2020)
8. Batista, E., Solanas, A.: A uniformization-based approach to preserve individuals' privacy during process mining analyses. Peer-to-Peer Netw. Appl. **14**(3), 1–20 (2021)
9. Jiang, H., Li, J., Zhao, P., Zeng, F., Xiao, Z., Iyengar, A.: Location privacy-preserving mechanisms in location-based services: a comprehensive survey. ACM Comput. Surv. **54**, 1–36 (2021)
10. Lee, J.-S., Jun, S.-P.: Privacy-preserving data mining for open government data from heterogeneous sources. Gov. Inf. Q. **38**, 101544 (2021)
11. Alamaniotis, M., Bourbakis, N., Tsoukalas, L.H.: Enhancing privacy of electricity consumption in smart cities through morphing of anticipated demand pattern utilizing self-elasticity and genetic algorithms. Sustain. Cities Soc. **46**, 101426 (2019)
12. Zajc, M., Kolenc, M., Suljanović, N.: 11—Virtual power plant communication system architecture. In: Yang, Q., Yang, T., Li, W. (eds.) Smart Power Distribution Systems, pp. 231–250. Academic Press, New York (2019)
13. Syed, S., Syed, M., Syeda, H.B., Garza, M., Bennett, W., Bona, J., et al.: API driven on-demand participant ID pseudonymization in heterogeneous multi-study research. Healthc. Inform. Res. **27**, 39–47 (2021)
14. Abbasi, M., Najafi, A., Rafiee, M., Khosravi, M.R., Menon, V.G., Muhammad, G.: Efficient flow processing in 5G-envisioned

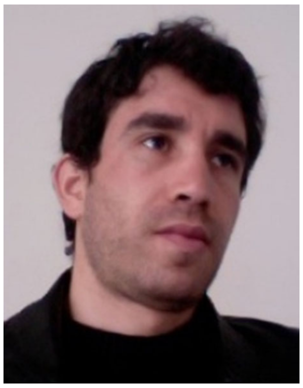
SDN-based Internet of Vehicles using GPUs. IEEE Trans. Intell. Transp. Syst. (2020)
15. Abbasi, M., Shokrollahi, A.: Enhancing the performance of decision tree-based packet classification algorithms using CPU cluster. Clust. Comput. **23**, 3203–3219 (2020)
16. Abbasi, M., Fazel, S.V., Rafiee, M.: MBitCuts: optimal bit-level cutting in geometric space packet classification. J. Supercomput. **76**, 3105–3128 (2020)
17. Abbasi, M., Tahouri, R., Rafiee, M.: Enhancing the performance of the aggregated bit vector algorithm in network packet classification using GPU. PeerJ Comput. Sci. **5**, e185 (2019)
18. Taylor, D.E.: Survey and taxonomy of packet classification techniques. ACM Comput. Surv. **37**, 238–275 (2005)
19. Henty, D.S.: Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, p. 10 (2000)
20. Pao, D., Liu, C.: Parallel tree search: an algorithmic approach for multi-field packet classification. Comput. Commun. **30**, 302–314 (2007)
21. Nottingham, A., Irwin, B.: Parallel packet classification using GPU co-processors. In: Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, pp. 231–241 (2010)
22. Hung, C.-L., Lin, Y.-L., Li, K.-C., Wang, H.-H., Guo, S.-W.: Efficient GPGPU-based parallel packet classification, Presented at the Trust, Security and Privacy in Computing and Communications (TrustCom) (2011)
23. Hung, C.-L., Guo, S.-W.: Fast parallel network packet filter system based on CUDA. Int. J. Netw. Distrib. Comput. **2**, 198–210 (2014)
24. Hung, C.-L., Lin, C.-Y., Wang, H.-H.: An efficient parallel-network packet pattern-matching approach using GPUs. J. Syst. Architect. **60**, 431–439 (2014)
25. Srinivasan, V., Suri, S., Varghese, G.: Packet classification using tuple space search. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 135–146 (1999)
26. López, P., Baydal, E.: Teaching high-performance service in a cluster computing course. J. Parallel Distrib. Comput. **117**, 138–147 (2018)
27. Rico-Gallego, J.A., Díaz-Martín, J.C., Manumachu, R.R., Lastovetsky, A.L.: A survey of communication performance models for high-performance computing. ACM Comput. Surv. **51**, 126 (2019)
28. Wu, X., Li, W.: Performance models for scalable cluster computing. J. Syst. Architect. **44**, 189–205 (1998)
29. Martin, R.P., Vahdat, A.M., Culler, D.E., Anderson, T.E.: Effects of communication latency, overhead, and bandwidth in a cluster architecture. In: ACM SIGARCH Computer Architecture News, pp. 85–97 (1997)
30. Buyya, R., Jin, H., Cortes, T.: Cluster computing. Future Gener. Comput. Syst. **18**, 5–8 (2002)
31. Talia, D.: Models and languages for high-performance computing. In: Ranganathan, S., Gribskov, M., Nakai, K., Schönbach, C. (eds.) Encyclopedia of bioinformatics and computational biology, pp. 215–220. Academic Press, Oxford (2019)
32. Smith, L., Bull, M.: Development of mixed mode MPI/OpenMP applications. Sci. Program. **9**, 83–98 (2001)
33. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 427–436 (2009)
34. Grant, R.E., Olivier, S.L.: Chapter 6—Networks and MPI for cluster computing. In: Prasad, S.K., Gupta, A., Rosenberg, A.L., Sussman, A., Weems, C.C. (eds.) Topics in parallel and distributed computing, pp. 117–153. Morgan Kaufmann, Boston (2015)
35. Nottingham, A., Irwin, B.: GPU packet classification using OpenCL: a consideration of viable classification methods. In: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, pp. 160–169 (2009)
36. Pong, F., Tzeng, N.-F.: HaRP: rapid packet classification via hashing round-down prefixes. IEEE Trans. Parallel Distrib. Syst. **22**, 1105–1119 (2011)
37. Kang, K., Deng, Y.S.: Scalable packet classification via GPU metaprogramming. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–4 (2011)
38. Varvello, M., Laufer, R., Zhang, F., Lakshman, T.: Multi-layer packet classification with graphics processing units. In: Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 109–120 (2014)
39. Abbasi, M., Rafiee, M.: A calibrated asymptotic framework for analyzing packet classification algorithms on GPUs. J. Supercomput. **75**, 6574–6611 (2019)
40. Zhou, S., Qu, Y.R., Prasanna, V.K.: Multi-core implementation of decomposition-based packet classification algorithms. In: International Conference on Parallel Computing Technologies, pp. 105–119 (2013)
41. Qu, Y.R. et al.: Optimizing many-field packet classification on fpga, multi-core general purpose processor, and gpu. In: Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 87–98 (2015)
42. Razaque, A., Jararweh, Y., Alotaibi, B., Alotaibi, M., Hariri, S., Almiani, M.: Energy-efficient and secure mobile fog-based cloud for the Internet of Things. Future Gener. Comput. Syst. **127**, 1–13 (2022)
43. Seyhan, K. et al.: Lattice-based cryptosystems for the security of resource-constrained IoT devices in post-quantum world: a survey. Clust. Comput. (2021)
44. Sah, D.K. et al.: Load-balance scheduling for intelligent sensors deployment in industrial internet of things. Clust. Comput. (2021)
45. Jafarian, T., Masdari, M., Ghaffari, A., Majidzadeh, K.: A survey and classification of the security anomaly detection mechanisms in software defined networks. Clust. Comput. **24**, 1235–1253 (2021)
46. Chiang, M.-L., et al.: SDN-based server clusters with dynamic load balancing and performance improvement. Clust. Comput. **24**, 537–558 (2021)
47. Hutter, J., Curioni, A.: Dual-level parallelism for ab initio molecular dynamics: reaching teraflop performance with the CPMD code. Parallel Comput. **31**, 1–17 (2005)
48. Cappello, F., Etiemble, D.: MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks. In: Supercomputing, ACM/IEEE 2000 Conference, p. 12 (2000)
49. Ferretti, M., Santangelo, L.: Hybrid OpenMP-MPI parallelism: porting experiments from small to large clusters. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 297–301 (2018)
50. Jiao, Y.-Y., Zhao, Q., Wang, L., Huang, G.-H., Tan, F.: A hybrid MPI/OpenMP parallel computing model for spherical discontinuous deformation analysis. Comput. Geotechn. **106**, 217–227 (2019)
51. Katz, M.J., Papadopoulos, P.M., Bruno, G.: Leveraging standard core technologies to programmatically build linux cluster appliances. In: Proceedings of IEEE International Conference on Cluster Computing, pp. 47–53 (2002)
52. Taylor, D.E., Turner, J.S.: ClassBench: a packet classification benchmark. In: Proceedings IEEE 24th Annual Joint Conference

of the IEEE Computer and Communications Societies, pp. 2068–2079 (2005)
53. Zheng, J., Zhang, D., Li, Y., Li, G.: Accelerate packet classification using GPU: a case study on HiCuts. In: Park, J.J., Stojmenovic, I., Jeong, H.Y., Yi, G. (eds.) Computer Science and its Applications: Ubiquitous Information Technologies, pp. 231–238. Springer, Berlin (2015)
54. Zhou, S., Singapura, S.G., Prasanna, V.K.: High-performance packet classification on gpu. In: 2014 IEEE on High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2014)

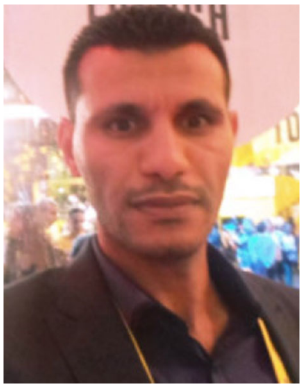
Mahdi Abbasi is an Associate Professor of Computer Architecture at Bu-Ali Sina University, Hamedan, Iran. He received his B.Sc. degree in Computer Engineering and his M.Sc. degree in Computer Architecture from Sharif University of Technology, Iran, in 2000 and 2005 respectively. He received his Ph.D. in computer Architecture, from University of Isfahan, in 2012. Since 2021, he is a Postdoctoral Researcher at the University of Aix-Marseille and he is with the LIS (UMR 720 CNRS) Lab. His main research interests include Computer Architecture, IoT, Optimization and High-Performance Computing.


Azam Fazel Najafabadi received M.Sc. degree in computer networks from Bu-Ali Sina University, in 2017. Her research interests include Computer Networks, Network Processors, distributed systems and Internet of things (IoT).


Seifeddine Ben Elghali was born in Téboulba, Tunisia, in 1981. He received the B.Sc. degree in Electrical Engineering in 2005 from ENIT, Tunis, Tunisia, the M.Sc. degree in Automatic Control in 2006 from the University of Poitiers, Poitiers, France, and the Ph.D. degree in Electrical Engineering in 2009 from the University of Brest, Brest, France.After receiving the Ph.D. degree, he joined the French Naval Academy, Brest, France as a Teaching and Research Assistant. Since 2010, his is an Associate Professor of Electrical Engineering at Aix-Marseille University, Marseille, France. His current research interests include modeling and control of renewable energy applications.


Mohamed Zerrougui is an Associate Professor in Automatic Control. He received his engineering degree in Industrial Control from the department of electronics in 2007, he obtained his Master's Degree in Automatic Control from the University of Reims Champagne Ardenne (URCA), France, in 2008. He received his PhD degree in Automatic Control from the Université Henri Poincaré of Nancy I, France, in 2011. He was a postdoctoral fellow at the Ecole Polytechnique de Bruxelles between 2011 and 2012. Since 2013, he has been an Associate Professor an Associated Professor in Automatic Control at the Université d'Aix Marseille and is with LIS (UMR CNRS 7020) lab. His main research interests are in energy management, smart grid, distributed control, filtering, estimation and diagnosis of complex systems.

Mohammad R. Khosravi is now with the Department of Computer Engineering, Persian Gulf University, Bushehr, Iran, and has been with Department of Electrical and Electronic Engineering, Shiraz University of Technology, Shiraz, Iran. Mohammad has studied electrical engineering with expertise in communications and signal processing for BSc, MSc and PhD degrees. His main interests include statistical signal and image processing, medical bioinformatics, radar imaging and satellite remote sensing, computer communications.



Habib Nasser holds three high degrees as mechatronic engineer, Master in information technology and Ph.D. in complex systems (data scientist) and is specialist in smart technologies development. At Lis-Lab Habib has developed algorithms and mechatronic system for autonomous systems based on the analysis of mutli-sensors. He was previously a scientific research (qualified ISO13485) at Symme Lab to realize an advanced simulator for infant and analyse the problems of bronchitis. He has equipped the body with flexible sensors and a web-based application to clean and analyse the different information. At Ecosteering, he has developed a patented connected occupational clothing to detect painful postures in real time, especially to promote active aging and mitigate MSDs. At RDI'UP, Habib is the CEO and co-founder and he is interested in ML field and data analytics in healthcare and energy sectors.