JOIN (Jurnal Online Informatika)
p-ISSN: 2528-1682, e-ISSN: 2527-9165
Volume 9 Number 1 | June 2024: 1-9
DOI: 10.15575/join.v9i1.1181

A Comparison of Ryu and Pox Controllers: A Parallel Implementation

Muhammad Ikhwananda Rizaldi[1], Elsa Annas Sonia Yusuf[2], Denar Regata Akbi[3], Wildan Suharso[4]
[1,2,3,4]Department of Informatics, University of Muhammadiyah Malang, Indonesia

Article Info

Article history:

Received September 07, 2023
Revised December 03, 2023
Accepted December 05, 2023
Available Online April 23, 2024

Keywords:

Networking
Parallel Programming
POX
RYU
Software Defined Network

ABSTRACT

Software Defined Network (SDN) network controllers have limitations in handling large volumes of data generated by switches, which can slow down their performance. Using parallel programming methods such as threading, multiprocessing, and MPI aims to improve the performance of the controller in handling a large number of switches. By considering factors such as memory usage, CPU consumption, and execution time. The test results show that although RYU outperforms POX in terms of faster execution time and lower CPU utilization rate, POX shows its prowess by exhibiting less memory usage despite higher CPU utilization rate than RYU. The use of the parallel approach proves advantageous as both controllers exhibit enhanced efficiency levels. Ultimately, RYU's impressive speed and superior resource optimization capabilities may prove to be more strategic than POX over time. Taking into account the specific needs and prerequisites of a given system, this research provides insights in selecting the most suitable controller to handle large-scale switches with optimal efficiency.

Corresponding Author:

Muhammad Ikhwananda Rizaldi,
Informatics Department, Faculty of Engineering, University of Muhammadiyah Malang,
Jl. Raya Tlogomas No.246, Babatan, Tegalgondo, Kec. Lowokwaru, Malang, East Java, Indonesia
Email: aldirizaldy977@webmail.umm.ac.id

1. INTRODUCTION

RYU and POX are popular network controller options used to control software defined networks (SDN). However, the performance of these network controllers has limitations when handling networks with a larger number of switches and complexity [1], [2]. The performance of RYU and POX has compared in a number of previous studies. Some of these studies [3], [4] have shown that RYU excels in managing complex networks. However, when dealing with a large number of switches, it is evident that RYU and POX have some serious limitations. These controllers have difficulty keeping up with the volume of data generated by a large number of switches, which can degrade the performance of the network [5]. Researchers have suggested using parallel programming techniques, including MPI, multiprocessing, and threading, to improve the efficiency of RYU and POX. These methods can better utilize the CPU's processing capabilities [6]–[8] to improve the performance of these two controllers [9] in terms of processing speed and memory allocation. The effectiveness of parallel methods in improving the efficiency of network controllers has not been thoroughly investigated. Applying parallel programming techniques such as MPI, multiprocessing, and threading are suggestions given by researchers. So, the purpose of this study is to find out how RYU and POX differ in terms of how well parallel programming techniques such as MPI, multiprocessing, and threading can help improve network controller performance [10], [11]. In this study, we provide an introduction to parallel programming and explain the benefits of using these techniques for SDN controllers. We also explain how to implement parallel programming in RYU and POX.

RYU and POX have been extensively studied and compared in terms of their ability to manage network switches. Examples of research showing the superior performance of RYU in handling very large network switches can be found in [12], [13]. After comparing RYU and POX, these studies concluded that RYU is superior in handling large-scale network switches. Similarly, [14] found that RYU outperforms POX in terms of robustness, efficiency, and field adaptability. According to the analysis from research [4], RYU and

POX are equally effective in handling network switches. This study compared the performance of both network controllers under various workloads and found that RYU is superior in handling networks under heavy loads. To improve their efficiency, network controllers can utilize a method called paralleling. The effectiveness of parallel methods in improving network performance has been the subject of several research efforts. With parallel implementation, researchers can apply more processor cores at lower frequencies to more network controllers, closing the performance gap in multi-controller scenarios, as done in the research referenced in [15]. Lowering the operating frequency of each core and implementing controllers in parallel results in greater energy efficiency. For a constant throughput scenario, experimental results showed a 28% improvement in processor energy efficiency compared to the single-core strategy. Research conducted by [16] performing a parallel flow installation to mitigate the effects of DoS attacks on software-defined networking (SDN) without losing the ability to monitor and control network traffic. This approach reduces the effects of DoS attacks by installing flows in parallel, which reduces control channel traffic and controller utilization.

In this study, the researchers will use parallel programming techniques, specifically MPI, multiprocessing, and threading. The goal of this research is to advance the understanding of how parallel can improve the functionality of network controllers. The results obtained from this research will be used in the development of improved SDN network controller solutions that can proficiently manage more complicated and extensive networks with higher reliability and efficiency. The researchers will use parallel programming approaches, namely MPI, multiprocessing, and threading, to compare RYU and POX. The research will also evaluate the efficacy of the network controller in managing networks with a larger number of switches. The utilization of parallel programming approaches, specifically MPI, multiprocessing, and threading, will enable the evaluation of RYU and POX as well as the assessment of the network controller's effectiveness in managing a larger number of switches.

2. METHOD

2.1. Research topology and scenario

In this study, the SDN network simulation was conducted on a computer running the Ubuntu 20.04 LTS operating system (OS), which is equipped with an Intel® CoreTM i5-10400 CPU clocked at 2.90 GHz, 8 GB RAM, and a 240 GB SSD as storage media. To ensure the success of this research, several tools were used including the Mininet [17] emulation system which uses a linear topology for network simulation. RYU and POX [14] were chosen as platforms to implement parallel programming techniques on SDN networks [18], while TOP was used to monitor hardware usage such as program execution time, CPU usage, and memory usage [19]. The decision to use the linear topology shown in Figure 1 is based on its simplicity and effectiveness in organizing the network. In this topology, switches are connected sequentially to create a straight path [18]. This setup is particularly suitable when connecting multiple switches in series as seen in SDN models involving 30 or 50 switches. The linear topology offers ease of setup and management, scalability, efficient bandwidth utilization, and reduced network latency [20]. To simulate the network organization in this research study, Mininet was used along with RYU (simple switch 13) and POX (learning l2) controllers. To explore different types of parallel programming techniques on RYU and POX [21], [22], message passing interface (MPI), multiprocessing, and threading are used. Various factors such as execution time, CPU usage, and memory usage were tested to evaluate the effectiveness of parallel implementation on RYU and POX controllers [23]. The pathfinding test scenario aims to identify all possible delivery routes. For timing testing, a stopwatch measures the average time taken by the controller to find a complete path. CPU usage testing uses the TOP application to determine the hardware CPU usage percentage (%). Similarly to using TOP, memory usage was calculated and expressed as a percentage. The resulting data in text format was extracted and further processed [24].

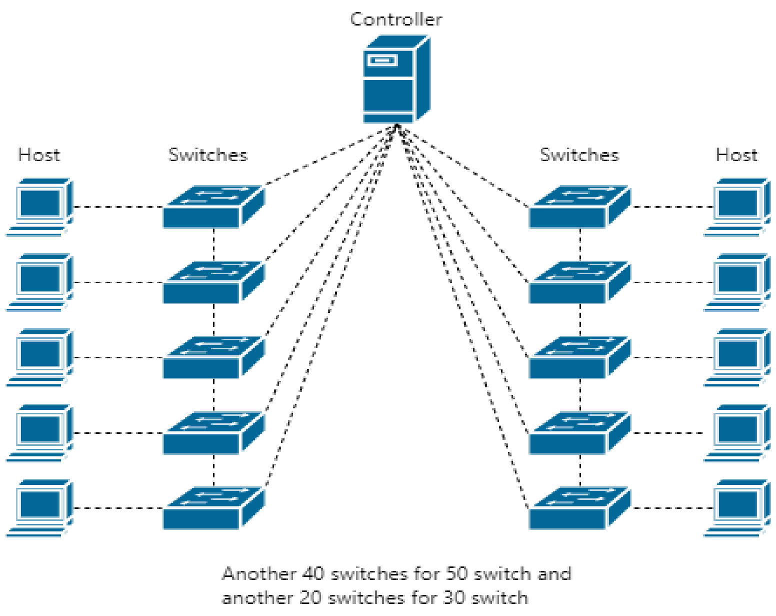
Controller
Host
Switches
Switches
Host
Another 40 switches for 50 switch and
another 20 switches for 30 switch

Figure 1. Linear topology of 30 switches and 50 switches on the mininet

2.2. *Parallel tasks for controller*

The system design that will be used in this research is illustrated in the block diagram in Figure 2. On the left side of the block diagram is the network topology created with the Mininet emulator, while the about part is the mechanism by which the parallel process takes place and the right part of the block diagram is the result generated by the parallel process. Requires initializing the controller and running parallel scripts simultaneously. Event handlers are then implemented to manage switch connection events, where the corresponding data is obtained from the switch. In handling the data generated by the switches, each script executes a parallel task for each connected switch by issuing a request for description statistics using an appropriate parallel method such as MPI, multiprocessing, or threading. After receiving the request, the script waits for a response from the switch and uses the appropriate function to process the description statistics received from the switch. After that, the description statistics will be displayed on the terminal console, the process is done in parallel based on the switches connected to the controller. Monitoring the progress of parallel tasks relies heavily on output messages and logging. Regardless of the different parallel methods used, the fundamental aspects of initialization, event handling, switch feature processing, parallel tasks, and response management remain consistent, reflecting an overarching framework for switch control on the network through a designated controller whether it is RYU or POX.

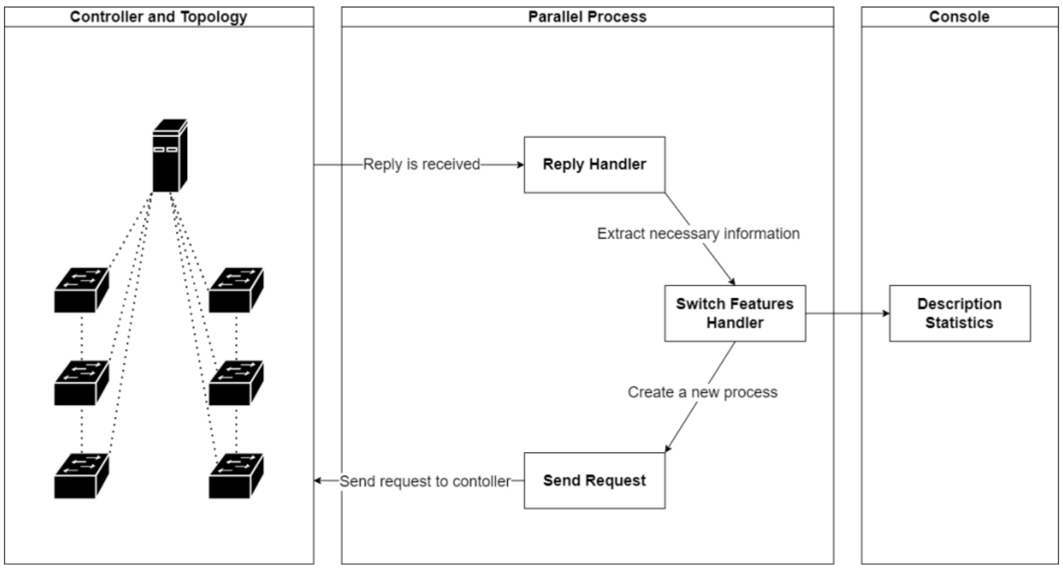

Figure 2. System's block diagram

3. RESULT AND DISCUSSION

3.1. Execution time program

The processing time of all hosts was tested 10 times with 30 and 50 switches using RYU and POX, respectively. The results are shown in Tables 1-2. This is done to find out how long it takes for the parallel application to find all paths in the linear network topology that has been built previously. The average time taken to run 10 tests with 30 switches on RYU was 21 seconds and 39 milliseconds. RYU controllers that use parallel programming techniques, such as MPI, took 21 seconds and 18 milliseconds, multiprocessing took 21 seconds and 25 milliseconds, and threading took 21 seconds and 20 milliseconds. The average execution time of 10 tests with 50 switches on RYU is 1 minute 53 seconds, 1 minute 53 seconds for RYU controllers using parallel programming techniques such as MPI, multiprocessing takes 1 minute 53 seconds, and threading takes 1 minute 51 seconds. The average execution time is based on 10 tests with 30 switches on POX, with a duration of 49 seconds and 24 milliseconds. Next are POX controllers that implement parallel programming, such as MPI, which takes 49 seconds and 19 milliseconds, multiprocessing takes 49 seconds and 24 milliseconds and threading, takes 49 seconds and 14 milliseconds. The average execution time of 10 tests with 50 switches on POX was 4 minutes 59 seconds. Controllers that use parallel programming, such as MPI, took 4 minutes and 59 seconds, multiprocessing took 5 minutes and threading took 5 minutes. At 30 and 50 switches, the average time difference between RYU and POX is very different.

According to the research results [3], in the RYU test 30 switches had the fastest average execution time. In the MPI test, it took 21 seconds and 18 milliseconds, which was the fastest time. In the POX test with 30 switches, threading was recorded the fastest with a time of 49 seconds and 14 milliseconds. On the test with 50 RYU switches, threading was the fastest, taking 1 minute, 51 seconds. In the test with 50 POX switches, POX was the fastest, taking 4 minutes, 59 seconds. From the results of testing with 30 switches, this study shows that RYU has a faster execution time than POX, and the results of testing with 50 switches also show that RYU has a faster execution time than POX.

Table 1. Results on RYU and POX using 30 switches

Experiment Number	**RYU**	**MPI**	**Multiprocessing**	**Threading**
1	0:22:00	0:20:51	0:21:40	0:21:03
2	0:24:05	0:21:06	0:21:01	0:21:35
3	0:23:06	0:20:47	0:21:18	0:21:39
4	0:22:53	0:21:17	0:21:31	0:21:07
5	0:22:07	0:21:32	0:21:14	0:21:22
6	0:20:19	0:21:02	0:21:14	0:21:03
7	0:20:32	0:21:17	0:21:56	0:21:17
8	0:21:01	0:21:07	0:21:05	0:21:20
9	0:20:29	0:22:19	0:21:37	0:21:36
10	0:20:01	0:21:39	0:21:37	0:21:13
Average	0:21:39	0:21:18	0:21:25	0:21:20
Experiment Number	**POX**	**MPI**	**Multiprocessing**	**Threading**
1	0:47:30	0:50:07	0:48:52	0:49:31
2	0:49:14	0:48:52	0:49:09	0:49:04
3	0:50:00	0:49:39	0:48:51	0:49:36
4	0:49:12	0:49:31	0:49:24	0:49:34
5	0:49:30	0:49:26	0:48:40	0:48:58
6	0:49:15	0:49:12	0:50:44	0:48:57
7	0:49:33	0:47:26	0:49:09	0:49:09
8	0:50:16	0:49:29	0:49:33	0:49:12
9	0:49:30	0:49:49	0:49:54	0:49:12
10	0:50:00	0:49:38	0:49:47	0:49:04
Average	0:49:24	0:49:19	0:49:24	0:49:14

Table 2. Results on RYU and POX using 50 switches

Experiment Number	**RYU**	**MPI**	**Multiprocessing**	**Threading**
1	1:53:05	1:53:07	1:52:53	1:51:01
2	1:53:28	1:53:17	1:53:22	1:51:26
3	1:52:53	1:53:42	1:52:49	1:51:12
4	1:53:08	1:53:16	1:53:31	1:51:45
5	1:53:24	1:53:06	1:53:10	1:50:46
6	1:53:12	1:53:29	1:53:14	1:51:30
7	1:52:33	1:52:01	1:52:58	1:52:00
8	1:52:17	1:53:33	1:52:09	1:51:35

Experiment Number	RYU	MPI	Multiprocessing	Threading
9	1:52:16	1:53:28	1:52:54	1:51:49
10	1:54:04	1:53:15	1:53:29	1:51:37
Average	1:53:02	1:53:13	1:53:03	1:51:28
Experiment Number	**POX**	**MPI**	**Multiprocessing**	**Threading**
1	4:58:20	5:00:19	5:00:11	5:00:00
2	4:58:01	4:59:16	5:00:31	5:01:22
3	5:00:01	5:00:38	5:00:23	5:00:47
4	4:59:09	4:58:02	4:59:23	4:58:28
5	5:00:01	5:02:16	4:59:41	5:00:14
6	4:59:08	4:59:40	4:59:02	4:59:33
7	4:59:28	4:59:16	5:01:18	5:00:19
8	4:58:27	4:59:18	5:00:45	5:00:46
9	4:58:56	4:59:22	5:00:15	5:00:02
10	4:59:34	5:00:27	4:59:29	4:59:01
Average	4:59:07	4:59:51	5:00:06	5:00:03

3.2. CPU Usage

CPU usage testing is done by looking at statistics about the network under test and collecting measurement data using TOP. For example, how many CPU usage results are displayed in text format. Once the data is collected, it is processed and calculated to find out what the average CPU usage is. From the data in Table 5-6, it can be seen how the testing of CPU usage at 30 and 50 switches differs. RYU uses 40% of the CPU when there are 30 switches. Controllers that use parallel programming, such as MPI, use 41%, multiprocessing uses 9%, and threading uses 39%. And at 30 switches using POX, it requires 53% of the CPU, followed by controllers using parallel programming such as MPI, which uses 54%, multiprocessing uses 54%, and threading uses 53%. At 50 switches using RYU, it takes 48% of the CPU, while controllers using parallel programming such as MPI use 50%, multiprocessing uses 11%, and threading uses 50%. In the case of using 50 POXs, 62% of the CPU is required. Controllers that use parallel programming such as MPI and multiprocessing use 61% of the CPU, and threading uses 62%. From the results of testing CPU usage with RYU and POX at 30 and 50 switches, the results are different.

The results show in Figure 3-4 that the CPU is used the least when multiprocessing is used, no matter how many controls or RYU switches are used. This is possible because each new process uses different CPU resources. But with multiprocessing, the program can use all CPU cores, which speeds up performance and processing. With multiprocessing, programs can run on more than one CPU at the same time [25]. In the case of RYU, with 30 switches, multiprocessors allow data processing to be performed on different CPUs simultaneously. This speeds up processing and reduces CPU utilization [26]. In multiprocessors, how the CPU is used also depends on how many CPUs are available and what they can do. If there are only a few CPUs available, multiprocessing techniques may not use less CPU time than other methods [27]. When using threading, CPUs tend to be used more than when using multiprocessing. However, when using threading, program threads can run at the same time in one process, which does not use too much CPU and makes it easier to start a new process [28]. On the other hand, when using MPI, CPU usage tends to be higher than when using threading or multiprocessing. This is because MPI uses connections between processes (in this case, processes on different nodes) to process data simultaneously. This connection process uses a lot of overheating, which means the CPU will be used more [29]. Multiprocessing has lower performance compared to threading and MPI in terms of CPU consumption due to scalability limitations, IO-bound tasks, memory usage, and the presence of Global Interpreter Lock (GIL) in Python. Multiprocessing can effectively use multiple cores, but its scalability is limited by the number of cores available. For IO-bound tasks where the bottleneck is not the CPU, multiprocessing may not have an advantage over threading and MPI. Multiprocessing usually requires more memory compared to threading due to the separate memory space for each process. GIL limits parallel execution of threads within a single process, so only one thread can be executed at a time, even with multiple threads created.

Table 5. CPU utilization results from RYU with switches 30 and 50

Experiment Type	RYU	MPI	Multiprocessing	Threading
CPU Usage 30 Switch	40%	41%	9%	39%
CPU Usage 50 Switch	48%	50%	11%	50%

Table 6. CPU utilization results from POX with switches 30 and 50

Experiment Type	POX	MPI	Multiprocessing	Threading
CPU Usage 30 Switch	53%	54%	54%	53%
CPU Usage 50 Switch	62%	61%	61%	62%

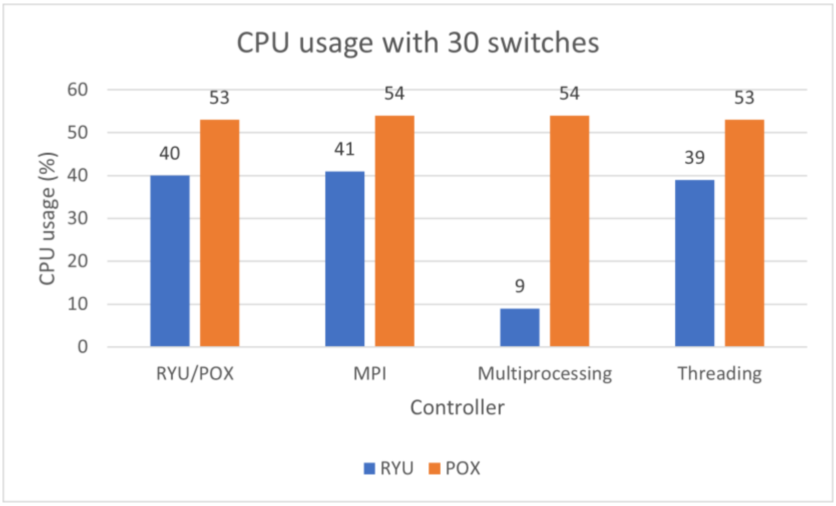

Figure 3. Diagram CPU Usage 30 Switches

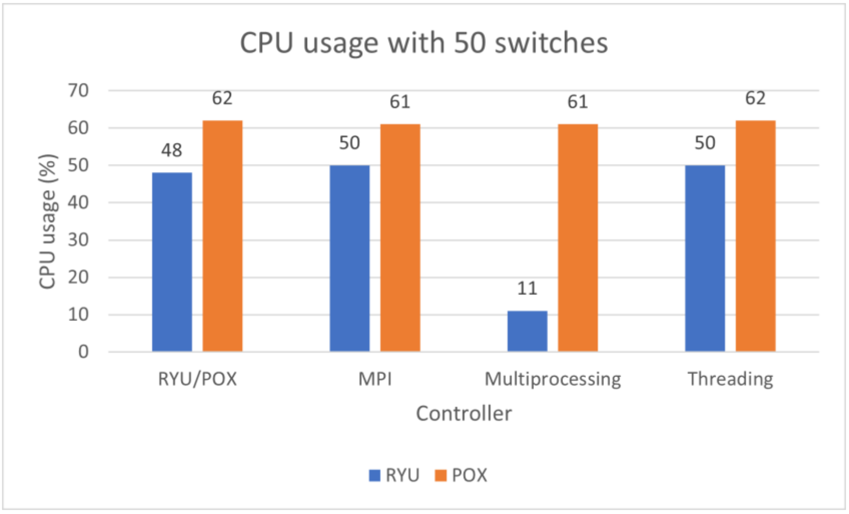

Figure 4. Diagram CPU Usage 50 Switches

3.3. Memory Usage

Memory usage testing is a way to find out how memory is used on the gadget being used. Memory usage testing is done by using TOP to collect data, which is then displayed in text format. The data is then processed, and calculations are made to find out how much memory is used on average, which is given in the form of percent (%). The test results are shown in Tables 7-8, which show that there is a big difference between how much memory RYU and POX use on switches 30 and 50. Using RYU, switch 30 uses 60% of the available memory. Controllers that use parallel programming, such as MPI, use 70%, multiprocessing uses 13%, and threading uses 60%. And on switch 30 that uses POX, it takes up 30% of the memory. Controllers that use parallel programming, such as MPI, use 40%, multiprocessing uses 30%, and threading uses 30%. In contrast, 50 switches that use RYU use 60% memory. Controllers that use parallel programming, such as MPI, use 80%, multiprocessing uses 13%, and threading uses 60%. For 50 switches that use POX, 30% memory is required. Controllers that use parallel programming, such as MPI, use 40%, multiprocessing uses 30%, and threading uses 30%. The memory usage test results show that the numbers for RYU and POX at 30 switches and 50 switches are not the same.

According to the research results in Figure 5-6, in RYU with 30 switches, multiprocessing uses the least amount of memory at 13%, while threading and RYU use the same amount at 60%, and MPI uses the most at 70%. Multiprocessing uses the least amount of memory on RYUs with 50 switches, at 13%. While threading and RYU use the same amount at 60%, MPI uses the most at 70%. Multiprocessing uses the least amount of memory on RYU with 50 switches, at 13%. RYU and threading use the same amount of memory at 60%, while MPI uses the most at 80%. Threading and RYU use the same amount of memory, 60%, and MPI uses the most memory, 80%. Multiprocessing uses the least amount of memory on RYU with 50 switches, which is 13%. When testing 30 or 50 switches in POX, all three parallel methods use the same amount of memory, 30%. MPI uses 40% memory, which is the largest amount. Different parallel techniques use different amounts of memory due to things like the type of data being processed, the size of the data being processed, the complexity of the computer, and how each parallel technique works. In general, MPI requires more memory because it uses communication between processes (in this case, processes on different nodes) to process data in parallel and because each process needs its own memory [29]. This connection process allows for overheating, which means more memory is used. On the other hand, each process in multiprocessing needs its own memory, while threading runs in a single process and uses the same memory [25]. Multiprocessing has lower power consumption compared to threading and MPI because communication between threads is faster than communication between processes, especially when using shared memory. In multiprocessing, a separate memory space for each process requires data to be copied between processes, which can be memory intensive. Threads share the same memory space, allowing them to share memory and data without the need for copying. This reduces memory consumption compared to multiprocessing, where each process has its own memory space. In multiprocessing, when passing data between processes, the data needs to be copied, which can be expensive in terms of memory usage. This is especially important when dealing with large data sets or objects. Multiprocessing in Python has limited shared memory facilities provided by the operating system. By default, no memory is shared between processes, and data needs to be explicitly shared, which can add complexity and overheat to memory management.

Table 7. Memory usage results from RYU with switches 30 and 50

Experiment Type	**RYU**	**MPI**	**Multiprocessing**	**Threading**
Memory Usage 30 Switch	60%	70%	13%	60%
Memory Usage 50 Switch	60%	80%	13%	60%

Table 8. Memory usage results from POX with switches 30 and 50

Experiment Type	**POX**	**MPI**	**Multiprocessing**	**Threading**
Memory Usage 30 Switch	30%	40%	30%	30%
Memory Usage 50 Switch	30%	40%	30%	30%

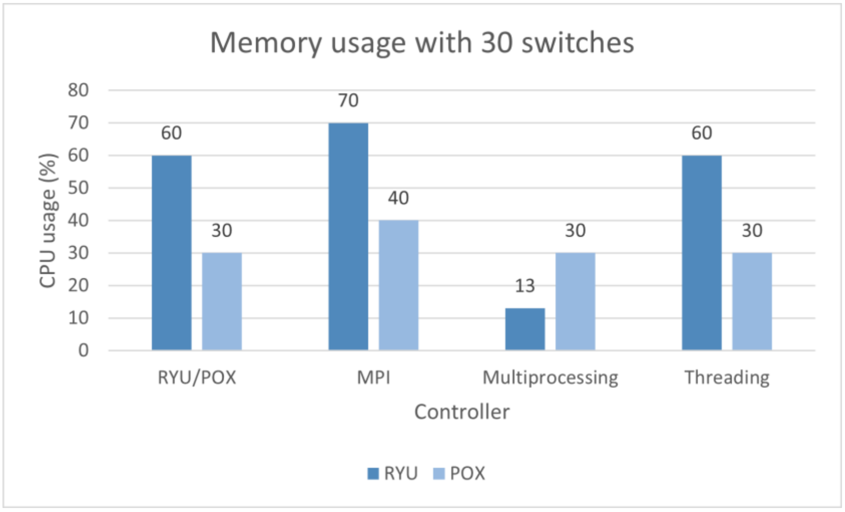

Figure 5. Diagram Memory Usage 30 Switches

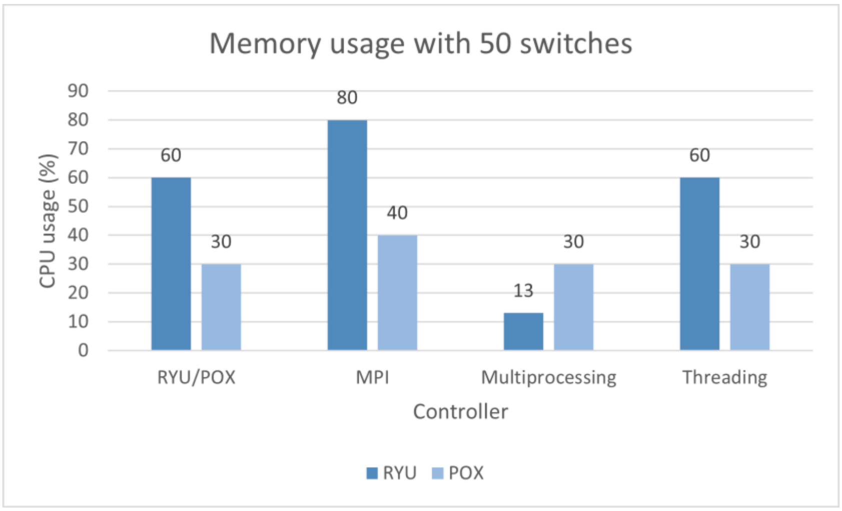

Figure 6. Diagram Memory Usage 50 Switches

4. CONCLUSION

Research findings show that RYU shows superior speed performance in terms of execution time. Specifically, RYU's mean time duration of 21 seconds outperformed POX's mean time duration of 49 seconds in the 30-switch test. In addition, RYU's mean time duration of 1 minute 52 seconds outperformed POX's mean time duration of 4 minutes 59 seconds on the 50-switch test. In addition, RYU offers superior CPU utilization compared to POX as evidenced by an overall average of 32% compared to an overall average of 53% POX for 30-switch tests. Similarly, RYU showed better CPU utilization with an overall average of 40% compared to POX's overall average of 61% for a 50-switch test.

However, it should be noted that POX excels in terms of memory usage with an overall average of 30%, whereas RYU has an overall average memory usage rate of 50% for 30 and 50 switch tests. In conclusion, implementing parallel programming on SDN controllers such as RYU and POX can significantly improve speed performance, especially in terms of execution time. However, this increase comes at the expense of increased CPU and memory usage. Therefore, we recommend considering parallel implementations of SDN controllers when dealing with a large number of switches. Ultimately, the choice between these platforms depends on the requirements of the specific use case as each platform has its own advantages and disadvantages in terms of performance and resource utilization.

ACKNOWLEDGEMENTS

The author would like to thank the University of Muhammadiyah Malang and the UMM Informatics Laboratory for supporting this research.

REFERENCES

[1] L. Mamushiane, A. Lysko, and S. Dlamini, “A comparative evaluation of the performance of popular SDN controllers,” in *2018 Wireless Days (WD)*, IEEE, Apr. 2018, pp. 54–59. doi: 10.1109/WD.2018.8361694.

[2] S. Mostafavi, V. Hakami, F. Paydar, R. Article, and S. A. Mostafavi, “Performance Evaluation of Software-Defined Networking Controllers: A Comparative Study,” *Journal of Computer and Knowledge Engineering*, vol. 2, no. 2, 2019, doi: 10.22067/cke.v2i2.84917.

[3] J. Ali, S. Lee, and B. Roh, “Performance Analysis of POX and Ryu with Different SDN Topologies,” in *Proceedings of the 2018 International Conference on Information Science and System*, New York, NY, USA: ACM, Apr. 2018, pp. 244–249. doi: 10.1145/3209914.3209931.

[4] N. Z. Abidin, A. Fiade, Arini, S. Aripiyanto, Nuryasin, and V. Handayani, “Performance Analysis of POX and RYU Controller on Software Defined Network with Spanning Tree Protocol,” in *2021 9th International Conference on Cyber and IT Service Management (CITSM)*, IEEE, Sep. 2021, pp. 1–5. doi: 10.1109/CITSM52892.2021.9588867.

[5] A. V. Priya and N. Radhika, “Performance comparison of SDN OpenFlow controllers,” *International Journal of Computer Aided Engineering and Technology*, vol. 11, no. 4/5, p. 467, 2019, doi: 10.1504/IJCAET.2019.100444.

[6] A. Blot and J. Petke, “A COMPREHENSIVE SURVEY OF BENCHMARKS FOR AUTOMATED IMPROVEMENT OF SOFTWARE’S NON-FUNCTIONAL PROPERTIES,” 2022. [Online]. Available: https://bloa.github.io/nfunc_survey/.

[7] H. Zolfaghari, D. Rossi, and J. Nurmi, “An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks,” in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, IEEE, Oct. 2019, pp. 1–7. doi: 10.1109/NORCHIP.2019.8906959.

[8] S. Bhardwaj and S. N. Panda, "Performance Evaluation Using RYU SDN Controller in Software-Defined Networking Environment," *Wirel Pers Commun*, vol. 122, no. 1, pp. 701–723, Jan. 2022, doi: 10.1007/s11277-021-08920-3.

[9] V. Kelefouras and K. Djemame, "A methodology correlating code optimizations with data memory accesses, execution time and energy consumption," *J Supercomput*, vol. 75, no. 10, pp. 6710–6745, Oct. 2019, doi: 10.1007/s11227-019-02880-z.

[10] Y. Babuji *et al.*, "Parsl: Pervasive Parallel Programming in Python," *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 25–36, Jun. 2019, doi: 10.1145/3307681.3325400.

[11] X. Wang, Q. Zhang, J. Ren, S. Xu, S. Wang, and S. Yu, "Toward efficient parallel routing optimization for large-scale SDN networks using GPGPU," *Journal of Network and Computer Applications*, vol. 113, pp. 1–13, Jul. 2018, doi: 10.1016/j.jnca.2018.03.031.

[12] N. M. Kazi, S. R. Suralkar, and U. S. Bhadade, "Evaluating the Performance of POX and RYU SDN Controllers Using Mininet," 2021, pp. 181–191. doi: 10.1007/978-3-030-91244-4_15.

[13] K. Rohitaksha and A. B. Rajendra, "Analysis of POX and Ryu Controllers Using Topology Based Hybrid Software Defined Networks," 2020, pp. 49–56. doi: 10.1007/978-3-030-34515-0_6.

[14] S. Askar and F. Keti, "Performance Evaluation of Different SDN Controllers: A Review," 2021, doi: 10.5281/zenodo.4742771.

[15] T. F. Oliveira, S. Xavier-de-Souza, and L. F. Silveira, "Improving Energy Efficiency on SDN Control-Plane Using Multi-Core Controllers," *Energies (Basel)*, vol. 14, no. 11, p. 3161, May 2021, doi: 10.3390/en14113161.

[16] M. Imran, M. H. Durad, F. A. Khan, and A. Derhab, "Reducing the effects of DoS attacks in software defined networks using parallel flow installation," *Human-centric Computing and Information Sciences*, vol. 9, no. 1, p. 16, Dec. 2019, doi: 10.1186/s13673-019-0176-7.

[17] Md. T. Islam, N. Islam, and Md. Al Refat, "Node to Node Performance Evaluation through RYU SDN Controller," *Wirel Pers Commun*, vol. 112, no. 1, pp. 555–570, May 2020, doi: 10.1007/s11277-020-07060-4.

[18] D. Kumar and M. Sood, "Performance Analysis of Impact of Network Topologies on Different Controllers in SDN," 2021, pp. 725–735. doi: 10.1007/978-981-15-5148-2_63.

[19] K. Houenoussi, R. Boukheloua, J.-P. Vernadet, D. Gautheret, G. Vergnaud, and C. Pourcel, "TOP the Transcription Orientation Pipeline and its use to investigate the transcription of non-coding regions: assessment with CRISPR direct repeats and intergenic sequences," 2020, doi: 10.1101/2020.01.15.903914.

[20] J. N. C. Especial, A. Rey, and P. F. N. Faísca, "A Note on the Effects of Linear Topology Preservation in Monte Carlo Simulations of Knotted Proteins," *Int J Mol Sci*, vol. 23, no. 22, Nov. 2022, doi: 10.3390/ijms232213871.

[21] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, Sep. 2019, pp. 1–11. doi: 10.1109/CLUSTER.2019.8891015.

[22] Naimullah, S. I. Ullah, A. W. Ullah, A. Salam, M. Imad, and F. Ullah, "Performance Analysis of POX and RYU Based on Dijkstra's Algorithm for Software Defined Networking," 2021, pp. 24–35. doi: 10.1007/978-3-030-77246-8_3.

[23] D. CABARKAPA and D. RANCIC, "Performance Analysis of Ryu-POX Controller in Different Tree-Based SDN Topologies," *Advances in Electrical and Computer Engineering*, vol. 21, no. 3, pp. 31–38, 2021, doi: 10.4316/AECE.2021.03004.

[24] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, Dec. 2018, pp. 312–324. doi: 10.1109/RTSS.2018.00047.

[25] M. N. V. Sesha Saiteja, K. Sai Sumanth Reddy, D. Radha, and M. Moharir, "Multi-Core Architecture and Network on Chip: Applications and Challenges," *J Comput Theor Nanosci*, vol. 17, no. 1, pp. 239–245, Jan. 2020, doi: 10.1166/jctn.2020.8657.

[26] I. K. Wibowo, A. R. A. Besari, and Muh. R. Rizqullah, "Implementation of Multiprocessing and Multithreading for End Node Middleware Control on Internet of Things Devices," *Inform : Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi*, vol. 6, no. 1, pp. 54–60, Jan. 2021, doi: 10.25139/inform.v6i1.3346.

[27] A. Mohsin Abdulazeez, N. Rashid Ali, and Q. Zeebaree, "Effect of Multi-Core Processors on CPU-Usage with Heavy-Load Problem Solving," 2020. [Online]. Available: www.ijicc.net

[28] M. Karsten and S. Barghi, "User-level Threading," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 1, pp. 1–30, May 2020, doi: 10.1145/3379483.

[29] E. Soto Gómez, "MPI vs OpenMP: Un caso de estudio sobre la generación del conjunto de Mandelbrot MPI vs OpenMP: A case study on parallel generation of Mandelbrot set," *Revista Innovación y Software*, vol. 1, no. 2, 2020.