Weight-based Load Balancing in Raspberry Pi MPICH Heterogeneous Cluster with Fuzzy Estimation of Node Computational Performance

Dmytro Zubov [1] and Andrey Kupin [2]

[1] *University of Central Asia, 138 Toktogul St., Bishkek, 720001, Kyrgyzstan*
[2] *Kryvyi Rih National University, 11 Matusevycha St., Kryvyi Rih, 50027, Ukraine*

Abstract

Dynamic load balancing is a key methodology that is used to speed up computing clusters nowadays. The work sharing and the work stealing are two opposite algorithms with various implementations based on predefined metrics like performance, response time, fault tolerance, resource utilization, and scalability. For the heterogeneous soft-/hardware in the Raspberry Pi MPICH cluster, weight-based load balancing is the most efficient approach since different parameters such as the type of CPU and OS, RAM, workload, microSD card capacity, power consumption, and OS release date can be taken into consideration. These parameters are described numerically and linguistically, and hence the fuzzy logic is employed to estimate the performance weights of the cluster nodes. Then, this information is used to distribute the workload among the cluster nodes to minimize the execution time. In this project, fuzzification, formation of fuzzy rules, fuzzy inference, and defuzzification are applied to find the performance weights. The defuzzification is based on the multiplication operation. In the cluster with two Raspberry Pi 4B boards with 2 GB RAM and Raspberry Pi 64-bit OS with desktop and one Raspberry Pi 3B board with 1 GB RAM and Raspberry Pi 32-bit OS Lite, recommended performance weights are (5, 5, 1), respectively. The Python program for the prime numbers finding algorithm implements the proposed weight-based load balancing approach that is approximately five times faster than the basic algorithm with equal loading for the maximum integer number 300000.

Keywords

Weight-based load balancing, MPICH cluster, fuzzy logic, Raspberry Pi

1. Introduction

MPICH is a free high-performance implementation of the Message Passing Interface (MPI) standard [1]. It has been tested on several platforms such as Linux, Mac OS, Solaris, and Windows. As of June 2016, this technology was exclusively used on nine of the top ten supercomputers, including Sunway TaihuLight which is ranked third in the TOP500 list as of November 2018. Concerning power consumption and performance, the Raspberry Pi MPICH cluster [2-4] is one of the most efficient approaches nowadays. Because of the heterogeneous soft-/hardware, e.g., Raspberry Pi 32/64-bit and Ubuntu operating systems (OSs) and Raspberry Pi 3B/3B+/4B boards with 1/2/4/8 GB RAM, the dynamic load balancing is one of the main challenges that require to distribute the workload across all the nodes to minimize the execution time.

The work sharing and the work stealing are two opposite algorithms in dynamic load balancing [5] that can be split up into various implementations based on predefined metrics such as throughput, response time, overhead, performance, fault tolerance, migration time, resource utilization, and

ICTERI-2021, Vol I: Main Conference, PhD Symposium, Posters and Demonstrations, September 28 – October 2, 2021, Kherson, Ukraine
EMAIL: dzubov@ieee.org (A. 1); kupin@knu.edu.ua (A. 2)
ORCID: 0000-0002-6807-0924 (A. 1); 0000-0001-7569-1721 (A. 2)

scalability [6]. Biased random sampling, active clustering, honeybee foraging, join idle queue, load balance min-min, and ant colony optimization are examples of these implementations [6]. Since the soft-/hardware configuration of the MPICH cluster is not fixed, active clustering is the most suitable approach, where the master node analyzes the performance of existing nodes and then distributes the workload in accordance with that. This approach is implemented in Oracle clusters as the weight-based load balancing [7]. For the initial phase, when existing nodes start communicating with each other, the Python software development is a trivial task: two blocking, send and recv, and two non-blocking, isend and irecv, communication methods send and receive data, respectively [8]. Hence, the key problem is the calculation of the performance weights associated with the cluster nodes. Concerning the processor performance, this problem is solved by the Exploitation of the Fastest Processor method which is a centralized algorithm for dynamic load distribution of parallel applications based on the Single Program Multiple Data paradigm [9]. However, other parameters, e.g., OS and RAM, are not taken into consideration in [9].

Performance weights are calculated based on heterogeneous parameters such as processor, OS, RAM, capacity of MicroSD card, current workload on the node. These parameters can be described numerically, e.g., 1 GB RAM, and linguistically, e.g., processor ARM Cortex-A53 1.2 GHz is "significantly below norm", and hence the fuzzy logic [10-19] is proposed to estimate the performance weights associated with the cluster nodes.

This paper presents an approach of the performance weights assessment using fuzzy logic to optimize the workload distribution on the nodes in the Raspberry Pi MPICH heterogeneous cluster. The prime numbers funding algorithm is proposed to demonstrate a common technique used in parallel programming – solving a smaller case to speed up the solution of the full problem [20]. The modified version of the Python program to find prime numbers [4] was developed to speed up the Raspberry Pi MPICH heterogeneous cluster.

The paper is organized as follows: In Section 2, a basic implementation of the prime numbers finding algorithm with equal load balancing and its performance evaluation is presented, as well as Mamdani, Sugeno, and Tsukamoto fuzzy inference systems are compared and analyzed for the performance weights assessment. In Section 3, the fuzzy inference for the performance weights assessment of the Raspberry Pi MPICH heterogeneous cluster nodes is implemented, as well as equal and weight-based load balancing approaches are compared using the prime numbers finding algorithm. Conclusions are summarized in Section 4.

2. Background

2.1. Prime Numbers Finding Algorithm: Basic Implementation with Equal Load Balancing and Its Performance Evaluation

The basic implementation of the prime numbers finding algorithm with equal load balancing written in Python for the Raspberry Pi MPICH cluster [4] can schematically be represented as the pseudocode in Figure 1.

This algorithm equally loads all the working nodes in a cluster. If `end_number` equals 100, three nodes analyze numbers as follows:

1. Master node, Raspberry Pi 4B board, `my_rank`=0: 1, 7, 13, 19, 25, 31, 37, 43, 49, 55, 61, 67, 73, 79, 85, 91, 97.

2. Second node, Raspberry Pi 4B board, `my_rank`=1: 3, 9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99.

3. Third node, Raspberry Pi 3B board, `my_rank`=2: 5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89, 95.

Python program [4] was executed on the Raspberry Pi MPICH heterogeneous cluster with three nodes:

1. Two Raspberry Pi 4B boards: 2 GB RAM; Raspberry Pi OS with desktop, 64-bit beta test version, release date – August 24, 2020.

2. One Raspberry Pi 3B board: 1 GB RAM; Raspberry Pi OS Lite, 32-bit version, release date – January 11, 2021.

```
(1) Start
(2) Find the cluster size cluster_size and the rank my_rank of the current cluster node
(3) Set integer variable candidate_number=(my_rank*2+1)
(4) Initialize integer variable end_number with the value provided by the user
(5) Create an empty list primes of the discovered primes in the current node
(6) Repeat the steps until candidate_number≤end_number
 (6.1) Set Boolean variable found_prime=True
 (6.2) Set integer variable div_number=2
 (6.3) Repeat the steps until div_number≤candidate_number
  (6.3.1) If the reminder of candidate_number divide div_number equals 0,
          Set found_prime=False
          Go to step (6.4)
  (6.3.2) Increment div_number
 (6.4) If found_prime=True,
       Append candidate_number to the list primes
 (6.5) Set candidate_number=candidate_number+cluster_size*2
(7) Gather the list primes from the current node with other nodes' lists in the list results
    on the master node
(8) If my_rank=0
    Find the number of discovered primes in the list results and display it to the user
(9) Stop
```

Figure 1: Pseudocode of the prime numbers finding algorithm written in Python for the Raspberry Pi MPICH cluster [4]

Python program [4] generates the first prime number 1, but it must be 2; other values are correct.

The Raspberry Pi rack mount and the router DIR-300A with a built-in 4-port switch are shown in Figure 2. Master, second, and third nodes have IP addresses 192.168.0.92, 192.168.0.41, and 192.168.0.13, respectively. Boards are connected only by Ethernet cables via the router DIR-300A and the power supply since they are remotely controlled by the Secure Shell (SSH) protocol, as well as the standard VNC (Virtual Network Computing) server was installed on the master node to control the Raspberry Pi 4B through the RealVNC remote access software [21]. The cluster has a two-dimensional mesh topology [22].

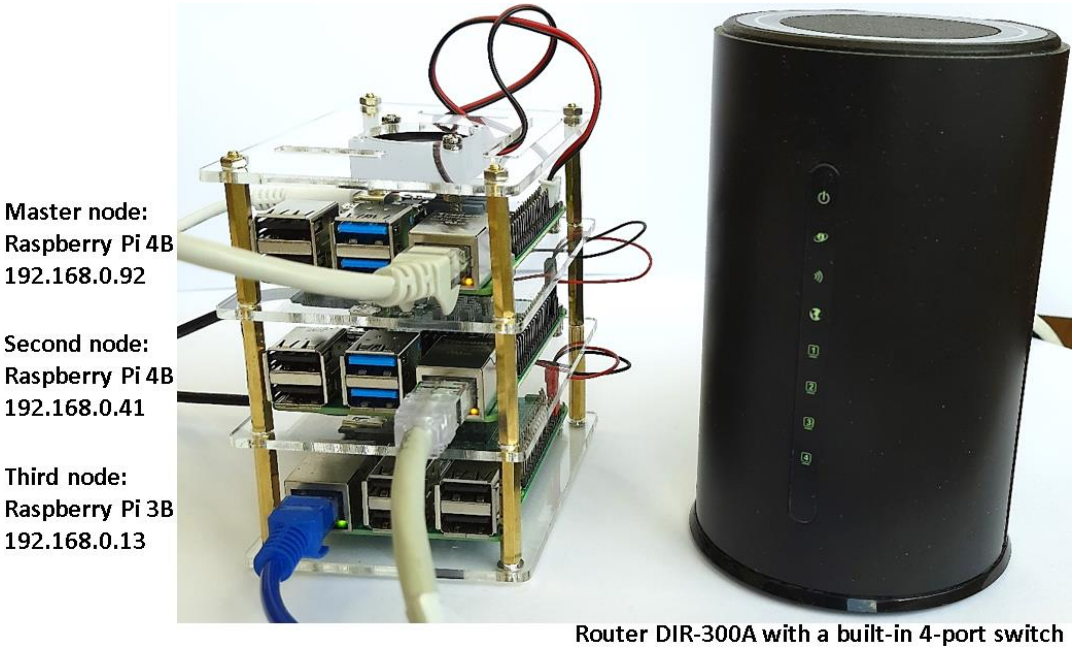

Figure 2: MPICH heterogeneous cluster: The Raspberry Pi rack mount and the router DIR-300A with a built-in 4-port switch

Execution times for one (data series 1, performance weights (1, 0, 0), Raspberry Pi 4B board), two (data series 2, performance weights (1, 1, 0), two Raspberry Pi 4B boards), and three (data series 3, performance weights (1, 1, 1), two 4B and one 3B Raspberry Pi boards) cluster nodes are presented in Figure 3 at different values of the variable `end_number`. Here, the execution time for a specific point is the average of three appropriate values. Maximum two MPICH commands `mpiexec` [4] and an additional Python program were run in parallel in different terminals on Raspberry Pi 4B boards and only one MPICH Python program was run on Raspberry Pi 3B board to avoid overloading and overheating of the CPU and RAM. Analysis of the data series presented in Figure 3 shows that the basic implementation of the prime numbers finding algorithm has the lowest performance with three nodes. This result does not contradict Amdahl's law [22] since the third node with Raspberry Pi 3B board is significantly slower than the first and the second ones. The fluctuations in data series can be explained by the non-linear CPU speed-temperature dependencies of Raspberry Pi boards [23] and different Ethernet connectivity (maximum throughput 300 Mbps for 3B and 1 Gbps for 4B), as well as the unstable and low-speed network bandwidth leads to substantial performance degradation in the grid environment [24, 25].

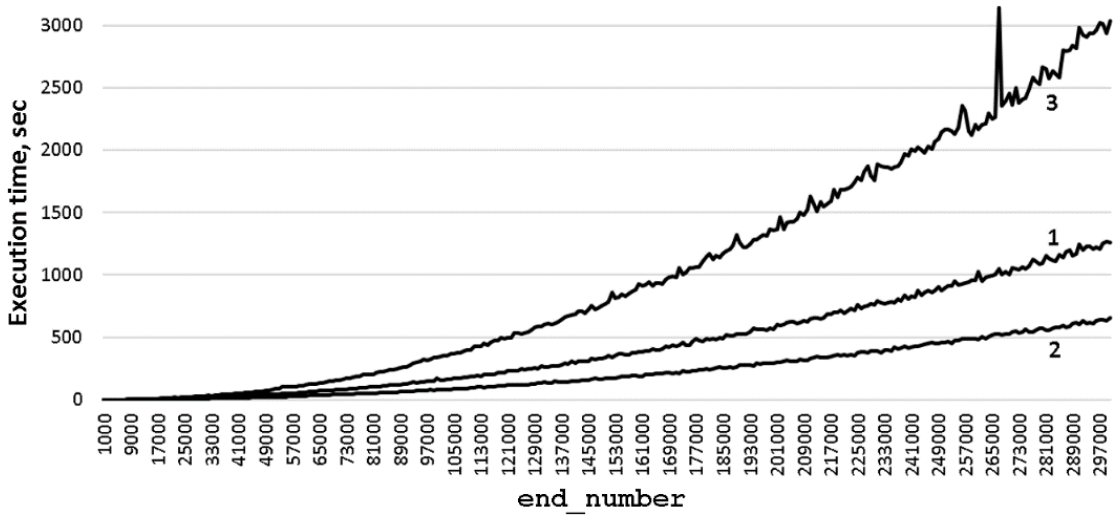

Figure 3: Execution times for one (data series 1), two (data series 2), and three (data series 3) cluster nodes at different values of the variable end_number: Basic implementation of the prime numbers finding algorithm [4]

Figure 4 represents the Raspberry Pi command `top` [26] showing the list of processes currently running on the cluster nodes with equally distributed workload: the processor's share of the CPU time is close to 0 % for the second node (finished its part), it is usually around 10 % for the first node (finished its part and waiting to gather data), and it is around 25 % for the third node, i.e., the first and the second nodes finish their parts and then they wait for the third node to accomplish its work.

2.2. Comparative Analysis of Mamdani, Sugeno, and Tsukamoto Fuzzy Inference Systems for the Performance Weights Assessment

The fuzzy logic theory was introduced by Dr. Lotfi Zadeh in 1965 and was advanced by many other researchers since that time, e.g., [11-18]. In fuzzy logic, four main stages are fuzzification, formation of fuzzy rules, fuzzy inference, and defuzzification. Fuzzy inference methods that are generally used are Mamdani, Sugeno, and Tsukamoto. Concerning the defuzzification, they can be differentiated as follows: fuzzy Tsukamoto uses the centralized mean algorithm, fuzzy Mamdani uses the centroid method [17], fuzzy Sugeno generates fuzzy singleton values [19]. Many other defuzzification methods have been proposed, but none of them generates the output for the common sense of knowledge

modeling [19]. The computational results of defuzzification often conflict [13-18], as well as they do not have a uniform framework [19].

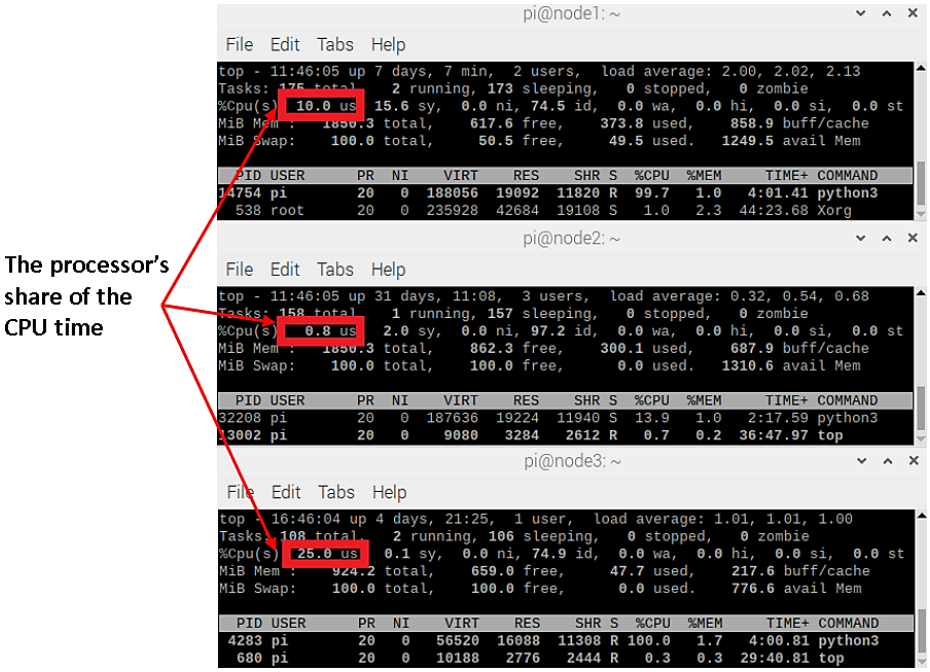

Figure 4: Raspberry Pi command top: An example of the processes currently running on the cluster nodes with the equally distributed workload

In this project, the approach [10-12], where the defuzzification is based on the multiplication operation, is proposed to estimate the performance weights of the cluster nodes. The cluster node j, $j \in [1,N]$ and N is the general number of cluster nodes, is described by the vector $X_j = \{x_{j1}, x_{j2}, \dots, x_{jK}\}$, K is the general number of soft-/hardware characteristics. Fuzzy set g_i, $i \in [1,M]$ and M is the general number of the cluster parameters z_i such as processor/OS/RAM, contains the requirements to the cluster parameters z_i. Components g_i, $G=\{g_1,g_2,\dots,g_M\}$, are specified by the membership functions $\mu_i(z_i) \rightarrow [0,1]$. Finding vector G is the solution represented by the fuzzy set $V=\{v_j\}$ that is the intersection of $\{X_j\}$ and G. Estimation of the performance weights includes the following steps:

1. Determine set $\{X_j\}$, $j \in [1,N]$.

2. Calculate membership functions $\mu_i(z_i)$ as follows (z_{ij} must be positive since the cluster soft-/hardware are described by positive numbers in this project):

$$\mu_{ij} = \frac{z_{ij}}{max\{z_{ij}\}}, \text{ if the best characteristic equals } \max\{z_{ij}\}; \quad (1)$$

$$\mu_{ij} = \frac{min\{z_{ij}\}}{z_{ij}}, \text{ if the best characteristic equals } \min\{z_{ij}\}. \quad (2)$$

Linguistic variables in the membership functions $\mu_i(z_i)$ (1) and (2) are as follows:

- "significantly above norm" that is equal to 1,
- "above norm" that is equal to 3,
- "norm" that is equal to 5,
- "below norm" that is equal to 3,
- "significantly below norm" that is equal to 1.

Membership functions (1) and (2) calculate μ_{ij}, $0 \le \mu_{ij} \le 1$, and hence the following rule must be taken into consideration: if $\alpha_1 \ge \alpha_2$, then $\mu_{ij}^{\alpha_1} \le \mu_{ij}^{\alpha_2}$.

3. Convert the vector $\{X_j\}$ to the vector $\{z_i\}$ of the cluster characteristics, and then to the fuzzy intervals $\mu_i(z_i)\rightarrow[0,1]$.

4. Find vector V as follows:

$$v_j = \prod_{i=1}^{M} \mu_{ij}^{\alpha_i}, \tag{3}$$

where α_i – weights that represent the ranks of requirements $\{z_i\}$.

5. Calculate the performance weights of the cluster nodes using vector V.

3. Weight-Based Load Balancing in Raspberry Pi MPICH Heterogeneous Cluster with Fuzzy Estimating of the Nodes Computational Performance

3.1. Fuzzy Inference for the Performance Weights Assessment of the Raspberry Pi MPICH Heterogeneous Cluster Nodes

Linguistic and numeric variables z_{ij} and weights α_i are shown in Table 1 for the Raspberry Pi MPICH heterogeneous cluster soft-/hardware. Data presented in Table 1 is the authors' subjective view based on the practical experience with respect to the performance.

Performance weights of the cluster nodes are calculated in Table 2. Analysis of the fuzzy set V shows that the calculated output variable for the third node is approximately five times less compared with the first and the second nodes: 0.975/0.229≈4.258 and 0.985/0.229≈4.301, i.e., the closest larger integer number is 5. Hence, recommended performance weights are (5, 5, 1) for the first, the second, and the third nodes, respectively.

3.2. Prime Numbers Finding Algorithm: Weight-Based Load Balancing Implementation and Its Performance Evaluation

The prime numbers finding algorithm with the weight-based load balancing using recommended performance weights for the Raspberry Pi MPICH cluster nodes can schematically be represented as the pseudocode in Figure 5. This algorithm loads all the working cluster nodes in accordance with their performance weights. If `end_number` equals 100 and the performance weights are (5, 5, 1), three nodes analyze numbers as follows:

1. Master node, Raspberry Pi 4B board, `my_rank_new`={6, 7, 8, 9, 10}: 13, 35, 57, 79, 15, 37, 59, 81, 17, 39, 61, 83, 19, 41, 63, 85, 21, 43, 65, 87. The master node is loaded less than the second node that compensates the additional workload related to acquiring and gathering data from other nodes.

2. Second node, Raspberry Pi 4B board, `my_rank_new`={0, 1, 2, 3, 4}: 1, 23, 45, 67, 89, 3, 25, 47, 69, 91, 5, 27, 49, 71, 93, 7, 29, 51, 73, 95, 9, 31, 53, 75, 97.

3. Third node, Raspberry Pi 3B board, `my_rank_new`={5}: 11, 33, 55, 77, 99.

The developed Python program was executed on the Raspberry Pi MPICH cluster with the above-mentioned three nodes. The execution times for performance weights (4, 4, 1), (5, 5, 1), and (6, 6, 1) are presented in Figure 6 at different values of the variable `end_number`: dash, grey, and black lines, respectively. Here, the execution time for a specific point is the average of three appropriate values as well. Also, maximum two MPICH commands `mpiexec` and an additional Python program were run in parallel in different terminals on Raspberry Pi 4B boards, and only one MPICH Python program was run on Raspberry Pi 3B board to avoid overloading and overheating of the CPU and RAM. Analysis of the data series presented in Figure 6 shows that the mean signed deviation for data series with performance weights (5, 5, 1)-(6, 6, 1) is -3.01 sec and (5, 5, 1)-(4, 4, 1) is -155.74 sec, i.e., data series with recommended performance weights (5, 5, 1) has less average execution time across the entire set of all observations. Also, a three-tuple (5, 5, 1) is preferable compared with (6, 6, 1) since it has less workload for the first and the second nodes along with a greater workload for the third node that corresponds with Table 2. This result can be explained by the minimum time-out of the third node with a low-performance Raspberry Pi 3B board. The fluctuations in data series appear because of unstable network bandwidth and non-linear CPU speed-temperature dependencies of Raspberry Pi boards too.

Table 1

Linguistic and numeric variables z_{ij} and weights α_i for the Raspberry Pi MPICH heterogeneous cluster soft-/hardware

The soft-/hardware characteristic (designation; α_i)	The cluster nodes – X_j, $N=3$		
	Node 1 (master)	Node 2	Node 3
Processor (z_1; α_1=0.8)	ARM Cortex-A72 1.5 GHz ("norm"; z_{11}=5)	ARM Cortex-A72 1.5 GHz ("norm"; z_{12}=5)	ARM Cortex-A53 1.2 GHz ("significantly below norm"; z_{13}=1)
Type of OS (z_2; α_2=0.1)	Raspberry Pi 64-bit ("norm"; z_{21}=5)	Raspberry Pi 64-bit ("norm"; z_{22}=5)	Raspberry Pi 32-bit ("significantly below norm"; z_{23}=1)
RAM (z_3; α_3=0.02)	2 GB LPDDR4 SDRAM ("norm"; z_{31}=5)	2 GB LPDDR4 SDRAM ("norm"; z_{32}=5)	1 GB LPDDR2 SDRAM ("below norm"; z_{33}=3)
Additional workload such as the data gathering (z_4; α_4=0.02)	Master node with worker load ("above norm"; z_{41}=3)	Worker node ("norm"; z_{42}=5)	Worker node ("norm"; z_{43}=5)
MicroSD card capacity (z_5; α_5=0.02)	8 GB (z_{51}=8)	8 GB (z_{52}=8)	4 GB (z_{53}=4)
Power consumption, 400 % processor load [27] (z_6; α_6=0.02)	6.4 W (z_{61}=6.4)	6.4 W (z_{62}=6.4)	5.1 W (z_{63}=5.1)
OS release date (z_7; α_7=0.02)	August 24, 2020 ("below norm"; z_{71}=3)	August 24, 2020 ("below norm"; z_{72}=3)	January 11, 2021 ("norm"; z_{73}=5)

Table 2

Assessment of the performance weights for the cluster nodes

z_i	Formula (1) or (2); $\max\{z_{ij}\}$ or $\min\{z_{ij}\}$	μ_{i1}	μ_{i2}	μ_{i3}
z_1	(1); $\max\{z_{1j}\}$=5	μ_{11}=5/5=1	μ_{12}=5/5=1	μ_{13}=1/5=0.2
z_2	(1); $\max\{z_{2j}\}$=5	μ_{21}=5/5=1	μ_{22}=5/5=1	μ_{23}=1/5=0.2
z_3	(1); $\max\{z_{3j}\}$=5	μ_{31}=5/5=1	μ_{32}=5/5=1	μ_{33}=3/5=0.6
z_4	(1); $\max\{z_{4j}\}$=5	μ_{41}=3/5=0.6	μ_{42}=5/5=1	μ_{43}=5/5=1
z_5	(1); $\max\{z_{5j}\}$=8	μ_{51}=8/8=1	μ_{52}=8/8=1	μ_{53}=4/8=0.5
z_6	(2); $\min\{z_{6j}\}$=5.1	μ_{61}=5.1/6.4=0.8	μ_{62}=5.1/6.4=0.8	μ_{63}=5.1/5.1=1
z_7	(1); $\max\{z_{7j}\}$=3	μ_{71}=3/5=0.6	μ_{72}=3/5=0.6	μ_{73}=5/5=1
v_j		$1^{0.8}\times1^{0.1}\times1^{0.02}\times$ $\times0.6^{0.02}\times1^{0.02}\times$ $\times0.8^{0.02}\times0.6^{0.02}=$ $=0.975=v_1$	$1^{0.8}\times1^{0.1}\times1^{0.02}\times$ $\times1^{0.02}\times1^{0.02}\times0.8^{0.02}\times$ $\times0.67^{0.02}=0.985=v_2$	$0.2^{0.8}\times0.2^{0.1}\times$ $\times0.6^{0.02}\times1^{0.02}\times$ $\times0.5^{0.02}\times1^{0.02}\times$ $\times1^{0.02}=0.229=v_3$
Recommended performance weight		0.975/0.229≈4.258≈5	0.985/0.229≈4.301≈5	1

Figure 7 represents the Raspberry Pi command `top` showing the list of processes currently running on the cluster nodes with weight-based load balancing: the processor's share of the CPU time is close to 0 % for the third node (finished its part), it is usually around 25 % for the first and the second nodes, i.e., the third node finishes its part, and then it waits for the first and the second nodes to accomplish their work.

```
(1) Start
(2) Find the cluster size cluster_size and the rank my_rank of the current cluster node
(3) Initialize integer variable end_number with the value provided by the user
(4) Create an empty list primes of the discovered primes in the current node
(5) If my_rank<>0
 (5.1) Send data on the rank and the hostname to the master node with blocking
 (5.2) Receive data on the total number of threads total and new values for the node rank –
       start and end
 (5.3) Set my_rank_new=start, cluster_size=total
 (5.4) Repeat the steps until my_rank_new≤end
  (5.4.1) Set variable candidate_number=(my_rank_new*2+1)
  (5.4.2) Repeat the steps until candidate_number≤end_number
   (5.4.2.1) Set Boolean variable found_prime=True
   (5.4.2.2) Set integer variable div_number=2
   (5.4.2.3) Repeat the steps until div_number≤candidate_number
    (5.4.2.3.1) If the reminder of candidate_number divide div_number equals 0,
                Set found_prime=False
                Go to step (5.4.2.4)
    (5.4.2.3.2) Increment div_number
   (5.4.2.4) If found_prime=True,
             Append candidate_number to the list primes
   (5.4.2.5) Set candidate_number=candidate_number+cluster_size*2
 (5.5) Gather the list primes from the current node with other nodes' lists in the list results
       on the master node
(6) If my_rank_new=0
 (6.1) Create a list NodesDict with info on the performance weights of all nodes in the cluster
 (6.2) Receive data from the working nodes in the cluster
 (6.3) Calculate the integer variable AllNodesLoad as the sum of the performance weights of
       working nodes
 (6.4) Set integer variable start=0
 (6.5) Repeat the steps for all working nodes except the master
  (6.5.1) Set variable end=start+NodesDict[working node]-1
  (6.5.2) Send the data with variables start, end, and AllNodesLoad to the working node
          without blocking
  (6.5.3) Set start=end+1
 (6.6) Set cluster_size=AllNodesLoad
 (6.7) Set end=start+NodesDict[master node]-1
 (6.8) Execute steps (5.4) and (5.5)
 (6.9) Find the number of discovered primes in the list results and display it to the user
(7) Stop
```

Figure 5: Pseudocode of the prime numbers finding algorithm with the weight-based load balancing in the Raspberry Pi MPICH heterogeneous cluster

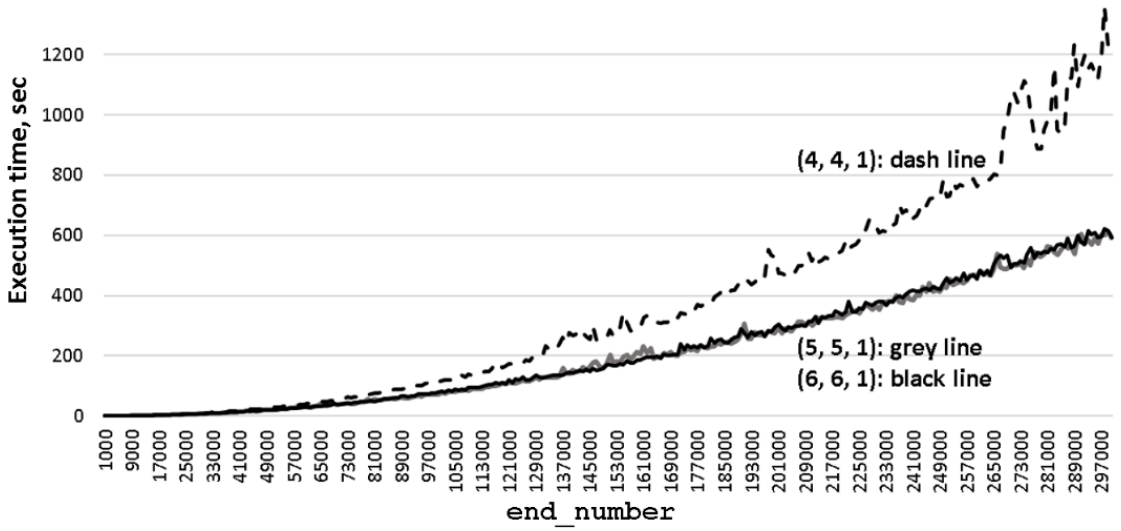

Figure 6: Weight-based load balancing implementation of the prime numbers finding algorithm: Execution times for different values of the variable end_number and performance weights (4, 4, 1), (5, 5, 1), and (6, 6, 1)

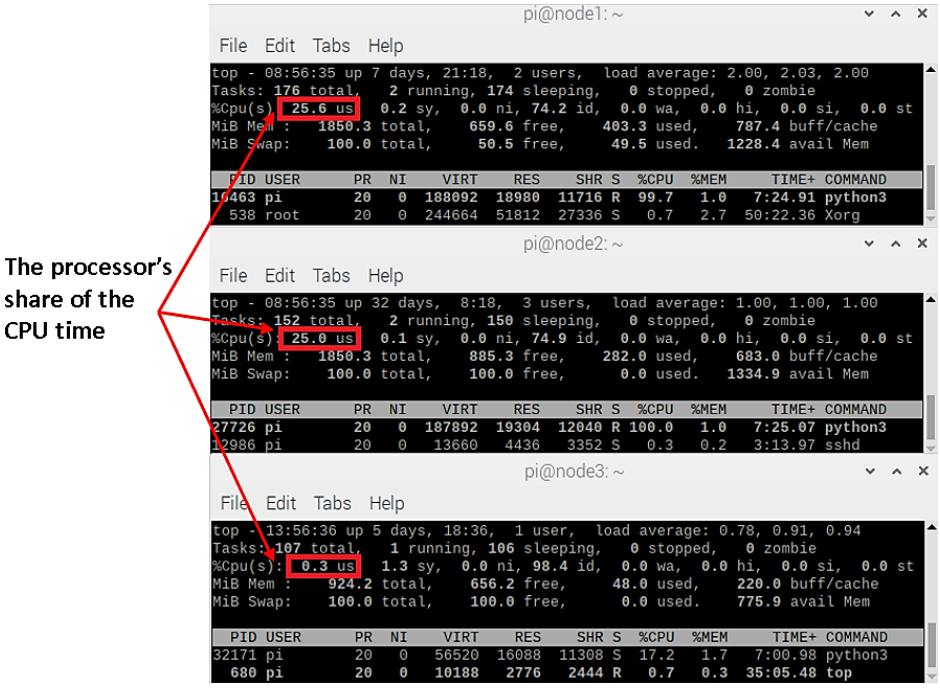

Figure 7: Raspberry Pi command top: An example of the processes currently running on the cluster nodes with weight-based load balancing

3.3. Prime Numbers Finding Algorithm: Comparison of the Basic Equal Load and the Weight-Based Load Balancing Implementations

Analysis of the execution times for equal and weight-based load balancing implementations of the prime numbers finding algorithm, Figure 3 and Figure 6 respectively, shows the following outcomes:

1. The value of speedup for the weight-based load balancing implementation with performance weights (5, 5, 1) and (6, 6, 1) compared with the equal load balancing with performance weights (1, 1, 1) is continuously growing approximately from 1 at the `end_number` 1000 to 5 at the maximum `end_number` that is equal 300000.

2. The weight-based load balancing implementation with performance weights (5, 5, 1) and (6, 6, 1) is slightly faster, around 5 %, than the equal load balancing with performance weights (1, 1, 0) for the `end_number` values greater than 25000 until maximum value 300000. The mean signed deviation for data series with performance weights (5, 5, 1)-(1, 1, 0) is -10.42 sec. Larger execution times for prime numbers less than 25000 represent the classical scalability problem [22] and its effect on the performance in parallel computing: for some cases, the intelligent parallel algorithms work slower compared with the methods without an intelligent component since they need additional time to analyze existing nodes and distribute the workload in accordance with the input dataset(s).

4. Summary and Conclusion

Weight-based load balancing in the Raspberry Pi MPICH heterogeneous cluster was implemented using the fuzzy estimating of performance weights. Fuzzy variables are the type of CPU and OS, RAM, workload, microSD card capacity, power consumption, and OS release date. Four main stages of fuzzification, formation of fuzzy rules, fuzzy inference, and defuzzification are applied to calculate the performance weights of the cluster nodes. In this project, the defuzzification is based on the multiplication operation.

The cluster consists of two Raspberry Pi 4B boards with 2 GB RAM and Raspberry Pi 64-bit OS with desktop and one Raspberry Pi 3B board with 1 GB RAM and Raspberry Pi 32-bit OS Lite. The Python program for the prime numbers finding algorithm is a test example of the developed approach:

1. Recommended weight-based load balancing with performance weights (5, 5, 1) is preferable compared with the next nearest (6, 6, 1) and (4, 4, 1) since three-tuple (5, 5, 1) has less average execution time across the entire set of all observations. The mean signed deviation for data series with performance weights (5, 5, 1)-(6, 6, 1) is -3.01 sec and (5, 5, 1)-(4, 4, 1) is -155.74 sec, i.e., data series with recommended performance weights (5, 5, 1) has less average execution time across the entire set of all observations.

2. The value of speedup for the weight-based load balancing implementation with performance weights (5, 5, 1) and (6, 6, 1) compared with the equal loading on all nodes is continuously growing approximately from 1 at the `end_number` 1000 to 5 at the maximum `end_number` 300000.

Since load balancing is an important task not only for the considered Raspberry Pi MPICH heterogeneous cluster but also for distributed computing systems in general, obtained results may be used in a wide spectrum of parallel programming applications.

The most likely prospect of the presented weight-based load balancing method is the development of a multithread Python application.

5. Acknowledgements

This paper and the research behind it could be much complicated without the support of universities where authors have been conducting the presented project. Authors sincerely appreciate the management and colleagues of the University of Central Asia (Kyrgyzstan) and the Kryvyi Rih National University (Ukraine) for all their patience and kind assistance in the completion of this work.

6. References

[1] High-Performance Portable MPI: MPICH Overview, n.d. URL: https://www.mpich.org/about/overview/.

[2] A. Pajankar, Raspberry Pi Supercomputing and Scientific Programming: MPI4PY, NumPy, and SciPy for Enthusiasts, Apress, New York, NY, 2017. doi: 10.1007/978-1-4842-2878-4.

[3] S.J. Cox, J.T. Cox, R.P. Boardman, S.J. Johnston, M. Scott, N.S. O'Brien, Iridis-pi: A low-cost, compact demonstration cluster, Cluster Computing 17 (2014), pp. 349-358. doi: 10.1007/s10586-013-0282-7.

[4] P.J. Evans, Build a Raspberry Pi Cluster Computer, The MagPi magazine newsletter, 2020. URL: https://magpi.raspberrypi.org/articles/build-a-raspberry-pi-cluster-computer.

[5] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, C. Tseng, Dynamic Load Balancing of Unbalanced Computations Using Message Passing, in: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, IEEE, Long Beach, CA, 2007, pp. 1-8. doi: 10.1109/IPDPS.2007.370581.

[6] S. Kumar, D.H. Rana, Various Dynamic Load-Balancing Algorithms in Cloud Environment: A Survey, International J. of Computer Applications 6(129), 2015, pp. 15-19.

[7] Load Balancing in a Cluster: Oracle ® Fusion Middleware Administering Clusters for Oracle WebLogic Server, 2015. URL: https://docs.oracle.com/middleware/1212/wls/CLUST/load_balancing.htm#CLUST171.

[8] X. Li, Parallel Programming in Python: mpi4py (part 1): PDC Blog, 2019. URL: https://www.kth.se/blogs/pdc/2019/08/parallel-programming-in-python-mpi4py-part-1/.

[9] M. Aldasht, J. Ortega, C.G. Puntonet, Dynamic Load Balancing in Heterogeneous Clusters. PICCIT, 2007. URL: https://www.researchgate.net/publication/237067125_Dynamic_Load_Balancing_in_Heterogeneous_Clusters.

[10] V. Grujoski, V. Talevski, D. Zubov, Microsoft Private Cloud Virtual Machine Logical Processors Settings' Relative Weight Calculation Using Fuzzy Logic. In: Proceedings of Conf. "Computer Intelligent Systems and Networks", Kryvyi Rih National University, Ukraine, 2014, pp. 106-111. doi: 10.13140/RG.2.1.1538.8002.

[11] M.M. Gupta, Soft Computing and Intelligent Systems: Theory and Applications, N.K. Sinha (Ed.), Academic Press, San Diego, CA, USA, 2000. doi: 10.1016/B978-012646490-0/50011-1.
[12] N.L. Prokhorov, Supervisory Computer Control Systems, Finances & Statistics Press Inc., Moscow, 2003.
[13] A. Saepullah, R.S. Wahono, Comparative Analysis of Mamdani, Sugeno And Tsukamoto Method of Fuzzy Inference System for Air Conditioner Energy Saving, J. of Intelligent Systems 2(1), 2015, pp. 143-147.
[14] T. Yulianto, S. Komariyah, N. Ulfaniyah, Application of Fuzzy Inference System by Sugeno Method on Estimating of Salt Production, in: Proceedings of AIP conference 1867, AIP Publishing, Melville, NY, 2017, pp. 020039-1 – 020039-7. doi: 10.1063/1.4994442.
[15] A. Yunan, M. Ali, Study and Implementation of the Fuzzy Mamdani and Sugeno Methods in Decision Making on Selection of Outstanding Students at the South Aceh Polytechnic, J. Inovasi Teknologi dan Rekayasa 2(5), 2020, pp. 152-164.
[16] F. Cavallaro, A Takagi-Sugeno Fuzzy Inference System for Developing a Sustainability Index of Biomass, J. Sustainability 7, 2015, pp. 12359-12371. doi: 10.3390/su70912359.
[17] E. Sonalitha, B. Nurdewanto, S. Ratih, N.R. Sari, A.B. Setiawan, P. Tutuko, Comparative Analysis of Tsukamoto and Mamdani Fuzzy Inference System on Market Matching to Determine the Number of Exports for MSMEs. In: Proceedings of the 9th EECCIS Electrical Power, Electronics, Communications, Controls, and Informatics Seminar, IEEE, Batu, East Java, Indonesia, 2018, pp. 440-445. doi: 10.1109/EECCIS.2018.8692989.
[18] Adriyendi, Fuzzy Logic using Tsukamoto Model and Sugeno Model on Prediction Cost, International J. of Intelligent Systems and Applications 6(10), 2018, pp. 13-21. doi: 10.5815/ijisa.2018.06.02.
[19] D.S.K. Mendis, H.U.W. Ratnayake, A.S. Karunananda, U. Samarathunga, A Statistical Fuzzy Inference System by PCA Based Defuzzification for the Improvement of Sugeno Defuzzification Method, J. of Engineering and Technology of The Open University of Sri Lanka (JET-OUSL) 1(7), 2019, pp. 38-52.
[20] G.E. Blelloch, Programming Parallel Algorithms, Communications of the ACM 39(3), 1996, pp. 85-97. doi: 10.1145/227234.227246.
[21] Raspberry Pi Foundation: VNC (Virtual Network Computing), n.d. URL: https://www.raspberrypi.org/documentation/remote-access/vnc/.
[22] M.A. Fienup, Scalability Study in Parallel Computing, Retrospective Theses and Dissertations, 10900, 1995. URL: https://lib.dr.iastate.edu/rtd/10900.
[23] A. Bate, Thermal Testing Raspberry Pi 4: Raspberry Pi Foundation, 2019. URL: https://www.raspberrypi.org/blog/thermal-testing-raspberry-pi-4/.
[24] Y. Cheng, D. Xu, G. Chen, L. Wang, W. Wu, Performance Analysis of Cluster File System on Linux. In: Proceedings of Computing in High Energy and Nuclear Physics Conference, CERN, Switzerland 2005. URL: https://indico.cern.ch/event/0/contributions/1294347/attachments/602/1146/chengyaodong-id72.pdf. doi: 10.5170/CERN-2005-002.1176.
[25] C. Yu, Scheduling and Resource Management for Complex Systems: From Large-Scale Distributed Systems to Very Large Sensor Networks (Publication No. CFE0002907) [Doctoral dissertation, University of Central Florida], Electronic Theses and Dissertations, 2004-2019, 2010. URL: https://stars.library.ucf.edu/etd/4005.
[26] Geek University: List Processes in Real-time, n.d. URL: https://geek-university.com/raspberry-pi/list-processes-in-real-time/.
[27] Power Consumption Benchmarks: Drupal 9 on a cluster of Raspberry Pis, n.d. URL: https://www.pidramble.com/wiki/benchmarks/power-consumption.