2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

ROS-lite2: Autonomous-driving Software Platform for Clustered Many-core Processor

Yuta Tajima[1], Shuhei Tsunoda[1], and Takuya Azumi[1]

Abstract— **In intelligent robotics, which spans from assistive devices to automation, the development of autonomous systems merges computational and physical capabilities, such as in autonomous wheelchairs. Many-core processor, essential for real-time operations, pose challenges to software adaptability. This paper introduces ROS-lite2, which is a software platform for autonomous vehicles, utilizing a ROS 2 framework based on many-core processor to boost flexibility and simplify deployment. Our method facilitates complex function integration and intuitive operation with less hardware knowledge. Experiments with an autonomous wheelchair demonstrate the platform's effectiveness in improving autonomy and reducing development effort, advancing robotic assistance.**

I. Introduction

In the field of robotics, the integration of deep learning [1] and computer vision [2] is pivotal for enhancing functionalities across various applications, including autonomous vehicles[3], healthcare, and industrial automation. This approach, promoting autonomy and efficiency, often depends on power-intensive CPU and GPU combinations, thus escalating computational demands and energy consumption, which encompasses extensive cooling requirements.

To meet rising energy needs, attention is on high-performance systems especially many-core processor [4], which cater to both processing capacity and energy efficiency. These processor, with multiple units at lower frequencies, offer strong performance and low energy use, supporting different applications per core for consistent execution. For example, Kalray MPPA3-80 Coolidge with 80 cores in five clusters, shows predictable performance. While most autonomous vehicles rely on platforms with integrated GPUs, Coolidge can perform deep learning without one [5].

Leveraging the many-core processor for autonomous driving, using embedded many-core platforms is gaining focus. Challenges include optimizing data transfers between memory banks, software reuse hurdles due to specific coding needs, and complex programming with the Robot Operating System (ROS). These complicate development, requiring advanced technical knowledge for effective implementation in the growing field of autonomous driving technology.

To address autonomous driving system challenges, ROS-lite [6] was created to improve communication on many-core platforms, enabling efficient multi-application operation. Built on a real-time OS, ROS-lite advances beyond traditional ROS in inter-component communication. Yet, its protocol struggles with complex embedded systems' flexibility. Current real-time tools, suited for single-core setups, fail in many-core environments, affecting timing and predictability [7], highlighting the need for better communication in advanced autonomous driving technologies.

This paper proposes ROS-lite2, which is an autonomous driving software platform tailored for many-core processor using ROS 2 [8], which boosts real-time capabilities via the Data Distribution Service (DDS) and efficient Publish/Subscribe communication. It includes bridge functions for better external connectivity and enhances processor performance with existing tools. The platform supports software reusability, enabling seamless integration with ROS 2 applications, simplifying management. Autoware Universe, chosen as the main application, showcases the platform's success on the Coolidge processor.

This paper contributes to the field by proposing a ROS 2-based communication middleware that facilitates inter-cluster node communication on many-core platforms. It introduces a capability for interaction between the proposed platform and the external environment, enhancing compatibility with existing ROS systems and flexibility in use cases. The implementation of this autonomous driving platform and its validation with Autoware Universe illustrate its practicality. Additionally, the paper details the parallelization and assessment of localization functions in autonomous driving applications, showcasing the scalability of many-core platforms.

The structure of this paper is as follows. Section II presents the system model of this study and prerequisite knowledge. Section III discusses the development flow, design, and implementation, and Section IV introduces a case study by using the proposed platform. Section V presents the evaluation results. Section VI compares related work, and Section VII provides the conclusions of this study.

II. System Model

This section presents the system model, as shown in Fig. 1. The proposed platform is the DDS part, and ROS 2 is used without modification for the other parts. This paper has selected Coolidge, a COTS component, as the target for the many-core processors because of the support provided by the hardware vendor and its availability on the market. First, the hardware model is described, followed by a detailed description of the software model.

A. Hardware Model

As shown in Fig. 2, Coolidge consists of five clusters (CC); each CC has 16 processing cores (PE cores) and a DMA connected to a 4 MB local memory called SMEM, which consists of 16 banks. In addition, two 100 Gb Ethernet and USB ports are available as external interfaces.

[1]Graduate School of Science and Engineering,Saitama University

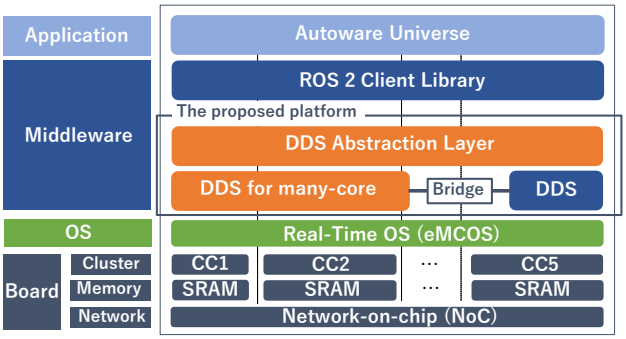

Fig. 1. Scope of the Proposed Platform.

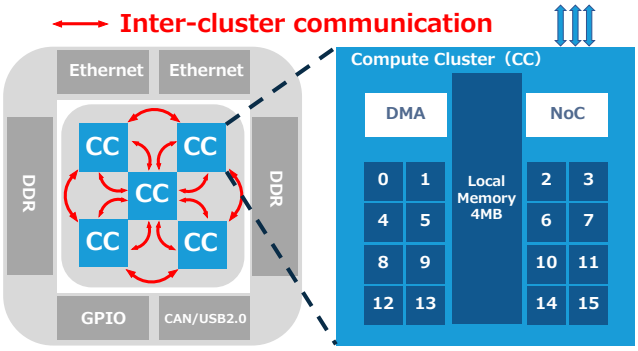

Fig. 2. Overview of the Coolidge Architecture.

The network depicted by the red line in Fig. 2 aims to be more contention-friendly than bus-connected memory access, enhancing system adaptability and scalability in the number of cores. Despite this, the system in this paper employs bus-connected memory access. Additionally, it includes two 100 Gb Ethernet ports and a USB port for external interfaces.

B. Operating System

This paper discusses eMCOS, a real-time OS, selected for its vendor support and Coolidge compatibility, addressing the parallelism and cache coherence issues in many-core platforms. eMCOS offers minimal programming interfaces and libraries for many-core support. Its distributed microkernel architecture allows each core to handle basic functions, including inter-core communication, scheduling, thread, and interrupt management. eMCOS messages facilitate thread communication across cores, offering synchronous/asynchronous modes and priority options.

C. Inter-Cluster Communication

The communication between clusters uses a proprietary method for running ROS 2 applications on the proposed platform. This method uses Emcos' proprietary message passing and distributed shared memory (DSM). Specifically, message passing is used to convey timing, and data content is shared by the DSM for communication. Therefore, FastRTPS and CycloneDDS, which are supported by ROS 2, cannot be used for inter-cluster communication in Coolidge.

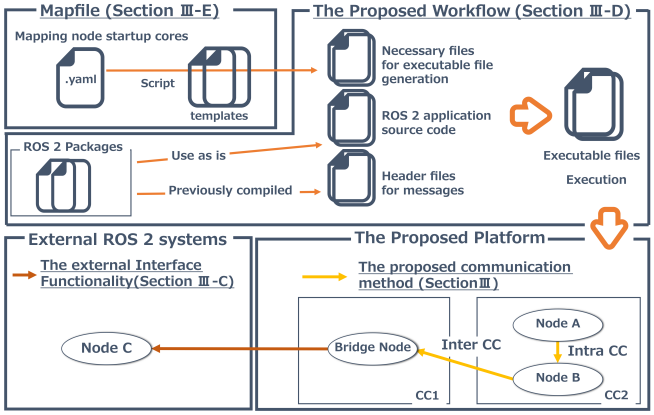

Fig. 3. Detailed Structure of the Proposed Framework.

D. ROS 2

ROS 2, a component-based middleware framework in robotics research, is a redesign of ROS [9] focusing on real-time and functional safety. Adopting Data Distribution Service (DDS) as the communication protocol, ROS 2 enables flexible communication for embedded real-time systems through the Publish/Subscribe model, abstracting software into nodes and topics. Nodes, standard programs in C, Python, and C++, exchange messages via topics: one node publishes, while others subscribe to access these messages.

E. Data Distribution Service

DDS, a middleware specification for the Publish/Subscribe model by Object Management Group (OMG) [10], is utilized in mission-critical sectors such as aviation, medical, and power, supporting a broad array of applications from embedded to infrastructure systems. The Real Time Publish Subscribe (RTPS) protocol, serving as the communication medium, enables low latency and fault-tolerant data transfer. Furthermore, distributed embedded real-time systems are optimized for efficient data transfer across diverse platforms.

F. Autoware Universe

Autoware Universe [11] is open-source software for autonomous vehicles based on ROS 2. It is a revision of the previous Autoware AI in terms of functional safety.

III. Design for Multi Cluster Communication

In this section, the design of the communication mechanism is mentioned first. Next, the proposed platform is explained. Finally, the design of the Publish/Subscribe communication in the proposed platform is described.

A. Design of Communication Mechanism

In developing the communication framework, we evaluated three approaches. The first approach (A) entails porting ROS 2 directly, utilizing Coolidge's internal communication instead of DDS, and reserving DDS solely for external communication. The second approach (B) involves porting both ROS 2 and DDS directly and operating them as they are. The third approach (C) proposes crafting a framework

with an API mirroring ROS 2, exclusively employing DDS for external communication. In this study, we adopted approach A, as it enables multi-cluster utilization and effective leveraging of existing software assets (ROS 2). Approach B was dismissed due to Coolidge's software restricting external communication to a single cluster, while approach C was set aside as replicating a ROS 2 Client Library (RCL) equivalent API posed significant challenges.

B. Design and Implementation of Proposed Platform

The proposed platform makes the development on the many-core platform more efficient by abstracting the communication layer. Note that it is the DDS portion that has been changed on this platform, and that ROS 2 is used without modification for the portion above the ROS 2 Client Library. It supports Publish/Subscribe communication on the many-core platform to provide a development environment where ROS 2 nodes run on each core and communicate with each other. Furthermore, the proposed platform allows existing ROS 2 applications to be used without modification. In this paper, Coolidge was used as the embedded many-core platform, and eMCOS was used as the RTOS for testing. Coolidge was chosen because it is a clustered many-core and offers better real-time performance than non-clustered systems. The overall picture and design relationships of the proposed platform, described in subsequent sections of this section, are shown in Fig. 3.

To realize Publish/Subscribe communication, the communication scheme is designed based on DDS used in ROS 2 [10] [12]. To take advantage of the distributed memory architecture of Coolidge, a clustered many-core processors, shared memory communication using the eMCOS API is employed. eMCOS has a message passing mechanism between threads and can deliver messages in real time, but when the data size becomes large, packet splitting occurs, resulting in delay and unpredictability. However, as the data size increases, packet splitting occurs, resulting in delays and reduced predictability. Shared memory communication avoids packet splitting and reduces the number of times data is copied, enabling fast and stable handling of large data. For this reason, a method was adopted using message passing for timing notifications and shared memory for data communication.

The communication flow when the publisher and multiple subscribers are in different CCs is shown in Fig. 4. Before going into a detailed description of the communication scheme, the following is a brief description of each element that appears in Fig. 4.

Receive thread: For each CC, a thread is created that accepts a publish and notifies waiting subscribers of the topic published. Receive threads handle data transmission through eMCOS messages.

Topic DSM: The topic DSM is realized in a ring buffer; one is generated for each topic. The number of topics in ROS 2 is variable and of variable length, whereas the area allocated for topic DSMs is of fixed length. The use of a ring buffer allows for more efficient use of memory compared to

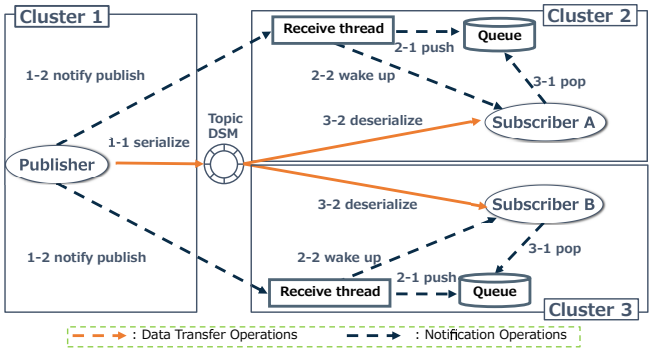

Fig. 4. Communication Flow of the Publish/Subscribe Model in the Proposed Platform.

using a queue. When storing data, the serialized data and a header identifying the serialized data are stored together. The memory address of the storage destination is managed by the distributed shared memory (DSM) technology provided by eMCOS. In this way, even data with a large message size can be accessed from each CC without memory copying. If the sender overwrites an existing message (the oldest) that has not been deserialized by one of the subscribers, the oldest message is deleted.

Queue: For each subscriber, a queue stores notification messages from the publisher, including the Topic DSM's identification and required ring buffer offset for subscription. The queue's maximum capacity is set by the ROS 2 Quality of Service (QoS) setting depth.

Next, the flow of the Publish/Subscribe communication in the proposed platform is described. Each solid line in Fig. 4 shows the access to the message data, which is realized using communication APIs (mcos_dsm_read() mcos_dsm_write())provided by eMCOS. Each dotted line in Fig. 4 describes notifications to subscribers/receiving threads and queue state updates, which are implemented using eMCOS messages.

As shown in Fig.4 (1-*), when a node publishes, serialized topic data are stored in the Topic DSM, and a notification is sent to the CC's receive thread where the subscriber resides. The receive thread receiving the notification forwards the message to the subscriber node queue and wakes the node (Fig.4 (2-*)). Upon wake-up, the subscriber retrieves subscription information from the queue and deserializes the topic data (Fig.4 (3-*)). Considering intercommunication with external ROS 2 systems, the serialization/deserialization method conforms to ROS 2 standards. The publisher deletes data once all subscribers have deserialized the topic.

In addition to complying with the ROS 2 serialization/deserialization specification, the proposed platform supports the following QoS settings provided by ROS 2.

HISTORY: The maximum number of messages to be stored in the queue can be set.

DEADLINE: The maximum time expected before a subsequent message is published on a topic can be set.

LIFESPAN: The deadline for the time between publishing a message and subscribing to the message can be set. Expired messages are deleted.

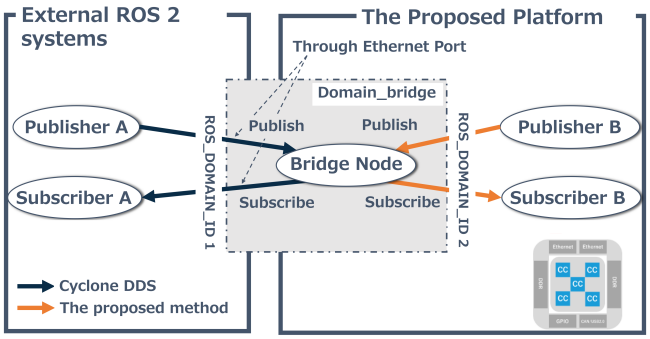

Fig. 5. Overview of External Interface Functionality of Proposed Platform.

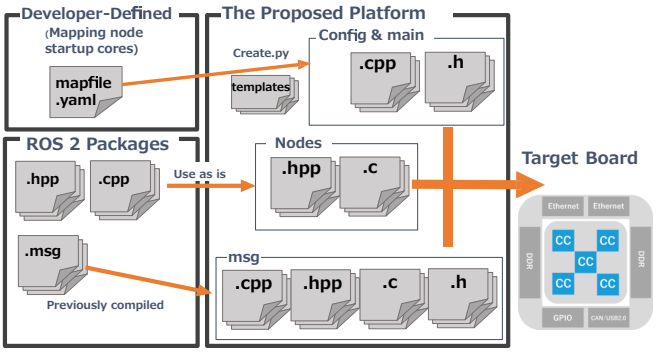

Fig. 6. Development Flow in the Proposed Platform.

```
configuration.topic_dsm_size = 10000;
configuration.topic_name = "/chatter";
configuration.topic_id = 1;
configuration.receiver_cc_no.push_back(2);
configuration.receiver_cc_no.push_back(3);
topic_configurations.push_back(configuration);
```

Fig. 7. Example of a Configuration File. Indicates that Subscribers in CC2 and CC3 Subscribe to the /chatter Topic.

LIVELINESS: The functionality to notify other nodes when a publisher fails and malfunctions.

Therefore, no restrictions are placed on external ROS 2 systems. The implementation of the QoS setting will make interactions with existing ROS 2 systems easier and lower the hurdle for development in many-core platforms.

C. External Interface Functionality

The proposed platform enables interaction with external ROS 2 systems through a designated Ethernet port on Coolidge. This interfacing function is essential as the ROS 2 communication within Coolidge does not natively extend to external systems, necessitating a bridge to facilitate this connection. Utilizing this interface, ROS 2 nodes on the proposed platform can not only interact with external ROS 2 systems but also leverage a range of ROS 2 tools, aiding in visualization and performance analysis. Moreover, this setup offers the flexibility to offload processes from existing systems and to cater to diverse use cases, enhancing scalability. A visual representation of this external communication is depicted in Fig. 5. The bridging between the external ROS 2 systems and the internal communication domain of the many-core platform is enabled by a bridge node. This bridge node acts as a conduit for communication and is the default ROS 2 node activated on our platform. Interaction with external ROS 2 systems is facilitated using the Cyclone DDS communication protocol. When a message is received from an external source, the bridge node routes it to the ROS 2 nodes on the platform using the internal communication method, as illustrated in Fig. 3. Conversely, when transmitting a message externally, the bridge node picks up the publication on the platform and disseminates the topic to the external world following the Cyclone DDS protocol. While Ethernet usage could potentially pose a bandwidth limitation, our primary focus is on optimizing operations within many-core systems, thereby mitigating any such bottleneck concerns.

In the proposed method, different cores handle the processes of subscribing to topics entering the platform and publishing topics to external destinations. Such distribution prevents overloading a single core. By default on Kalray MPPA3-80, both subscribing and publishing processes occur on the same core as the bridge node, which also manages memory and OS tasks, leading to potential overload issues. To address this, our method assigns these tasks to another core, enhancing its capability to handle an increased number of topics efficiently. Using Robot Visualization (Rviz) for debugging amplifies the topic count, but when more cores are free, our method can further distribute tasks for scalability.

Furthermore, the network domains of the proposed platform and the external ROS 2 system are partitioned and connected by a *domain_bridge*. Since ROS 2 employs broadcast communication, many packets besides topic arrive at the Ethernet port of the proposed platform. The *domain_bridge* provided by ROS 2 serves to connect the partitioned ROS 2 domains. By adopting this design, the number of packets reaching the proposed platform can be reduced, avoiding interference and reducing network traffic. Together with the aforementioned core load balancing, the design is more resilient to the increased number of topics handled by the bridge node.

D. Development Flow

This section describes the development flow in the proposed platform. An overview of the application execution process is shown in Fig. 6. Application developers are expected to run ROS 2 applications on many-core platforms without modifying the existing ROS 2 source code. In this way, developments can be performed using the many-core platform with little hardware knowledge. As shown in Fig. 6, the ROS 2 application code can be used as is. The header files associated with the messages are statically generated in advance. An executable file is generated based on those files, files related to the configuration, and files corresponding to the usual main functions. The configuration file contains information about the topic name, the activation core of the node that subscribes to the topic, and the Topic DSM size as shown in Fig. 7.

E. Mapfile

This paper provides a mechanism called mapfile that allows developers to statically provide information on the

```
nodes:
  talker:
      cluster: 1
      core: 1
      topic_in:
          chatter: /std_msgs/msg/string
      topic_to:
          none:
      param_file:
          none:
  listener:
      cluster: 2
      core: 1
      topic_in:
          none:
      topic_to:
          chatter: /std_msgs/msg/string
      param_file:
          none:
launch_path: /path/to/launch_file
```

Fig. 8. Example of a Mapfile Running the Listener Node on Core 1 of Cluster 1 and the Talker Node on Core 1 of Cluster 2.

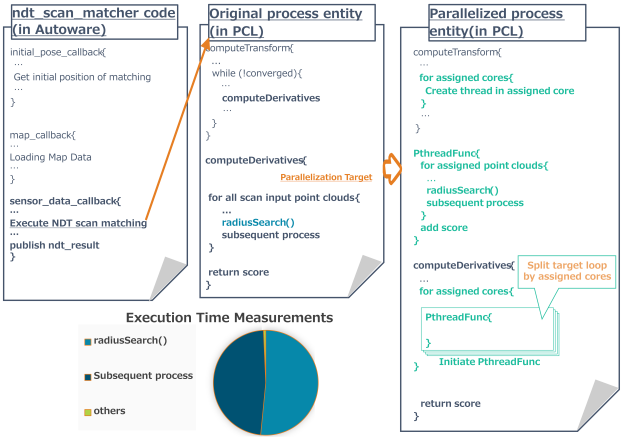

Fig. 9. Overview of Parallelization.

cores and CCs that will execute ROS 2 nodes. Developers can then allocate a node to any core/CC and generate an executable file without having to edit other files. The *mapfile.yaml*, shown in Fig. 8, specifies the name of the node to be executed, information on the core/CC to be activated, and the path to the launch file of the node and runtime parameter files. As elements of *nodes*, the node names to be executed exist. Each of these node names has the elements of *cluster*, *core*, *topic_in*, *topic_to*, and *param_path*. In addition, by specifying the *launch_path*, the topic rename information can be used directly when running on a ROS 2 system.

A *launch_path* is the path to the file describing the launch system, a functionality provided by ROS 2 to run a large set of nodes together. Large-scale systems, such as autonomous driving systems, always have a launch path, enabling a single command to execute many nodes. This paper assumes the node set has a launch path, sparing developers the trouble of knowing and setting the path to each node's source code file, unlike without a *launch_path*. When using bridge functionality to communicate outside the proposed platform, the topic name and type to be received in *topic_in*, and those to be sent outside in *topic_to*, can be simply set.

The script *create.py* is also provided, which automatically generates the files necessary for node assignment and executable file generation based on the information in *mapfile.yaml*. This design reduces the amount of code a developer must write before creating an executable file.

F. Parallelization of the Processing on Proposed Platform

Coolidge has 80 compute cores, each of which can execute processing independently. This design ensures that each application is less prone to be interrupted by other processes, leading to the improved predictability of the execution time. However, if a single core is not sufficient in terms of processing capacity, multiple cores can be used to achieve the required performance by executing processing in parallel. The proposed platform considers and evaluates the parallelization of processing-intensive localization functions in automated driving applications. In this paper, ndt_scan_matcher was selected for parallelization. This node performs location estimation by Normal Distribution Transform (NDT) scan matching. Scan matching is a method that compares point clouds on map data with point clouds acquired by LiDAR or laser scanners to obtain positional agreement and direction of movement.

Before parallelization, time measurements were taken within the NDT scan matching process. ndt_scan_matcher performs scan matching as a callback when sensor data is received, as shown in Fig. 9. Most of the execution time is allocated to radiusSearch() to search for neighboring point clouds, and the subsequent process of determining the overlap of the searched neighboring points with the point clouds on the map data. Measurements of the time required for scan matching as a whole, radiusSearch(), and the subsequent processing are shown in Bottom of Fig. 9. The results show that radiusSearch() and the subsequent process account for 99% of the total scan matching time. In this paper, these parts are distributed and allocated to each core for parallelization.

An overview of the parallelization used in this paper is shown in Fig. 9. The entity of the NDT scan matching process is provided by Point Cloud Library (PCL), an open-source library for handling 3D point clouds. The loop, including radiusSearch(), and the subsequent process (shown as the dotted orange line in Fig. 9), which was found to form a bottleneck in the scan matching process, is described in PCL source code. To distribute this loop across the cores, a thread is first created on each core used for the parallelization. Next, the loop is divided by the number of cores used for the parallelization, and each is assigned to these threads. In this case, the score that determines whether scan matching has converged is the sum of the results of each core.

IV. Case Study

We use Autoware Universe on the proposed platform to enable autonomous driving with an electric wheelchair[1]. To

[1]For this case study, not all processing required for WHILL's autonomous driving is executed on Coolidge. This is currently due to the absence of certain drivers, including those for LiDAR, sensors, and actuators, which are not central to the research's essence; hence, these driver nodes are handled on a laptop.

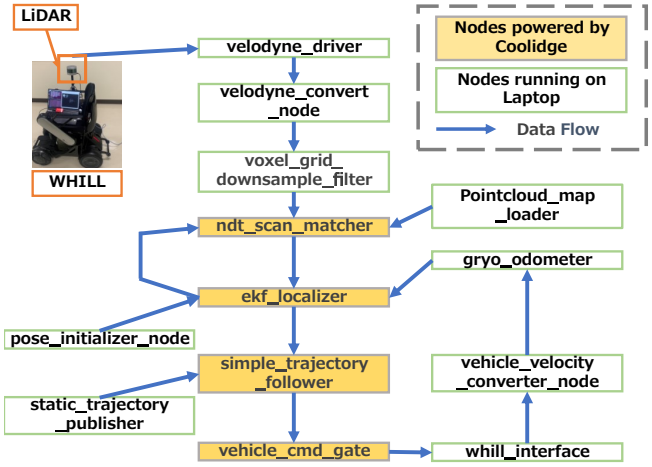

Fig. 10. Node Configuration for WHILL.

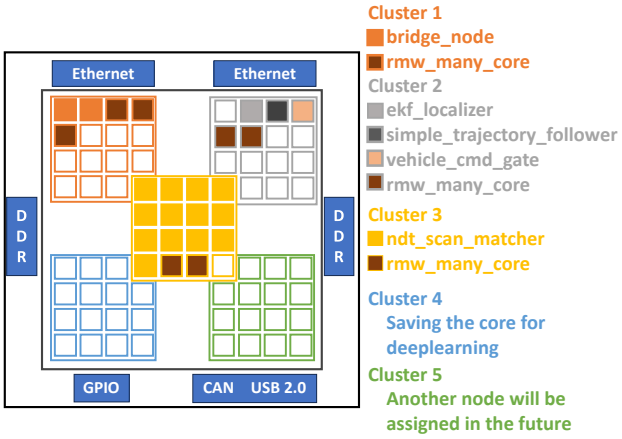

Fig. 11. Core Assignment for WHILL.

TABLE I
NODE DESCRIPTION

Node Name	Description
pointcloud_map_loader	Publishes point cloud data for the map.
ndt_scan_matcher	Estimates the current pose by matching new LiDAR scan data with existing maps.
ekf_localizer	Uses a Kalman filter to improve the accuracy of self-position estimation and synthesizes position and velocity information.
simple_trajectory_follower	Determines velocity and angular velocity to follow the trajectory.
vehicle_cmd_gate	Retrieves information from the emergency handler, planning module, and external controller, and sends messages to the vehicle. Generates the final control command.
static_trajectory_publisher	Delivers predefined trajectories.
pose_initializer_node	Sets the initial position and posture.
whill_interface	Converts Autoware control commands to wheelchair-specific commands.
vehicle_velocity _converter_node	Converts speed information into a general speed data format and publishes it.
gyro_odometer	Determines angular velocity and publishes corrected velocity data along with the velocity data.

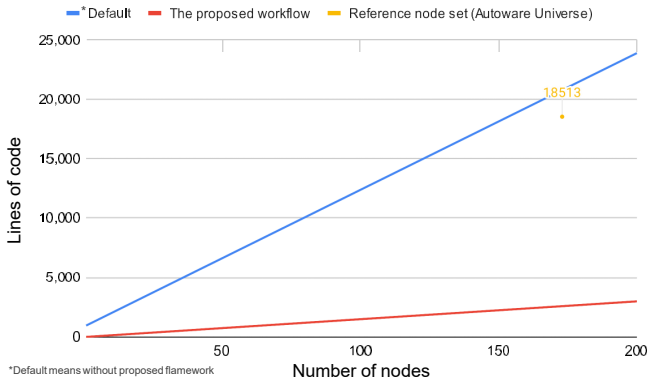

Fig. 12. The Number of Nodes Required to be Run on the Proposed Platform and Amount of Code Generation Required to Generate Executables.

conduct the test, a group of nodes configured based on the Autoware Universe is shown in Fig. 10. Yellow-highlighted nodes in Fig. 10 indicate those offloaded by Coolidge. Nodes offloaded are placed in the core as shown in Fig. 11. First, rmw_manycore is used for inter-cluster communication and occupies two cores in each cluster. The cores are fixed, but it is not important which core it runs on. In Cluster 1, there is bridge_node that is running. This is the node that enables communication between Coolidge and the outside world. In Cluster 2, each offloaded node except ndt_scan_matcher is assigned to a core. Cluster 3 parallelizes ndt_scan_matcher and allocates it to 12 cores, which is the maximum number of cores, excluding rmw_noc and one core for spare. Cluster 4 is not used for Deep Learning. Cluster 5 is used when the number of nodes to offload is increased. Cluster 5 becomes relevant when the number of nodes to offload increases. Finally, Table I shows a brief description of the nodes used.

V. EVALUATION

To assess the scalability of many-core processors, ndt_scan_matcher was parallelized on the proposed platform, revealing the core count's improvement on execution time of the number of cores. Coolidge served as the many-core platform with eMCOS, while experimental laptops had 16 GB memory, GeForce RTX 2060 GPU, Intel Core i7-10875H, and used Linux.

A. Amount of Code Generation Required for Execution on Proposed Platform

On the proposed platform, generating an executable file with the node configuration in Fig. 8 requires 600 lines of code, plus additional code for parameters of each node, increasing the total code. In this paper, we defined the increase in the amount of code generation required when the number of nodes to run increases as follows:

$$\Delta LoC = N * (10 + N_p + (N_t * 6))$$

where N is the number of nodes to be run, N_p and N_t are the number of parameters and topics handled by each node, respectively, and ΔLoC is the increase in the amount of code when the number of nodes to be run increases.

The relationship between the amount of code generation required and the increase in the number of nodes running on the proposed platform is illustrated in Fig. 12. The "Default"

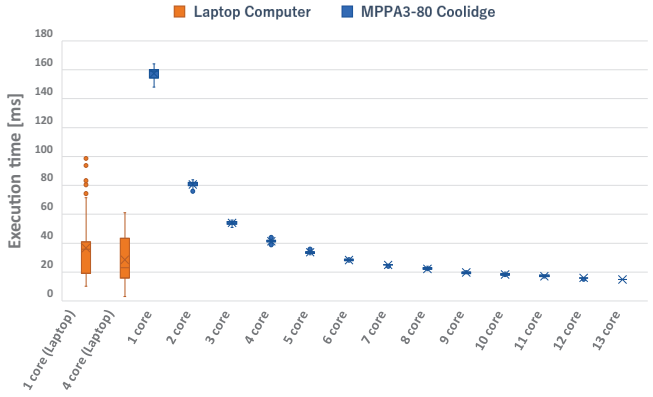

Fig. 13. Measurement of the Execution Time for NDT Scan Matching.

in Fig. 12 indicates that the proposed framework is not used. Here the results are estimated assuming that the average number of parameters and topics of a ROS 2 node are 15 and 15, respectively. Additionally, the results were also calculated from the number of parameters and Topic of the configuration used in the actual Autoware Universe reference node set [11] (shown as the yellow dot in Fig. 12). The results show that the formulation used in this paper is generally accurate. In addition, by running the actual set of reference nodes from Autoware Universe on the proposed platform, it was found that the amount of code the user needs to write can be reduced by 86% using this workflow. From the results of Fig. 12, it can be inferred that the difference widens further when the number of parameters and topics is increased.

B. Experimental Parallelization of Processing on Proposed Platform

Based on the research in Section III-F, the loop in ndt_scan_matcher that performs the radiusSearch() and the subsequent processing for each set of input point clouds was parallelized, and the execution time for the NDT scan matching was measured by varying the number of cores. The same set of input point clouds was also tested on the laptop computer. At this time, we measured the performance with a single core and with the default parallel execution using four cores. The results shown in Fig. 13 show that as the number of parallel cores increases, the execution time decreases. When seven cores were used, the execution time was below the average of the execution time of four parallel processes on a laptop computer. In addition, the variation in execution time is smaller than for the laptop computers for all core counts, indicating a high degree of predictability for the scan matching processing time. The reason for these results is that each core of the many-core processor is able to perform tasks independently. Since a certain process, in this case, the scan matching process for each point cloud, can be occupied by one core, high predictability of execution time can be achieved without interference from other processes. Static assignment of cores to processes can lead to resource wastage, but in this case, not all cores are utilized, resulting in less resource wastage compared to a scenario where no cores are used at all.

C. Lessons Learned

In the context of robotics systems, understanding the interoperability and real-time communication capabilities is essential. To validate the capabilities of the proposed platform, we conducted autonomous operations using WHILL. These tests showcased that our platform seamlessly interfaced with an externally hosted ROS 2 system, achieving real-time performance for the autonomous driving task. Furthermore, the paper introduced a communication method using distributed shared memory on many-core processors and the bridge functionality as an external interface, which was also found to be effective. These components were effectively utilized in practice, reinforcing their potential in real-world CPS applications.

To accommodate more nodes, optimizing core/CC process allocation is crucial. The bridge functionality via domain_bridge helps reduce network traffic, but stable performance requires improved core/CC allocation as message size and frequency grow. The node-initiated core static allocation in our workflow may optimize data receiver placement near senders, depending on the scenario.

In evaluating our proposed workflow, we formulated a model to analyze the relationship between the increase in the number of nodes and the corresponding increase in code generation required before execution on the proposed platform. By comparing the amount of code required to operate the actual Autoware Universe reference system, our analysis validated the formulated model. Notably, the results revealed that our workflow can reduce code generation by 86% when executing the reference system.

The results of the parallelization experiments show that the execution time generally decreases as the number of cores increases. For ndt_scan_matcher, the target of this paper, the use of five cores was found to complete the process faster than the average execution time on the laptop computer. The number of cores used in parallel is expected to vary depending on the requirements of the actual use case. The optimal core allocation should be considered by weighing the execution time reduction benefit of increasing the number of cores to a node against the execution time reduction when cores are allocated to other tasks. The high predictability demonstrated by the execution results on the proposed platform will be useful in the discussion of core allocation optimization.

VI. Related Work

In this section, the studies on many-core platforms and communication middleware, such as ROS, are presented, and the position of this paper is described.

embeddedRTPS [13]: This middleware, leveraging RTPS over FreeRTOS and lightweightIP, aims at embedded platforms, showing ROS adaptability on STM32 and automotive microcontrollers.

ROSCH [14]: This ROS-based framework introduces fixed priority-based DAG scheduling and fail-safe mechanisms for real-time performance, ensuring compatibility with existing software beyond traditional methods.

TABLE II
COMPARISON OF PREVIOUS WORK

	Embedded System	Real-time OS	many-core	QoS	ROS 2
embeddedRTPS [13]	✓			L	
ROSCH [14]		✓			
RT-ROS [15]		✓			
micro-ROS [16]	✓	✓			L
mROS 2 [17]	✓	✓			
SDN-SS [19]	✓		✓		
ROS 2-with-16 cores [18]	✓	✓		✓	✓
ROS-lite [6]	✓	✓	✓		
this paper	✓	✓	✓	L	✓

∗ In a table, "L" means "limited."

RT-ROS [15]: The framework enables real-time ROS tasks on a real-time OS and non-real-time tasks on Linux across separate cores, enhancing performance and real-time support for ROS, as evaluations confirm.

micro-ROS [16]: The middleware supports DDS-XRCE instead of classical DDS to run ROS 2 nodes on microcontrollers. The middleware provides a microcontroller-optimized client API supporting major ROS concepts, multi-RTOS support, and a generic build system.

mROS 2 [17]: mROS 2 offers a streamlined agentless runtime for embedded devices, promoting ROS 2 compatibility. Featuring publish/subscribe mechanisms, RTPS, UDP/IP stacks, and real-time kernel APIs, it allows embedded nodes to communicate directly with ROS 2 nodes via topics.
ROS 2-with-16 cores [18]: This framework bases on ROS 2 for embedded multi-core platforms.The effectiveness of the framework is demonstrated through the parallel execution of autonomous driving applications on a multi-core platform.
SDN-SS [19]: This framework improves network-on-chip communication in many-core systems, providing high flexibility, reduced complexity, efficient DoS and spoofing attack prevention, with low overhead and minimal communication impact, excluding ROS framework integration.

ROS-lite [6]: This framework employs NoC for efficient many-core platform communication, allowing ROS nodes on each core to interact, proving its utility in parallelizing self-driving software with confirmed deadline adherence for low-speed autonomy.

In Table II, the features of related frameworks and the proposed platform are briefly summarized. In the table, the "L" in the QOS field indicates a QOS that is supported by ROS 2 but not by embeddedRTPS. To the best of our knowledge, our study is the first work to adapt the ROS 2 framework to the embedded many-core platform. This paper has provided a ROS 2 node execution environment with multiple CCs 16 cores and supports for QoS functionalities to achieve ROS 2-like Publish/Subscribe communication.

VII. CONCLUSION

This paper proposes a software platform for running automated systems on many-core processors. The platform includes three proposals: a communication scheme for Publish/Subscribe communication using distributed shared memory on many-core processors, a bridge function to interface the proposed platform with the external environment, and a workflow that allows developers to execute software without deep hardware knowledge. The proposed platform based on ROS 2 will not only provide flexible use cases and easy integration with existing ROS 2 systems, but also enable the use of useful tools such as visualization tools. In addition, workflows that minimize the expertise required will be helpful when tackling larger and more complex use cases. Furthermore, our discussion and verification of parallelization on the proposed platform showed that the proposed platform has high predictability, as well as demonstrating scalability of the many-core processors.

ACKNOWLEDGMENT

This work was supported by JST PRESTO Grant Number JPMJPR21P1.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.
[3] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on Board: Enabling autonomous vehicles with embedded systems," in *Proceedings of ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018.
[4] P. Burgio, M. Bertogna, N. Capodieci, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, vol. 52, pp. 299–311, 2017.
[5] T. Yabe and T. Azumi, "Exploring the performance of deep neural networks on embedded many-core processors," in *Proc. of IEEE ICCPS*, pp. 193–202, 2022.
[6] T. Azumi, Y. Maruyama, and S. Kato, "ROS-lite: ROS framework for NoC-based embedded many-core platform," in *Proc. of IEEE/RSJ IROS*, 2020.
[7] Y. Maruyama, S. Kato, and T. Azumi, "Exploring scalable data allocation and parallel computing on NoC-based embedded many-cores," in *Proc. of IEEE ICCD*, 2017.
[8] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS 2," in *Proc. of ACM EMSOFT*, 2016.
[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *Proc. of IEEE ICRA*, 2009.
[10] Object Management Group (OMG), "DDSI-RTPS." https://www.omg.org/spec/DDSI-RTPS/.
[11] T. Kuboichi, A. Hasegawa, B. Peng, K. Miura, K. Funaoka, S. Kato, and T. Azumi, "CARET: Chain-Aware ROS 2 Evaluation Tool," in *Proc. of IEEE EUC*, 2022.
[12] Eclipse, "Cyclone DDS." `https://projects.eclipse.org/projects/iot.cyclonedds`.
[13] A. Kampmann, A. Wüstenberg, B. Alrifaee, and S. Kowalewski, "A portable implementation of the real-time publish-subscribe protocol for microcontrollers in distributed robotic applications," in *Proc. of IEEE ITSC*, 2019.
[14] Y. Saito, F. Sato, T. Azumi, S. Kato, and N. Nishio, "ROSCH: Real-time scheduling framework for ROS," in *Proc. of IEEE RTCSA*, 2018.
[15] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "RT-ROS: A real-time ROS architecture on multi-core processors," *Future Generation Computer Systems*, vol. 56, pp. 171–178, 2016.
[16] micro ROS. `https://micro-ros.github.io/`, 2023.
[17] mROS 2. `https://github.com/mROS-base/mros2`, 2022.
[18] S. Tsunoda and T. Azumi, "ROS 2 framework for embedded multi-core platform," in *Proc. of APRIS*, 2021.
[19] M. Ruaro, L. L. Caimi, and F. G. Moraes, "A systemic and secure SDN framework for NoC-based many-cores," *IEEE Access*, vol. 8, pp. 105997–106008, 2020.