Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

# Mapping Method Usable with Clustered Many-core Platforms for Simulink Model

Yutaro Kobayashi[1,a) ] Kentaro Honda[1,b)] Sasuga Kojima[2] Hiroshi Fujimoto[3]
Masato Edahiro[4] Takuya Azumi[1]

Received: May 24, 2021, Accepted: November 2, 2021

**Abstract:** Multi/many-core processors are being increasingly used to reduce power consumption and improve performance. The use of model-based development for embedded systems has also been increasing. Relative to these trends, the model-based parallelizer or MBP has an essential role in parallelizing applications at the model level. MBP maps Simulink blocks to cores using various types of information such as block characteristics, C code, and multi/many-core hardware implementation. However, MBP does not consider many-core hardware with cluster structures such as Kalray MPPA2-256 processor which contains 16 clusters of 16 cores for 256 general-purpose cores in total. This paper proposes an algorithm that determines core allocations by considering cluster structures. The proposed algorithm combines two other algorithms: one algorithm uses the core allocation of MBP and path analysis at the cluster-level and considers effects from communication contention when determining cluster allocations, and the other algorithm uses the results from MBP and remaps cluster allocations. The proposed algorithm produces better results than its component algorithms could produce separately. Evaluations demonstrate that the proposed algorithm obtained the best results among four methods in terms of execution time on Simulink models.

**Keywords:** embedded systems, model-based development, multi/many-core

## 1. Introduction

Recently, embedded control systems, such as those used in the automotive domain have become increasingly complex and process large amounts of data. The use of multi/many-core processors has consequently increased because these are effective in improving performance. In addition, development phases of embedded systems have become more efficient and easier with Model-Based Development (MBD) [7]. The use of models improves productivity, and the number of hardware prototypes can be reduced because control specifications can be simulated and verified at an early development stage. Furthermore, we can automatically generate code by using MBD.

MATLAB/Simulink [14], which is a graphical programming environment for MBD, has an add-on called Embedded Coder [13] that can automatically generate readable compact and fast C code for embedded processors. However, the Embedded Coder cannot generate a parallel code for multi/many-core processors. Therefore to exploit the inherent multi/many-core processor performance, parallelizing tools for MBD applications are important, e.g., Model-Based Parallelizer (MBP) [6], [8].

MBP generates parallelized C code using Simulink models, a sequential code generated by the Embedded Coder, and the Software-Hardware Interface for Multi-Many-core (SHIM) [10] XML, which has multi/many-core hardware implementations. MBP can allocate blocks to any number of cores, but cannot allocate blocks to many-core hardware with cluster structures such as MPPA2-256 developed by Kalray [1], [3], [12].

MPPA2-256 contains 16 clusters of 16 cores for 256 general-purpose cores. The clusters of cores can run independent applications separately to achieve the desired power envelope for embedded applications. MPPA2-256 connects distributed memory devices with network-on-chip (NoC). Since delays in communications vary depending on the communication objects (between cores or between clusters), the allocations of blocks to cores are very important. The challenge of core placement changes depending on whether processors have a cluster structure. For example, communication time varies depending on whether the clusters of cores that communicate are the same or not. However, MBP cannot handle hardware with cluster structures, such as MPPA2-256.

This paper proposes an algorithm that determines core allocations to MPPA2-256. A hybrid algorithm is an algorithm that combines two or more algorithms (Mapping Algorithm using Path Analysis (MAPA) and Mapping Algorithm from Core to Cluster (MACC)) to create a better algorithm. If users require N clusters, MAPA executes N core allocations of MBP, considers the allocations as clusters, and executes the path analysis at

1 Graduate School of Science and Engineering, Saitama University, Saitama 338–8570, Japan
2 Graduate School of Engineering Science, Osaka University, Suita, Osaka 565–0871, Japan
3 Technology Headquarters, eSOL Co., Ltd., Nakano, Tokyo 164–8721, Japan
4 Graduate School of Informatics, Nagoya University, Nagoya, Aichi 464–8601, Japan
a) y.kobayashi.858@ms.saitama-u.ac.jp
b) k.honda547@ms.saitama-u.ac.jp

the cluster-level. Then, MAPA determines cluster allocations by considering the communication contention of NoC and executes 16 core allocations to each cluster. MACC executes 16*N core allocations of MBP to create XML at core granularity. Next, it uses the core granularity XML to perform the allocation of the N clusters. Finally, the finer details of the allocation are modified and the allocation is determined.

**Contributions:** The main contributions of the proposed algorithm are as follows:

- The proposed algorithm provides a technical contribution to MATLAB/Simulink. This algorithm parallelizes and allocates tasks onto cores from distinct clusters.
- The proposed algorithm determines core allocations of Simulink models for the cluster-based many-core. As a result, the algorithm decreases the execution time more than the existing method does.
- The proposed algorithm reduces the burden on users who run C code generated by MBP on the cluster-based many-core.

**Organization:** The remainder of this paper is organized as follows. Section 2 describes the MBP system model and Kalray MPPA2-256. Section 3 discusses problems applying the MBP to the MPPA2-256 processor and explains how the proposed approach can mitigate such problems. Section 4 evaluates the proposed algorithms. Section 5 discusses related work. Section 6 presents the conclusions and suggestions for future work.

## 2. System Model

A system model of Model-Based Parallelizer (MBP) is shown in **Fig. 1**. MBP supported by Embedded Multicore Consortium [4] generates parallelized C code from a Simulink model, C code generated by the Embedded Coder, and SHIM [10]. SHIM has a multi/many-core hardware implementation. In the following sections, we describe SHIM, MBP, and MPPA2-256.

### 2.1 Software-Hardware Interface for Multi-Many-core (SHIM)

The SHIM specifications define an architecture description standard to provide a common interface that extracts the hardware properties. SHIM XML has hardware information such as the number and types of cores, memory maps. Software verification can be performed without the actual hardware by expressing it using SHIM. Therefore, the cost and time required to support new multi/many-core processors can be reduced.

### 2.2 Model-Based Parallelizer

MBP can generate a parallelized C code from a Simulink model as shown in Fig. 1. The MBP process consists of three major phases. The first phase is the addition of information for which various types of information, such as the C code and SHIM XML are combined at the block level into a BLXML obtained from a Simulink model. The second phase is core allocation, which maps blocks to cores using the cycle-count and code annotated BLXML to generate a parallelized BLXML. The third phase is the generation of parallelized C code.

#### 2.2.1 Adding Information Phase

MBP generates Simulink block information as a BLXML using the mdltoxml converter. In addition, MBP receives the C code generated by the Embedded Coder and separates the C code for each block. MBP combines the separated C code into the BLXML and generates a code-annotated BLXML (blxml2code). Furthermore, MBP receives a multi/many-core hardware implementation from SHIM XML and calculates the performance of each block. MBP combines the performance information into the BLXML and then generates the code and cycle-count as annotated BLXML.

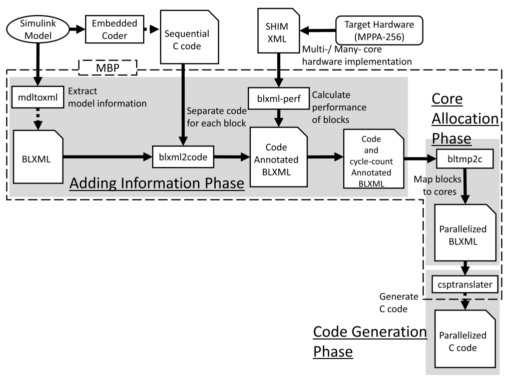

**Fig. 1** System model of MBP.

#### 2.2.2 Core allocation phase

According to the cycle-count and code annotated BLXML, MBP maps blocks to cores and then generates the mapping information as a parallelized BLXML. Note that three conditions must be satisfied for appropriate core allocations. First, the load balances of cores should be equal to each other as much as possible. Secondly, the communication overhead should be as small as possible. Finally, the cycle should be protected. To satisfy these conditions, core allocations can be considered combinatorial problems. MBP implements a double hierarchical clustering method to satisfy these conditions [8].

#### 2.2.3 Code Generation Phase

MBP reconfigures the C code for each block and generates a parallelized C code. In the parallelized C code, MBP inserts the communication APIs of *mbp_channel_receive* on the receiving side and *mbp_channel_send* on the sending side where communications occur between cores. According to the parallelized BLXML, which includes core allocations, MBP redesigns the C code according to the parallelized BLXML which includes core allocations.

### 2.3 Karlay MPPA2-256

The MPPA2-256 processor is based on an array of Compute Clusters (CCs) and I/O subsystems (IOSs) that are connected to NoC routers with a toroidal two-dimensional topology as shown in **Figs. 2** and **3**. In this following sections, we describe CCs, NoC, and eMCOS which are typical operating systems (OS) for many cores.

#### 2.3.1 Compute Clusters (CCs)

In MPPA2-256, the 16 inner routers of the NoC correspond to the CCs, which are composed of a Processing Engine (PE), Resource Manager (RM), SMEM, NoC Interface, and Debug Support Unit (DSU). Cluster local memory (SMEM) is shared with 16 Processing Engines (PEs) and a Resource Manager (RM), so

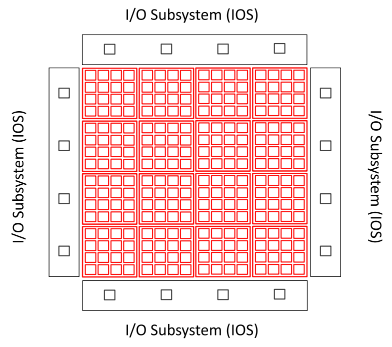

**Fig. 2** An overview of the architecture of Kalray MPPA2-256.

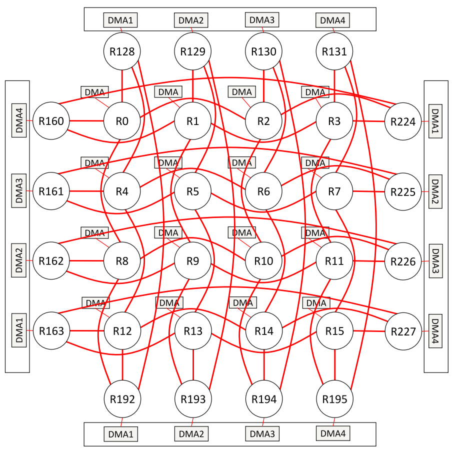

**Fig. 3** NoC connections of MPPA2-256.

that 17 k1-cores (a PE or the RM) share 2 MB SMEM. Developers spawn computing threads on Processing Engines (PEs), of which 16 PEs and a resource manager (RM), each has the Kalray-1 cores which implement a 32-bit 5-issue Very Long Instruction Word architecture with a frequency of 600 or 800 MHz. In addition, each core has its own instruction and data cache, which is 2-way associative with a capacity of 8 KB. The NoC interface has a DMA engine and a NoC router. The DMA engine that is instantiated in every cluster and connected to the SMEM has the following three NoC interfaces. A receive (Rx) interface is installed on the receiving side to receive data from the DMA. A transmit (Tx) interface and a microcore (UC) interface manage the sending side that programs use to send data between clusters.

#### 2.3.2 Network-on-Chip (NoC)

The 16 CCs and the 4 IOSs are connected to the NoC as shown in Fig. 3. MPPA2-256 adopts a torus topology [2] as a bus network that connects routers (CCs and IOSs). One router per CC and four routers per I/O subsystem hold their own routers. The routers R0–15, R128–131, R160–163, R224–227, and R192–195 shown in Fig. 3 belong to the NoC routers shown in Fig. 2. Each network router (a CC or an IOS) includes the 5-link NoC routers that have FIFOs queueing flits, each network router has four duplexed links for neighbors in the north, south, east, west directions and a duplexed link for the local address space attached to the NoC router. The data links are 4 bytes wide in each direction.

#### 2.3.3 eMCOS

eMCOS developed by eSOL is a real-time operating system supporting MPPA2-256. In the scheduling of the eMCOS, threads for an arbitrary number of processor cores are extracted from the core with the highest priority, and exclusively executed for each designated processor core. Since the time taken for execution can be calculated without movement or interruption of the processor core, it is possible to guarantee the real-time property. To provide communication, eMCOS provides an API using NoC. eMCOS message is a message API between threads via Tx interfaces. In the case of eMCOS messages, data can be exchanged between threads regardless of the cluster on which the thread is running.

## 3. Approach

In this section, we will describe an approach that determines core allocations by considering cluster structures. First, we will discuss a problem when using MBP for MPPA2-256. Then, we will propose two algorithms (Mapping Algorithm using Path Analysis (MAPA) and Mapping Algorithm from Core to Cluster (MACC)). MAPA determines core allocations in each cluster after determining the cluster allocation. MACC determines on cluster allocation based on the results of MBP at core granularity. Afterward, we will propose a hybrid algorithm which combines MAPA and MACC to compensate for the disadvantages of each.

### 3.1 Problem

MPPA2-256 has complex cluster structures as mentioned in Section 2.2.2. One of the features of the hardware with cluster structures (e.g., MPPA2-256) is the NoC, as described in Section 2.3.2. In addition, we use eMCOS message described in Section 2.3.3. The eMCOS message varies significantly during communication delays depending on the communications between clusters and cores. MBP can determine core allocations for any number of cores, but cannot determine core allocations considering cluster structures. Therefore, if we will use the original results of MBP, the overhead caused by communication delays becomes large. In this section, we propose algorithms for deciding on core allocations that consider the cluster structures.

### 3.2 Mapping Algorithm Using Path Analysis (MAPA)

Mapping Algorithm using Path Analysis (MAPA) is executed with a flow that determines cluster allocations and then core allocations in each cluster. **Figure 4** shows the MAPA system model. Firstly, if users use N clusters, then MAPA receives a Simulink model and SHIM XML for MPPA2-256 as input and executes the N core allocation using MBP. MAPA considers the result of MBP as a cluster and determines cluster allocations to avoid communication delays as much as possible. Cluster Allocation Method receives the parallelized BLXML and the parallelized C code generated by MBP. The method determines cluster allocations by considering cluster structures and then outputs the code and cycle-count annotated BLXML divided for each cluster, the method generates N code and cycle-count annotated BLXMLs. Then, MAPA executes 16 core allocations of MBP for each code and cycle-count annotated BLXML to determine core allocations in each cluster and generates N parallelized BLXMLs. MAPA integrates N parallelized BLXMLs and creates one parallelized BLXML that has the information on the 16*N core allocations. The code generation phase of MBP outputs a parallelized C code

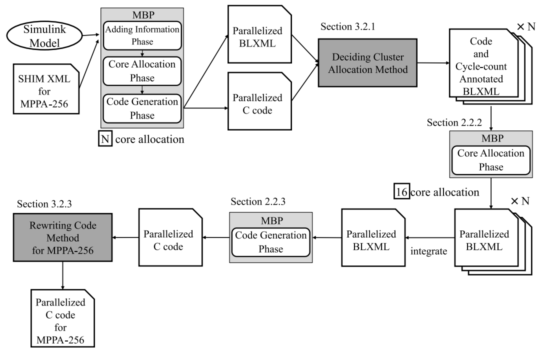

**Fig. 4** System model of Mapping Algorithm using Path Analysis (MAPA).

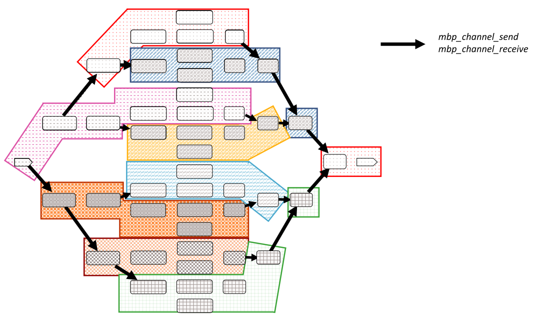

**Fig. 5** An example of converting the information to the cluster level.

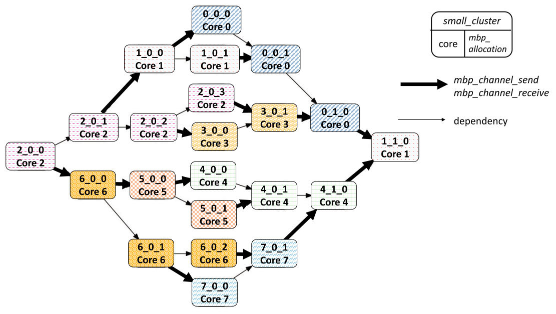

**Fig. 6** A result of the conversion of the information to the cluster-level.

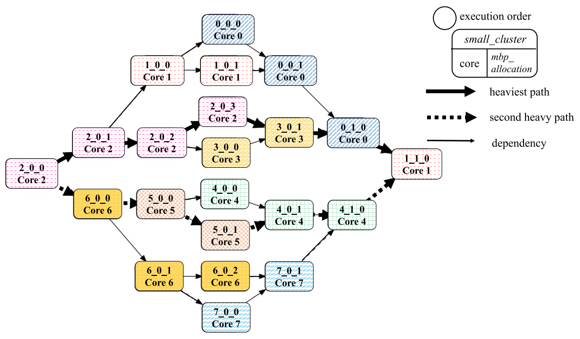

**Fig. 7** An example of confirming an order for determining cluster allocations.

from the parallelized BLXML. Finally, since the original parallelized C code generated by MBP cannot run on MPPA2-256, the code rewriting method for MPPA2-256.

### 3.2.1 Cluster Allocation Method

Cluster Allocation Method receives the parallelized BLXML and the parallelized C code as input, determines cluster allocations by considering the cluster structures, and outputs N code and cycle-count annotated BLXMLs for each cluster. The method flow consists of the following five steps.

**Convert the information to the cluster-level**: In this phase, the method converts the information on the core allocations generated by MBP to the cluster-level. In order to maintain the dependency, the method uses a parallelized C code and divides the mass of blocks. **Figure 5** shows the result of eight core allocations by MBP to execute an example model that has 47 blocks. The different colors represent the different cores. Here, the method regards the core allocations (*mbp_allocation*) divided by MBP as clusters and decides how to allocate these to 16 CCs. The names of the blocks and core allocations (*mbp_allocation*) determined by MBP are described in the boxes. The arrows indicate the dependencies between the blocks and each thread in the parallelized BLXML is boxed with a light color. In addition, *mbp_channel_send* and *mbp_channel_receive* are indicated by the thick arrows. The method divides the mass of blocks when the thread is changed or when *mbp_channel_send* and *mbp_channel_receive* appear in the parallelized C code and then outputs the information at the cluster-level. **Figure 6** shows the result when we executed the method on the example model. We named each node as *small_cluster* (e.g., 0_0_0, 1_0_0, and 2_0_0). Note that the name of *small_cluster* (e.g., 1_0_0) is configured by *mbp_allocation* (e.g., 1) and thread number (e.g., 0_0) as shown in Fig. 6.

**Adopt a path analysis**: This method applies a path analysis and a critical path method to the information of the cluster-level. The weight of each node (*small_cluster*) is the sum of the throughputs of the containing blocks. The method calculates the earliest start time (ES) and the latest start time (LS) of each node and derives the weight of all paths [8].

**Confirm an order for determining cluster allocations**: According to the results of the path analysis, the method confirms an order for determining cluster allocations. Since we must first deal with the path with heavier processing, the method sequentially allocates *small_cluster* on a heavier path to CC. For example, if the heaviest path is the thick arrow path and the second heavy path is a thick dotted arrow path in **Fig. 7**, the order is the number at the upper left of *small_cluster*. This process is repeated in order from heavier paths and is terminated when the order of all *small_cluster* is determined.

**Confirm cluster allocations by the number of hops**: When MPPA2-256 performs NoC communications, communication contention tends to occur as the number of hops increases. Since the execution of processing consumes more time when communication contention occurs, we must prevent the contention on a heavy processing path as much as possible. Therefore, executing blocks on a path with heavy processing by a small number of hops is necessary. To realize this, the core allocations of MBP (*mbp_allocation*) to the CC of MPPA2-256 are implemented using the determined order. The flow is described in Algorithm 1. *N* is the number of clusters to be implemented by the users. *sc* and *ma* mean *small_cluster* and *mbp_allocation*, respectively. *C* has 16 CCs. *ORDER* contains *sc* as the order determined above. E[$ma_i$] is an array that manages whether the cluster allocation has been determined or not. Note that **Table 1** summarizes the number of hops between CCs when we used eMCOS message.

**Table 1** The number of hops between CCs

| to / from | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 | CC11 | CC12 | CC13 | CC14 | CC15 | CC16 | Sum |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CC1 |  | 2 | 1 | 3 | 2 | 4 | 3 | 5 | 1 | 3 | 2 | 4 | 3 | 5 | 4 | 6 | 48 |
| CC2 | 2 |  | 3 | 1 | 4 | 2 | 5 | 3 | 3 | 1 | 4 | 2 | 5 | 3 | 6 | 4 | 48 |
| CC3 | 1 | 3 |  | 2 | 3 | 5 | 2 | 4 | 2 | 4 | 1 | 3 | 4 | 6 | 3 | 5 | 48 |
| CC4 | 3 | 1 | 2 |  | 5 | 3 | 4 | 2 | 4 | 2 | 3 | 1 | 6 | 4 | 5 | 3 | 48 |
| CC5 | 2 | 4 | 3 | 5 |  | 2 | 1 | 3 | 3 | 5 | 4 | 6 | 1 | 3 | 2 | 4 | 48 |
| CC6 | 4 | 2 | 5 | 3 | 2 |  | 3 | 1 | 5 | 3 | 6 | 4 | 3 | 1 | 4 | 2 | 48 |
| CC7 | 3 | 5 | 2 | 4 | 1 | 3 |  | 2 | 4 | 6 | 3 | 5 | 2 | 4 | 1 | 3 | 48 |
| CC8 | 5 | 3 | 4 | 2 | 3 | 1 | 2 |  | 6 | 4 | 5 | 3 | 4 | 2 | 3 | 1 | 48 |
| CC9 | 1 | 3 | 2 | 4 | 3 | 5 | 4 | 6 |  | 2 | 1 | 3 | 2 | 4 | 3 | 5 | 48 |
| CC10 | 3 | 1 | 4 | 2 | 5 | 3 | 6 | 4 | 2 |  | 3 | 1 | 4 | 2 | 5 | 3 | 48 |
| CC11 | 2 | 4 | 1 | 3 | 4 | 6 | 3 | 5 | 1 | 3 |  | 2 | 3 | 5 | 2 | 4 | 48 |
| CC12 | 4 | 2 | 3 | 1 | 6 | 4 | 5 | 3 | 3 | 1 | 2 |  | 5 | 3 | 4 | 2 | 48 |
| CC13 | 3 | 5 | 4 | 6 | 1 | 3 | 2 | 4 | 2 | 4 | 3 | 5 |  | 2 | 1 | 3 | 48 |
| CC14 | 5 | 3 | 6 | 4 | 3 | 1 | 4 | 2 | 4 | 2 | 5 | 3 | 2 |  | 3 | 1 | 48 |
| CC15 | 4 | 6 | 3 | 5 | 2 | 4 | 1 | 3 | 3 | 5 | 2 | 4 | 1 | 3 |  | 2 | 48 |
| CC16 | 6 | 4 | 5 | 3 | 4 | 2 | 3 | 1 | 5 | 3 | 4 | 2 | 3 | 1 | 2 |  | 48 |

**Algorithm 1** Confirm cluster allocations by the number of hops

```
Require: $C$ = [$cc\_1$,...,$cc\_16$] (16 CCs). $ORDER$ = [$sc_0$,...,$sc_n$]. $ma_i$
    ($mbp\_allocation$) represents the core allocations of $sc_i$ ($small\_cluster$).
    $ma_i$ = [0,...,N-1] (i=0,...,n). when $ma_i$ has determined allocations to $cc\_x$,
    then $cc\_x$ is plugged into E[$ma_i$].
 1: $i \leftarrow 0$
 2: while The allocation of $N$ pieces of $ma_i$ is determined do
 3:     $sc_i$ is extracted from $ORDER$.
 4:     if E[$ma_i$] has not been determined a cluster allocation yet then
 5:         if $sc_i$ has no input then
 6:             Allocate $ma_i$ to $cc\_x$ of $C$ which is randomly determined.
 7:         else
 8:             Allocate $ma_i$ to $cc\_x$ of $C$ with the smallest number of hops from
                E[$prev$] shown in Table 1. Note that $prev$ is $ma$ which has a de-
                pendency with $sc_i$.
 9:         end if
10:         Remove $cc\_x$ from $C$. E[$ma_i$] = $cc\_x$.
11:     end if
12:     $i \leftarrow i + 1$
13: end while
```

For example, if the order is determined as shown in Fig. 7, then *ORDER* is {2_0_0, 2_0_1, 2_0_2, 2_0_3, 3_0_1, 0_1_0, 1_1_0, ...}. 2_0_0 (*sc*) is extracted from *ORDER* (Line 2). Here, 2 (*ma*) has not been determined as a cluster allocation, E[2] is empty (Line 3). Since 2_0_0 has no input (Line 4), Line 5 is executed. Since CC6 (*cc*_6) is randomly determined in *C*, 2 (*ma*) is allocated to CC6, and then, *cc*_6 is removed from *C* and plugged into E[2] (Line 9). Then, since 2_0_1's, 2_0_2's, and 2_0_3's *ma* are 2 (Line 3) and E[2] has *cc*_6, these return to Line 1. 3_0_1's *ma* is 3 and E[3] is empty. Since 3_0_1 has dependencies with 2_0_3 (Line 4) as shown in Fig. 6, Line 7 is executed. Since 2_0_3's *ma* is 2, *prev* is 2. Since CC with the smallest number of hops from E[*prev*] (cc_6) is CC8, as shown in Table 1, 3 is allocated to CC8 (Line 7). Then, *cc*_8 is removed from *C* and plugged into E[3]. Note that, when there are two or more clusters with the smallest number of hops, the algorithm selects the cluster with the smallest cluster number. This process is repeated until the allocations of all clusters are determined. The proposed method can determine the application mapping after considering the number of hops and parallelization in the cluster, which is different from methods in current research.

**Divide information by each cluster**: After the cluster allocations are determined, it is necessary to decide how to allocate blocks to 16 cores in each cluster. Therefore, the method divides the information of the blocks by each cluster (each *mbp_allocation*) and generates N code and cycle-count annotated BLXMLs.

#### 3.2.2 Execution of the Second MBP

MAPA executes the core allocation phase of MBP to N code and cycle-count annotated BLXMLs in order to decide how to allocate blocks to the 16 cores in each cluster and generates N parallelized BLXMLs. In other words, 16*N core allocations of MPPA2-256 are determined by this execution of MBP. Then, MAPA integrates N parallelized BLXMLs to one parallelized BLXML. At this time, 16 cores belonging to CC1 are treated as core numbers 0 to 15, CC2's cores are treated as 16 to 31, CC3's cores are treated as 32 to 47, and so forth. Therefore, the parallelized BLXML with core numbers from 0 to 16*N - 1 is generated.

MAPA executes the code generation phase of MBP to the parallelized BLXML and generates a parallelized C code. However, we should rewrite the parallelized C code because it cannot execute on MPPA2-256.

#### 3.2.3 Rewriting Code Method for MPPA2-256

Since the parallelized C code generated from MBP cannot be implemented on MPPA, Rewriting Code Method for MPPA2-256 rewrites the code for MPPA2-256 and executes the following five items.

**1) Adding static to all thread functions**: MPPA2-256 generates execution files for each CC. If there is no static, all execution files capture all functions. In this case, a large amount of the code area of the memory is consumed. Therefore, the method adds static to all thread functions.

**2) Synchronizing the beginning of the thread functions**: In MPPA, CC1 to CC16 are activated in order. There is a considerable difference in activation time between CC1 and CC16. If MPPA2-256 immediately executes processing after starting up, then an error occurs when one CC communicates with other CCs that have not yet started. Therefore, the method synchronizes the thread function of each CC.

**3) Changing the thread activation process**: MPPA2-256 generates execution files for each CC. In each execution file, only the threads of the cores belonging to each CC need to be activated. Therefore, the method changes the thread activation process to run on only the CC containing each core according to the rules

(CC1: 0 to 15, CC2: 16 to 31, CC3: 32 to 47, and so forth) described in Section 3.2.2.

**4) Changing to cache thread ID**: The parallelized C code generated by MBP reads the thread ID of the communication destination each time communications between the cores occur. However, since another communication is required to read the thread ID, communication volume keeps increasing and the buffer is not sufficient to handle the load. Our method, therefore, reduces the communication traffic volume for reading the thread ID by caching it at the head of the C code.

**5) Adjusting the stack sizes of threads**: The memories that can be prepared for each CC are finite. In addition, the amount of stack required for each thread varies according to operations within a thread. Therefore, the method determines the stack sizes of the threads in each CC according to the amount of thread processing and memory area.

## 3.3 Mapping Algorithm from Core to Cluster (MACC)

MACC creates a core granularity XML from the MBP core allocation and performs cluster allocation. It then determines the cluster allocation by adjusting the core placement. If cluster allocation is created from core granularity XML, the same core allocation may be allocated to different clusters depending on the task. Therefore, adjust the core placement to make sure that the same cores are placed in the same cluster.

The MACC system model is shown in **Fig. 8**. It receives the parallelized BLXML containing the allocation information for 16*N cores and the parallelized C code and generates the core granularity XML. This generation part is similar to the Cluster Allocation Method described in Section 3.2.1. In Section 3.2.1, the BLXML is divided into clusters to generate the cluster granularity XML. However, MACC generates the core granularity XML without dividing the core. Next, MBP is used to allocate the clusters and adjust the core placement as described in Section 3.3.1. Finally, the parallelized C code is generated, and the code is rewritten using the Rewriting Code Method for MPPA described in Section 3.2.3.

### 3.3.1 Arranging Core Allocation Method (ACAM)

Cluster allocation using core granularity XML results in some cores being allocated to different clusters. The arranging core allocation method (ACAM) adjusts the placement of cores to ensure that tasks with the same core are allocated to the same cluster when viewed at core granularity.

As described above, Algorithm 2 adjusts the core placement. Line 3 signifies that no more than 16 cores can be allocated to the cluster. In Line 5, if all the tasks at the core granularity are assigned to the same cluster, no matter what, same cluster no matter what, then the cluster assignment is finalized. Lines 7–8 is the case where not all tasks at core granularity are assigned to the same cluster. In this case, it allocates the task to the cluster with the heaviest processing among the clusters to which it is allocated. In this way, the core placement is adjusted so that all tasks at core granularity are assigned to the same cluster.

## 3.4 Hybrid Algorithm

Hybrid Algorithm provides better core allocations than MAPA and MACC, whose advantages and disadvantages complement each other. MAPA has advantages when the number of clusters is large. However, when the number is small, MAPA sometimes does not output good results. Since reducing the number of clusters lowers the volume of communication traffic, MAPA's effect is reduced. On the other hand, MACC is more effective than MAPA when the number of clusters is smaller. Since MACC allocates tasks to equalize the amount of processing, good results are output even if the number of clusters is small. In addition, the algorithm that produces the best results depends on the Simulink model used as input. The hybrid algorithm executes both algorithms and produces better results by providing the best core allocations for any number of clusters.

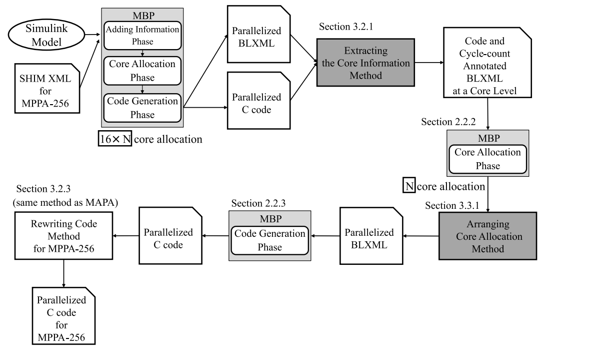

**Fig. 8** System Model of Mapping Algorithm from Core to Cluster (MACC).

**Algorithm 2** Reallocate cluster by the number of hops

```
Require: $C$ = [$cc\_1$,...,$cc\_16$] (16 CCs). $task_m$ (m = 0,...,n) (n = total number
    of tasks) represents the number of tasks. $c\_a_i$ (i = 0,...,16*N) represents
    the core allocations at granularity of $task_m$. $cc\_a_j$ (j = 0,...,N) represents
    the cluster allocations of $c\_a_i$. $cc\_a$ = [0,..,N] When $c\_a_i$ has determined
    allocations to $cc\_x$, $confirm[x] \leftarrow c\_a_i$.
 1: for $i \leftarrow 0$ to 16*N do
 2:   for $j \leftarrow 0$ to N do
 3:     if $c\_a$ placed on $confirm[x]$ is more than 16 then
 4:       break
 5:     else if all tasks to $c\_a_i$ are allocate to $cc\_a_j$ then
 6:       $x \leftarrow j$ and $confirm[x] \leftarrow c\_a_i$
 7:     else if tasks to $c\_a_i$ are allocate to more then two $cc\_a$ then
 8:       the heaviest processing $confirm[cc\_a] \leftarrow c\_a_i$
 9:     end if
10:   end for
11: end for
```

## 4. Evaluation

In this section, we describe the results of core allocation using the proposed method. First, we describe the evaluation environment. Next, we describe the evaluation results from assigning a Simulink model to each core of the cluster structure using the proposed method. Next, we evaluate and discuss the number of inter-cluster communications at each number of clusters. Finally, we evaluate the extent to which the proposed method reduces the burden on developers at the code level.

### 4.1 Environment

We used eMBP [5] developed by eSOL, which has the same functionality as MBP. We also used eMCOS as an OS and eM-

COS messages as the means for communication described in Section 2.3.3. We evaluated Hybrid Algorithm based on its execution time as compared with the MBP, MAPA, and MACC. The measurement of execution time is performed using an IOS that is not used for core allocations at this time, and all runtimes are performed under the same conditions. We also measured the communication count obtained via the algorithm.

### 4.2 Execution Time of Simulink Models

In this section, we used Simulink models consisting of various Simulink blocks such as Unary Minus, Add, Subsystem. We designed Simulink models with various block numbers (About 1,000 blocks, 1,500 blocks, 3,000 blocks). We applied the hybrid algorithm to these model assuming 4, 8, 12, and 16 clusters and compared them with the original core allocations of MBP (an existing method), MAPA, and MACC. The clusters of 4, 8, 12, and 16 have 64, 128, 192, and 256 cores, respectively.

We can see from **Fig. 9** that the execution time can be reduced by 2.9–7.2 times for MAPA and 2.5–3.8 times for MACC. In addition, in most cases, increasing the number of clusters shows better results. Comparing the results by the number of clusters, MAPA shows the best results for 4 and 16 clusters, and MACC shows the best results for 8 and 12 clusters.

Next, we compare the execution time for each number of clusters in MAPA and MACC used for the hybrid algorithm. The Simulink model used is a 1,500 block and 3,000 block model. This is the Simulink model of small scale, with the processing volume of one task of 1,500 blocks being a tenth of the processing volume of one task of 3,000 blocks.

The execution time of the model with 1,500 blocks is shown in **Fig. 10**. The execution time of MACC is faster than that of MAPA for 4, 8, and 12 clusters, and MAPA is faster than MACC for 16 clusters. As shown in **Fig. 11**, increasing the number of clusters in each algorithm reduces the execution time.

MACC has more allocations in the cluster allocation step than MAPA, by the number of cores per cluster. Therefore, MACC is more affected by the communication time than MAPA. If the number of clusters is large, the communication frequency increases and the communication time increases. In addition, if the processing time of one block is long, the waiting time in synchronous communication will increase and the communication time will be strongly affected. Therefore, when the processing time of one block is short and the number of clusters is small, the execution time of MACC is shorter than that for MAPA. Whereas, for models with a large number of blocks, the execution time of MAPA is shorter than that for MACC.

Next, the performance obtained based on the load balancing data is examined. The coefficient of variation, which is the standard deviation divided by the mean, is calculated as a value indicating the load variance. The values used in the calculations are the processing times for each cluster, and the results are shown in **Table 2**. In addition, a box-and-whisker diagram of the execution time for each cluster is shown in **Fig. 12**. The coefficient of variation for each algorithm is within 6.3%, and the results show that the variation is low. In addition, the hybrid algorithm improved the load balancing performance by 2.4% for 4 clusters of 1,500 blocks and by 4.1% for 16 clusters of 3,000 blocks. As the box-and-whisker diagram shows, the difference in processing time per cluster for each algorithm is almost negligible, and the variation becomes smaller as the number of clusters increases for MAPA and as the number of clusters decreases for MACC. From this result, we can see that the hybrid algorithm provides good load balancing. The hybrid algorithm has the shortest execution time, regardless of the number of clusters or blocks since both algorithms can be executed and give better results. Therefore, among the four algorithms, the hybrid algorithm has the best core allocation in the Simulink model.

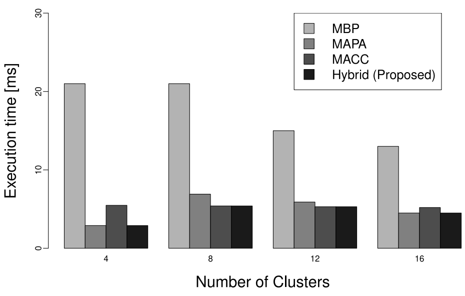

**Fig. 9** Execution time of Simulink Model (1,000 blocks).

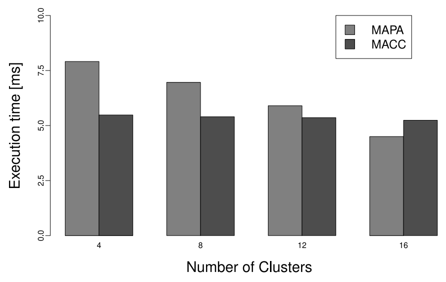

**Fig. 10** Execution time of Simulink Model (1,500 blocks).

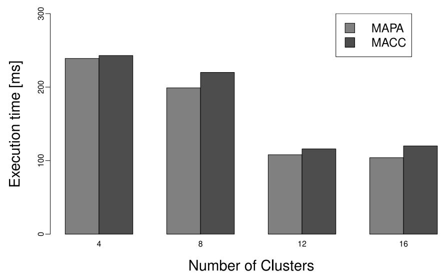

**Fig. 11** Execution time of Simulink Model (3,000 blocks).

**Table 2** Coefficient of variation of each algorithm based on the execution time of each cluster.

| | MAPA | | MACC | |
|---|---|---|---|---|
| | 1,500 blocks | 3,000 blocks | 1,500 blocks | 3,000blocks |
| 4 clusters | 0.036 | 0.031 | 0.012 | 0.010 |
| 8 clusters | 0.025 | 0.025 | 0.049 | 0.026 |
| 12 clusters | 0.023 | 0.018 | 0.054 | 0.050 |
| 16 clusters | 0.020 | 0.022 | 0.042 | 0.063 |

### 4.3 Communication time between the cluster

Inter-cluster communication has a longer communication time than inter-core communication, which affects the execution time. Therefore, we compared the number of inter-cluster communications for each algorithm. As shown in **Fig. 13**, the number of inter-cluster communications is significantly reduced for all algo-

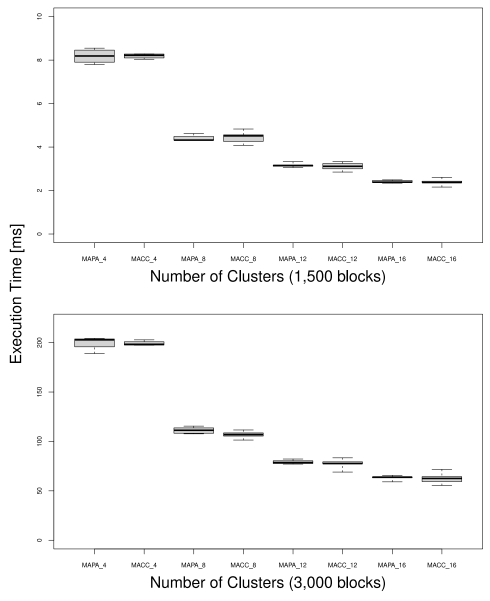

**Fig. 12** Box-and-whisker plot of processing time for each cluster.

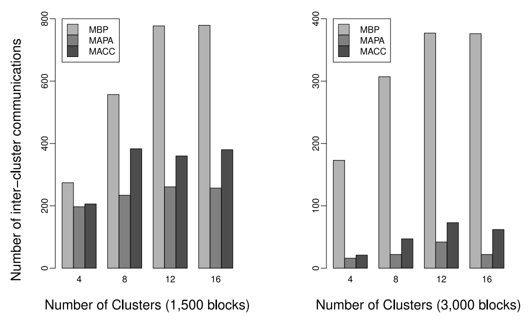

**Fig. 13** Number of communications between clusters

**Table 3** Reduction in number of lines of code

| | add | modify | total |
|---|---|---|---|
| (1) | 0 | $\alpha + 1$ | $\alpha + 1$ |
| (2) | $\alpha + 1$ | 2 | $\alpha + 3$ |
| (3) | $2 \times \alpha$ | 0 | $2 \times \alpha$ |
| (4) | 52 | $\beta + 2$ | $\beta + 54$ |
| (5) | 0 | $\alpha + 2$ | $\alpha + 2$ |
| total | $3 \times \alpha + 53$ | $2 \times \alpha + \beta + 7$ | $5 \times \alpha + \beta + 60$ |

(1): Adding static to all thread functions
(2): Synchronizing the beginning of the thread functions
(3): Changing the thread activation process
(4): Changing to cache thread ID
(5): Adjusting the stack sizes of threads
$\alpha$: The number of threads in the parallelized C code
$\beta$: The number of communication between cores and between clusters

rithms compared to the existing MBP method. From this result and the results of Section 4.2, MAPA is useful as an algorithm when the execution time of the input model assigned by MBP is greatly affected by the communication time, and MACC is useful when less affected by the communication time.

### 4.4 Reduction of the C Code Rewriting Burden

Rewriting Code Method for MPPA2-256 described in Section 3.2.3 rewrites the parallelized C code generated by MBP.

**Table 3** shows the extent to which our method reduces the burden on the user. The numbers in the leftmost column are linked with the numbers in Section 3.2.3. Here, *add* and *modify* signify the numbers of lines of the C code that have been newly inserted and modified, respectively, by the method. Next, $3 \times \alpha + 53$ lines of the C code are added and $2 \times \alpha + \beta + 7$ lines of the C code are modified. As a result, the proposed algorithm reduces the user burden in rewriting $5 \times \alpha + \beta + 60$ lines of the C code. For example, the Simulink model of 1,000 blocks mentioned in Section 4.2 has 256 threads and 322 communications when we executed MAPA while assuming 16 clusters. Thus, the method automatically rewrites 1,662 lines and decreases the user burden.

## 5. Related Work

AMALTHEA platform [7] was created to address the challenges in parallelism exploitations for multi-core systems. The platform creates meaningful solutions by providing effective processes for application distributions. Various methods such as the Critical Path Partitioning and Earliest Start Scheduling are introduced in the partitioning phase.

OSCAR Compiler [15] breaks down the behavior of the software into tasks of various granularities, analyzes the tasks in detail, and allocates them to multiple cores to accelerate the processing. Unlike MBP, which exploits models, it takes a sequential code that operates from a single core as its input.

Kumura et al. [9] described a method to generate a parallelized C code from Simulink models. To ease the parallelization, this paper analyzed the model and broke loop structures by dividing indirect feedthrough blocks. Then, task allocation and task scheduling are provided using a symmetric multi-processing OS.

Zhong et al. [17] used a mixed-integer linear programming formulation to determine core allocations while considering load balancing and inter-core communications across cores with respectively different performance. Moreover, this paper evaluated the approach to a single-ISA heterogeneous multi-core processor.

Kojima et al. [8] proposed two methods: the Remapping Blocks Method (RBM) which remaps blocks to cores by the results of MBP and the Deciding Execution Order Method (DEOM) which decides the execution order of the block. The methods enhance the parallelism of blocks and improve load balance.

SLX tool [11] profiles a sequential C code and finds functions with many executions. SLX tool consists of three parts: SLX Parallelizer, SLX Mapper, and SLX Generator. Parallelism is extracted by SLX Parallelizer from the sequential C code, and core allocations are determined by SLX Mapper. Then, a parallelized C code is generated by SLX Generator.

**Table 4** briefly summarizes the characteristics of several related tools and compares them with the proposed algorithm. AMALTHEA platform, OSCAR Compiler, MBP, and RBM and DEOM do not assume the use of hardware with cluster structures. The NoC SLX tool does assume their use but the tool does not use MATLAB/Simulink.

## 6. Conclusion

This paper proposes an algorithm to determine core allocations to many-core hardware with cluster structures such as MPPA2-256. If users need to use N clusters, the Mapping Algorithm using Path Analysis (MAPA) uses the 16*N core allocations of MBP and determines cluster allocations according to the communication contention of NoC. Mapping Algorithm from Core

**Table 4** Comparison of proposed and previous methods

| | Embedded | MATLAB/ Simulink | Path Analysis | Cluster- Structure | NoC |
|---|---|---|---|---|---|
| AMALTHEA platform [7] | ✓ | | ✓ | | |
| OSCAR Compiler [15] | ✓ | ✓ | ✓ | | |
| Model Based Parallelization [9] | ✓ | ✓ | | | |
| MBP [17] | ✓ | ✓ | ✓ | | |
| RBM and DEOM [8] | ✓ | ✓ | ✓ | | |
| SLX tool [16] | ✓ | | ✓ | ✓ | ✓ |
| Proposed Algorithm | ✓ | ✓ | ✓ | ✓ | ✓ |

to Cluster (MACC) executes 16*N core allocations of MBP to create XML at core granularity. Next, it uses the core granularity XML to perform the allocation of the N clusters. Finally, the finer details of the allocation are modified, and the allocation is determined. We evaluated these algorithms with the original MBP core allocations. Since the results of the Simulink show that Hybrid Algorithm has the shortest execution time, Hybrid Algorithm's core allocation is the best among the four algorithms. In future work, we will propose an algorithm that decides the optimal number of clusters according to a model. In addition, we will evaluate the system using highly parallel application models such as point cloud processing for self-driving systems.

**Acknowledgments** This work was partially supported JST PRESTO Grant Number JPMJPR1751.

## References

[1] Azumi, T., Maruyama, Y. and Kato, S.: ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform, *Proc. IROS* (2020).

[2] Dally, W.J. and Towles, B.: Route packets, not wires: On-chip interconnection networks, pp.684–689 (2001).

[3] De Dinechin, B.D., Van Amstel, D., Poulhiès, M. and Lager, G.: Time-critical computing on a single-chip massively parallel processor, *Proc. DATE*, pp.1–6 (2014).

[4] Embedded Multicore Consortium: Embedded Multicore Consortium, Embedded Multicore Consortium (online), available from ⟨http://www.embeddedmulticore.org/⟩ (accessed 2021-05-10).

[5] eSOL: eMBP, eSOL Co., Ltd. (online), available from ⟨https://www.esol.com/embedded/mbp.html⟩ (accessed 2021-05-10).

[6] Honda, K. and Azumi, T.: Performance Estimation for Many-core Processor in Model-Based Development, *Proc. MECO*, pp.1–6 (2019).

[7] Höttger, R., Krawczyk, L. and Igel, B.: Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems, *Proc. ICPDSSE* (2015).

[8] Kojima, S., Edahiro, M. and Azumi, T.: Remapping Method to Minimize Makespan of Simulink Model for Embedded Multi-core Systems, *Proc. CATA* (2018).

[9] Kumura, T., Nakamura, Y., Ishiura, N., Takeuchi, Y. and Imai, M.: Model Based Parallelization from the Simulink Models and Their Sequential C Code, *Proc. Workshop on SASIMI*, pp.186–191 (2012).

[10] Multicore Association: Software-Hardware Interface for Multi-Many-core (SHIM), Multicore-Association (online), available from ⟨http://www.multicore-association.org/workgroup/shim.php⟩ (accessed 2021-05-10).

[11] Onnebrink, G., Hallawa, A., Leupers, R., Ascheid, G. and Shaheen, A.-U.-D.: A heuristic for multi objective software application mappings on heterogeneous MPSoCs, *Proc. ASPDAC*, pp.609–614 (2019).

[12] Perret, Q., Maurere, P., Noulard, E., Pagetti, C., Sainrat, P. and Triquet, B.: Temporal Isolation of Hard Real-Time Applications on Many-Core Processors, *Proc. RTAS*, pp.1–11 (2016).

[13] The MathWorks, Inc.: Embedded Coder, The MathWorks, Inc. (online), available from ⟨http://jp.mathworks.com/products/embedded-coder/⟩ (accessed 2021-05-10).

[14] The MathWorks, Inc.: MATLAB/Simulink, The MathWorks, Inc. (online), available from ⟨http://www.mathworks.com/products/simulink/⟩ (accessed 2021-05-10).

[15] Umeda, D., Suzuki, T., Mikami, H., Kimura, K. and Kasahara, H.: Multigrain Parallelization for Model-Based Design Applications Using the OSCAR Compiler, *Proc. LCPC*, pp.125–139 (2016).

[16] Xilinx: SLX tool, Xilinx (online), available from ⟨https://silexica.com/⟩ (accessed 2021-05-10).

[17] Zhong, Z. and Edahiro, M.: Model Based Parallelizer for Embedded Control Systems on Single-ISA Heterogeneous Multicore Processors, *Proc. ISOCC* (2018).

**Yutaro Kobayashi** is a master student of Graduate School of Science and Engineering, Saitama University. He received his B.E. degree from School of Science and Engineering, Saitama University in 2021.

**Kentaro Honda** received his M.E. degree from Graduate School of Science and Engineering, Saitama University in 2021.

**Sasuga Kojima** received his M.E. degree from Graduate School of Engineering Science, Osaka University in 2019.

**Hiroshi Fujimoto** has been a member of Technology Headquarters in eSOL Co., Ltd., since October 2015. He is engaged in the development of parallel processing software that runs on multi/many-cores.

**Masato Edahiro** is a Professor at the Graduate School of Informatics Nagoya University. He received his Ph.D. degree in computer science from Princeton University, Princeton, NJ, USA, in 1999. He joined NEC Corporation in 1985, and was with its research center for 26 years, and moved to Nagoya University, Nagoya, Japan, in 2011. His research interests include graph and network algorithms and software for multi/many-core processors.

**Takuya Azumi** is an Associate Professor at the Graduate School of Science and Engineering, Saitama University. He received his Ph.D. degree from the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists for Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor at the College of Information Science and Engineering, Ritsumeikan University. From 2014 to 2018, he was an Assistant Professor at the Graduate School of Engineering Science, Osaka University. His research interests include real-time operating systems and component-based development. He is a member of IEEE, ACM, IEICE, and JSSST.