Multigrain Parallelization for Model-based Design Applications Using the OSCAR Compiler

Dan Umeda, Takahiro Suzuki, Hiroki Mikami,
Keiji Kimura, and Hironori Kasahara

Green Computing Systems Research Center
Waseda University - Tokyo, Japan - Tel./Fax. +81-3-3203- 4485/4523
{umedan,taka,hiroki}@kasahara.cs.waseda.ac.jp,
{keiji,kasahara}@waseda.jp,
http://www.kasahara.cs.waseda.ac.jp/

Abstract. Model-based design is a very popular software development method for developing a wide variety of embedded applications such as automotive systems, aircraft systems, and medical systems. Model-based design tools like MATLAB/Simulink typically allow engineers to graphically build models consisting of connected blocks for the purpose of reducing development time. These tools also support automatic C code generation from models with a special tool such as Embedded Coder to map models onto various kinds of embedded CPUs. Since embedded systems require real-time processing, the use of multi-core CPUs poses more opportunities for accelerating program execution to satisfy the real-time constraints. While prior approaches exploit parallelism among blocks by inspecting MATLAB/Simulink models, this may lose an opportunity for fully exploiting parallelism of the whole program because models potentially have parallelism within a block. To unlock this limitation, this paper presents an automatic parallelization technique for auto-generated C code developed by MATLAB/Simulink with Embedded Coder. Specifically, this work 1) exploits multi-level parallelism including inter-block and intra-block parallelism by analyzing the auto-generated C code, and 2) performs static scheduling to reduce dynamic overheads as much as possible. Also, this paper proposes an automatic profiling framework for the auto-generated code for enhancing static scheduling, which leads to improving the performance of MATLAB/Simulink applications. Performance evaluation shows 4.21 times speedup with six processor cores on Intel Xeon X5670 and 3.38 times speedup with four processor cores on ARM Cortex-A15 compared with uniprocessor execution for a road tracking application.

Key words: automatic parallelization, multi-core, model-based design, MATLAB/Simulink, automatic code generation

1 Introduction

The Model-based design like MATLAB/Simulink [1] has been widely used since it enables high software productivity in reduced turn-around times for embedded

systems [2]. Commercial model-based design tools support auto-code generation from a model that is represented by a block diagram [3, 4]. MATLAB/Simulink is one of the most popular tools for the model-based design of automotive systems, aircraft systems, and medical systems. This tool can generate C/C++ code for embedded systems with Embedded Coder [5] (formerly known as Real-Time Workshop). The automatic code generation feature saves programmers from developing embedded applications in error-prone programming languages, however, this code generator does not optimize the application for target systems. Of course, it does not parallelize the application, even though a target system has a multi-core processor.

Several approaches have been proposed to utilize multi-cores for the application developed in MATLAB/Simulink. Some products have supported semi-automatic parallelization techniques for a multi-core processor using task partitioning by an application developer [6, 7]. These technique can achieve a functional distribution of MATLAB/Simulink application, but cannot reduce load balancing which is most important for embedded real-time application. In addition, these tools support parallel processing in limited environments for simulation using Simulink. For an automatic parallelization of MATLAB/Simulink applications, Arquimedes et al. proposed an automatic equation-level parallelization technique of a Simulink model [8]. Their approach exploited parallelism among Mealy blocks such as integrators, derivatives, unit delays and so on. However, their method is only applicable to applications for simulation including mealy blocks. This approach does not focus on embedded systems. As an automatic parallelization technique for embedded applications, Kumura et al. proposed a model based parallelization by analyzing of dependencies from block connections among Simulink blocks [9]. This technique makes it possible to perform a parallel processing by exploiting block level parallelism from a model. However, exploiting this parallelism does not always allow us to exploit the full capability of multi-cores since a granularity of task depends on how the MATLAB/Simulink users define a block. A model information is too abstract to represent multi-grain parallelism including parallelism intra-blocks such as library Simulink blocks and users customized blocks. Therefore, this may lose an opportunity for optimizations, for example, by causing unequal workload on each core.

Unlike these prior approaches, this paper proposes an automatic parallelization method using an automatic multigrain parallelizing compiler, or the OSCAR compiler [10] from auto-generated C code developed by MATLAB/Simulink. While this approach successfully analyzes the C code because it is easy for the compiler to exploit parallelism using pattern matching and the code does not require a sophisticated pointer analysis for readability and MISRA-C, it is possible that future versions of Embedded Coder could limit the analysis of parallelism. The compiler exploits both of coarse grain parallelism inter-block and loop level parallelism intra-block from the auto-generated C code. Then, the compiler adjusts a task granularity with the minimum overhead by performing inline expansion and task fusion for conditional branches to improve the

utilization of each core. After optimization, the compiler assigns parallel task onto processor cores using a static task scheduling considering profiling information on MATLAB/Simulink. Then, the compiler finally generates parallelized C code regardless of target processors. Although this paper focuses on the application developed by MATLAB/Simulink, the proposed method has a potential to apply to other model-based design tools since it exploits parallelism from the auto-generated C code regardless of a grammar of the tools. The features of the proposed method include:

- Fully automatic parallelization technique of the C code generated by a model-based design tool for embedded systems without dependence on a grammar of this tool.
- Construction of automatic profiling framework for a MATLAB/Simulink model to improve performance of the statically scheduled parallel code.
- Multigrain parallelization technique of model-based design applications that enables to overcome the limitation of the block level parallelization technique that is common in the field of model-based design.

The rest of this paper is organized as follows: Section 2 provides a framework for parallelization of model-based design applications. Section 3 introduces how to exploit parallelism from MATLAB/Simulink application using the OSCAR compiler. Section 4 describes multi-grain parallel processing method for the applications. Section 5 shows performance evaluation for the applications using the proposed method. Finally, section 6 represents some conclusions.

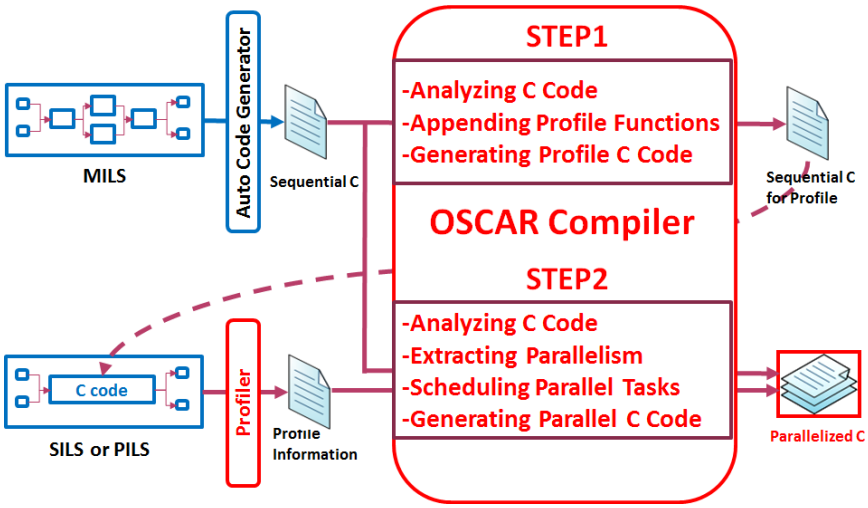

Fig. 1. Overview of the proposed framework for parallelization of model-based design applications

2 Framework for Parallelization of Model-based Design Applications

This section describes a framework for parallelization of model-based design applications. The model-based design tools with an automatic code generator like MATLAB/Simulink are widely used since it enables high software productivity for embedded systems. However, the code generator like Embedded Coder does not optimize the application for target multi-cores. Therefore, several researchers have proposed the parallelization technique for the application on multi-cores. The previous works [8] and [9] analyzed a model file *model.mdl* to exploit parallelism among blocks in the model. This may lose an opportunity to exploit the whole parallelism in the model. For example, their approaches lose to exploit hierarchical multigrain parallelism, even though a model has parallelism inner Simulink blocks. It may cause unequal workload on each core. Additionally, they depend on a grammar of the model-based design tool. Indeed, the model file *model.mdl* have changed to a new model file *model.slx* from MATLAB R2012a.

In contrast, our proposed method analyzes auto-generated C code developed by MATLAB/Simulink with Embedded Coder to exploit hierarchical multigrain parallelism which is not represented in the model file. This approach does not depend on the grammar of model-based design tools since it analyzes the code to extract parallelism. Additionally, the proposed framework uses profiling information including execution counts and time to handle dynamic features of programs such as conditional branches and fluctuations in the number of iterations of loops.

Fig.1 shows an overview of the proposed framework. At the step1, the OSCAR compiler analyzes C code that is generated by Embedded Coder from a model. Then, the compiler instruments a sequence of C code inserting profile functions and the MATLAB/Simulink interface (MEX function [11]). This code is used to gather profiling information about a program execution on MATLAB/Simulink. Thereby, this framework can gather the profiling information in software-in-the-loop simulation (SILS) or processor-in-the-loop simulation (PILS) on the model-based design tool. Then, the profiler generates the profiling information during executing a model including the profile C code. At the step2, the compiler analyzes the auto-generated C code and exploits hierarchical multigrain parallelism in the whole program. After exploiting parallelism, the compiler schedules parallel tasks onto processor cores and finally generates parallelized C code using the profiling information.

3 Exploiting Parallelism Using the OSCAR Compiler

This section explains a method to exploit multigrain parallelism from auto-generated C code developed by MATLAB/Simulink using the OSCAR compiler.

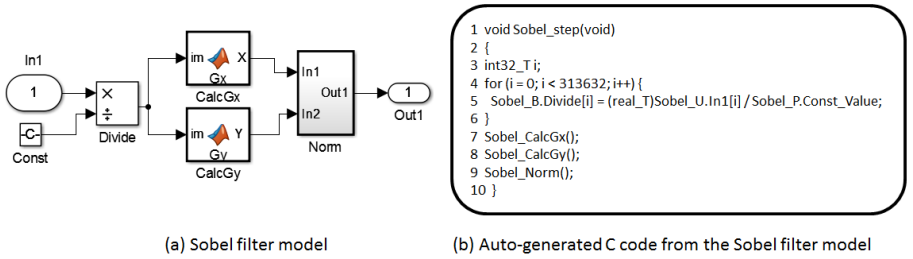

Fig. 2. Sample Simulink model and auto-generated C code from the model using the Embedded Coder

3.1 Example of MATLAB/Simulink Application

This paper takes an example of MATLAB/Simulink application to describe the parallelism in it. The example is simple to explain parallelism of the model, however, real applications are too sophisticated to extract all parallelism because there are many of block connections and feedback loops. Therefore, it is difficult to achieve efficient performance on multi-cores using manual parallelization. Fig.2-(a) shows a model of Sobel filter that performs edge detection of a binary image. It consists of a `Const` block, a `Divide` block, `MATLAB Function` blocks (user's library functions) named as `CalcGx` and `CalcGy`, and a `Subsystem` block named as `Norm`. Evidently, the model has parallelism among `CalcGx` and `CalcGy` because there is no connection among them.

Fig.2-(b) shows auto-generated C code from the model in Fig.2-(a) by Embedded Coder. A loop as shown in line 4-6 corresponds to the `Divide` block in Fig.2-(a). Each of functions of `Sobel_CalcGx` and `Sobel_CalcGy` corresponds each of the `MATLAB Function` blocks named as `CalcGx` and `CalcGy` in Fig.2-(a). A function of `Sobel_Norm` corresponds to the `Subsystem` block named as `Norm` in Fig.2-(a).

3.2 Coarse Grain Task Parallel Processing

Coarse grain task parallel processing uses parallelism among three kinds of coarse grain tasks, namely macro-tasks (MTs). Parallelism is expressed graphically as a macro-task graph (MTG) including data dependencies and control dependencies among MTs. The MTs on the MTG are assigned to processor cores by a static or a dynamic task scheduling method. As a result of the assignment, the OSCAR compiler generates parallelized C code while preserving the original semantics of the program.

Generation of Macro-tasks In the coarse grain task parallelization of the OSCAR compiler, auto-generated C code from a MATLAB/Simulink model is decomposed into the MTs. The MTs include basic blocks (BBs), repetition blocks or loops (RBs), and subroutine blocks (SBs). The MTs can be hierarchically

defined inside each sequential loop or a function [10]. Moreover, the RB is transformed LOOP, which means the compiler analyzes this loop as a sequential loop, or DOALL which means the compiler analyzes this loop as a parallelizable loop.

Exploiting of Coarse Grain Task Parallelism After generation of MTs, data dependencies, and control flow among MTs are analyzed. The compiler generates a hierarchical macro-flow graph (MFG) which represents control flow and data dependencies among MTs [10].

Then, the Earliest Executable Condition Analysis [10] is applied to the MFG to exploit coarse grain task parallelism among MTs by taking into account both the control dependencies and the data dependencies. This analysis generates a hierarchical macro-task graph (MTG). The MTG represents coarse grain task parallelism among MTs. If SB or RB has nested inner layer, MTGs are generated hierarchically. Fig.3 shows a hierarchical MTG of the C code in Fig.2-(b). Nodes represent MTs. Small circles inside a node represents conditional branches, for example, `bb1` and `bb4` in MTG2-1. Solid edges represent data dependencies. Dotted edges in MTG2-1, MTG3-1, and MTG4-1 represent extended control dependencies. The extended control dependency means ordinary control dependency and the condition on which a data dependence predecessor of an MT is not executed. Solid and dotted arcs, connecting solid and dotted edges have two different meanings. The solid arc represents that edges connected by the arc are in AND relationship. The dotted arc represents that edges connected by the arc are in OR relationship. In an MTG, edges having arrows represents original control flow edges or branch direction.

`sb2` and `sb3` in MTG0 are in parallel. Therefore, block level parallelism among `CalcGx` and `CalcGy` in Fig.2-(a) are exploited from the auto-generated C code. Additionally, loop level parallelism which is not represented in Fig.2-(a) is exploited from the auto-generated C code since the compiler analyzes `doall` in Fig.3 as parallelizable loops. Therefore, coarse grain task parallel processing using the compiler allows us to exploit hierarchical multigrain parallelism of MATLAB/Simulink applications from the auto-generated C code.

Scheduling of Coarse Grain Task onto Multi-cores After exploit of hierarchical multigrain parallelism, a static task scheduling or a dynamic task scheduling is chosen for each MTG to assign MTs onto multi-cores. If an MTG has only data dependencies and is deterministic, a static task scheduling at compilation time is applied to it by the OSCAR compiler. In the static task scheduling, the compiler uses four heuristic scheduling algorithms including CP/ETF/MISF, ETF/CP/MISF, DT/CP/MISF and CP/DT/MISF [12]. The compiler chooses the best schedule from those scheduling. If an MTG is non-deterministic by conditional branches or runtime fluctuations among MTs, the dynamic task scheduling at runtime is applied to it to handle the runtime uncertainties. The compiler generates dynamic task scheduling routines for non-deterministic MTGs and inserts it into a parallelized code. The static task scheduling is generally more

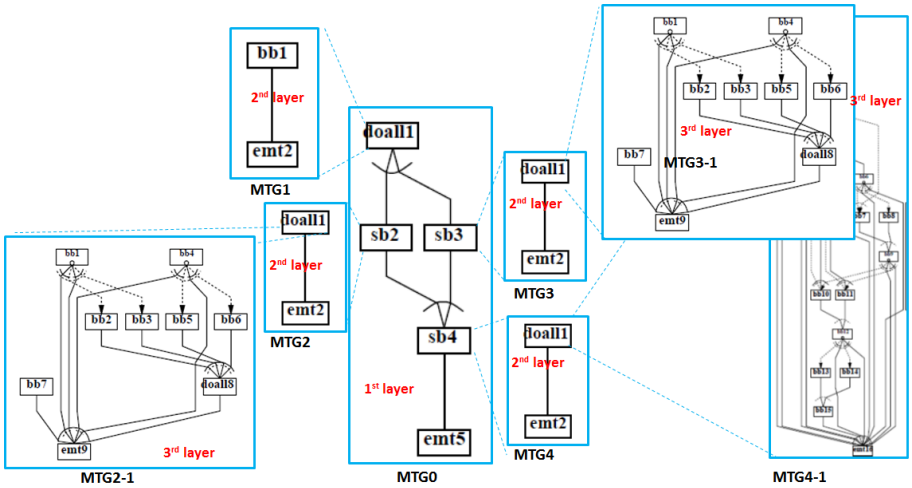

Fig. 3. Hierarchical MTG

effective than the dynamic task scheduling since it can minimize data transfer and synchronization an overhead without a runtime scheduling overhead.

Parallelized C Code Generation Using the OSCAR API The OSCAR compiler generates parallelized C code with the OSCAR API [13] that is designed on a subset of OpenMP for preserving portability over a wide range of multi-core architectures. If data is shared on threads, the compiler inserts synchronizing instructions using spin locks. Additionally, MEX functions are inserted as necessary to execute parallelized C code in the SILS or PILS on MATLAB/Simulink.

4 Multigrain Parallel Processing Method for MATLAB/Simulink Applications

This section describes a proposed multigrain parallel processing method for MATLAB/Simulink applications. Embedded applications are generally executed repeatedly within a short period. Therefore, reducing overhead as much as possible is important for efficient parallel processing on multi-cores. The proposed method enables us to parallelize the application using hierarchical multigrain parallelism with a minimum overhead for embedded systems. The kernel technique is to generate the statically scheduled parallel code using multigrain parallelism. The proposal method consists of the following steps.

step1 Automatic profiling in SILS or PILS on MATLAB/Simulink to handle dynamic features of programs.

step2 Inline expansion to exploit more parallelism over hierarchies or program structure.
step3 Macro task fusion for conditional branches to generate statically scheduled parallel code.
step4 Converting loop level parallelism into task level parallelism to perform efficient parallel processing among loops and other MTs without an overhead of loop level parallelization.

The following provides details of the proposed method.

4.1 Automatic Profiling in model-based development

Profiling is an important technique for improving the preciseness of static task scheduling by a parallelizing compiler. Moreover, it is particularly effective for handling dynamic features of programs such as conditional branches and the fluctuations in the number of loop iterations. For this purpose, the compiler generates a sequence of code to collect profiling information. Additionally, MEX functions as the interface between C code and MATLAB/Simulink are inserted into this code to obtain the profiling information in the SILS or the PILS on the model-based tool. Two types of profile functions are inserted immediately before and after each MT. The one is a function to measure execution counts of each MT. This information is utilized for estimating branch probability and the number of loop iterations. The other is a function to measure the execution time of each MT. This information is utilized for optimization and the static task scheduling in the compiler. In the other words, execution counts and time in the level of MT are attained with executing the code. The profiler finally generates the profiling information including longest path, shortest path, and average path in repeated executions during executing a model including the profile C code.

4.2 Inline Expansion

The OSCAR compiler generates a hierarchical MTG to perform hierarchical parallelization [10]. It is effective to perform parallel processing for applications having large execution time, for example, simulation of scientific computation. However, real embedded applications are generally executed repeatedly within a short period. Therefore, it is not enough parallelism to parallelize efficiently in each hierarchy. Thus, the proposed method uses an inline expansion technique [14] to exploit multigrain parallelism from programs over hierarchies or nested levels. This technique analyzes parallelism after each SB is inline expanded. After the analysis, the compiler selects SBs to improve parallelism and expands them. Fig.4 shows an overview of the inline expansion technique. In Fig.4-(a), it is not enough parallelism to parallelize hierarchically in MTG0, MTG1, and MTG3. The inline expansion applies `sb2` in MTG0 including parallelism inner the block to improve parallelism. As a result, the compiler generates an MTG in Fig.4-(b). As shown in Fig.4-(b), more coarse grain parallelism is exploited than that of the MTG in Fig.4-(a).

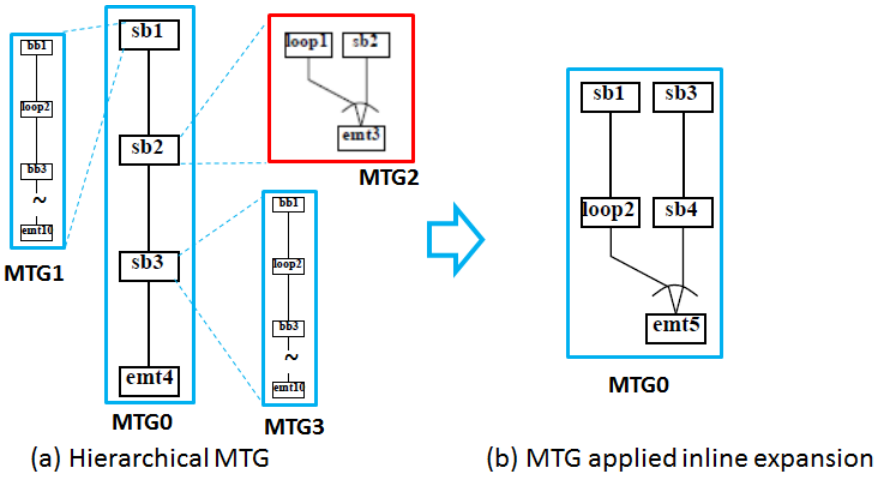

Fig. 4. Overview of the inline expansion technique

4.3 Macro Task Fusion

The OSCAR compiler has two types of task scheduling as mentioned in section 3.2. The one is the dynamic task scheduling that is applied to an MTG including conditional branches. The other is the static task scheduling that is applied to an MTG including no conditional branches. The static task scheduling is preferable for parallelization of the embedded applications because of its few runtime overhead. However, most of MATLAB/Simulink applications have `Switch`, `Saturation` and `Trigger` blocks that are converted into if-statements by Embedded Coder. It introduces to choose the dynamic task scheduling including the runtime overhead. Since these conditional branches cannot be handled by the static task scheduling, the proposed scheme applies macro task fusion to MFG to hide conditional branches inside MTs. The method is described as follows.

step 1 Search MFG nodes having a conditional branch.
step 2 For each conditional branch node found in step 1, apply step 3-6.
step 3 Search a set of MFG nodes that is post-dominated by the conditional branch node.
step 4 Define a post-dominator node having a minimum number of the MT with the exception of the conditional branch node as an exit node.
step 5 Merge a group from the conditional branch node and the exit node into a single MT.
step 6 Generate a fused MT including conditional branches inner the MT.

This process eliminates all conditional branches from an MFG. After the technique, if the fused MT has enough parallelism inner the MT, duplications of if-statements [15] is applied to it for an improvement of parallelism.

Fig.5 shows an overview of the macro task fusion technique. At the step 1, the compiler searches MFG nodes having a conditional branch from an MFG. At the step 2, the compiler applies step 3-6 to each conditional branch node found step 1. At the step 3, the compiler searches a post-dominator of the conditional branch node. At the step 4, the compiler chooses a node having a minimum number in the post-dominators with the exception of the conditional branch node. Then, the node is defined as an exit node of the conditional branch node. At the step 5, the compiler merges a group from the conditional branch node and the exit node into a single MT. As a result, the compiler generates a fused MT including conditional branches inner the MT at the step 6.

In this example, the compiler chooses `bb1` and `bb7` having a small circle inside a node that represents a conditional branch in Fig.5-(a). In Fig.5-(a), `bb1` dominates `bb1`, `bb5`, `sb6`, `bb7`, `bb10` and `emt11`. Additionally, `bb7` dominates `bb7`, `bb10` and `emt11`. Therefore, the compiler chooses `bb5` and `bb10` as the exit node for each of the conditional branch nodes. Merging `bb1-5` and `bb7-10`, the compiler generates an MFG as shown in Fig.5-(b). In this figure, `block` shows the merged MT by the technique. Exploiting parallelism using the Earliest Executable Condition Analysis, the compiler generates an MTG as shown in Fig.5-(c). Then, the duplication of if-statements applies to inner `block3` not to eliminate parallelism. `bb1` including if-statements is duplicated, and `block3` is divided into two nodes. As a result, the compiler generates an MTG having duplicated MTs such as `block3` and `block4` as shown in Fig.5-(d). Thus, a compiler coarsens MTs without losing parallelism and can apply static task scheduling without runtime overhead to an MTG having conditional branches.

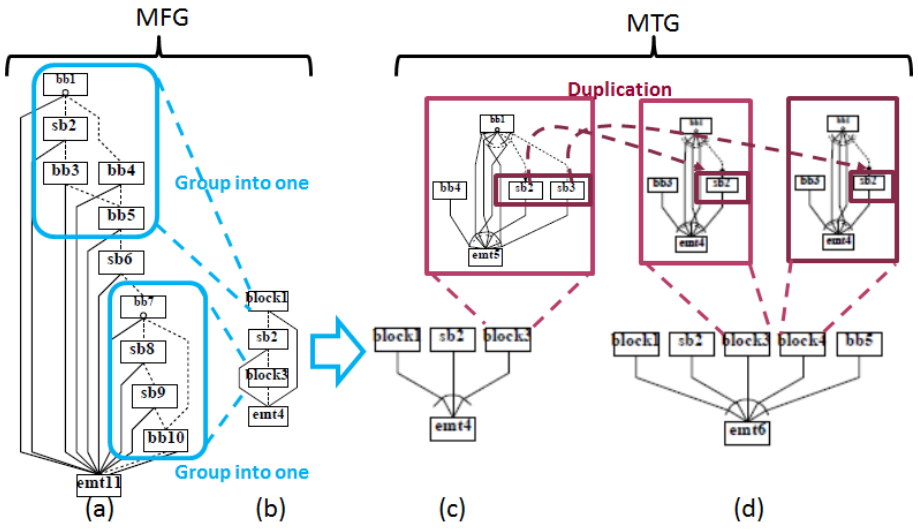

Fig. 5. Overview of the macro task fusion technique

4.4 Converting Loop Level Parallelism into Task Level Parallelism

Kumura et al. [9] has proposed the block level parallelization technique from the data flow graph of the block diagram of a model. This method enables us to exploit parallelism among blocks in the model. However, it is difficult to exploit parallelism in a block using only information of the block diagram. In contrast, this paper proposes the multigrain parallelization technique from auto-generated C code. The code level analysis in this method enables us to exploit loop level parallelism in addition to task level parallelism. In this paper, parallelizable loops shown as Doall are decomposed into `n` small Doalls (or MTs) statically to perform parallel processing efficiently without a runtime overhead of loop level parallelization. In this method, the `n` is a number decided by less than a number of processor cores and T_{min}. T_{min} is defined as a minimum task cost for loop level parallelization considering overheads of parallel thread fork/join and task scheduling on each target multi-core [10].

These decomposed small Doalls can be executed in parallel among other MTs. After parallelizable loop decomposition, the static task scheduler in section 3.2 assigns all MTs including decomposed Doalls onto processor cores.

5 Performance Evaluation of MATLAB/Simulink Applications on Multi-cores

This section describes performance evaluation of the proposed multigrain parallelization technique for MATLAB/Simulink applications on several multi-core platforms.

5.1 Target MATLAB/Simulink Applications

This section evaluates the performance on Intel and ARM multi-cores using three important applications such as road tracking for self-driving cars, vessel detection for medical image recognition, and anomaly detection for pattern recognition really used an industry. These applications have both parallelism among Simulink blocks and inner a block. Therefore, they are suitable to be parallelized by the automatic multigrain parallelization technique. Each application is described in the following.

Road Tracking. Road tracking in a model of [16] is an image processing to detect and track edges set in primarily residential settings where lane markings may not be present. The model has over one hundred Simulink blocks and block level parallelism among Simulink blocks in the left road and right road. The size of an input image is 320 x 240 pixels. In this evaluation, `For Iterator` blocks are expanded and S-Function blocks of parallel Hough transformation [17] are used instead of library `Hough Transformation` block to be close real embedded applications.

Vessel Detection. Vessel detection model implemented from [18] is an image processing to detect vessels from retinal images for a diagnosis of various eye diseases. The model is simplest in the three applications and includes one `Data Type Conversion`, one `MinMax`, one `Switch` and eight `MATLAB Function` blocks using the Kirsch's edge operator blocks. The operator blocks are in parallel. The size of an input image is 200 x 170 pixels.

Anomaly Detection. Anomaly detection model is a real product application in A&D CO., LTD. and an image processing to detect anomaly from an input image. The model is most complex and has longest execution time in the three applications. It includes `morphological opening`, `morphological dilation`, `blob analysis` blocks and so on. There is parallelism among some image processing block. The size of the input image is 600 x 600 pixels.

5.2 Evaluation Environment.

This evaluation uses the Intel Xeon X5670 and the ARM Cortex-A15. The Xeon X5670 processor has six processor cores with each processor core running at 2.93 GHz. Each processor core has 32 KB L1-cache and 256 KB L2-cache. 12 MB L3 cache is shared on six processor cores. The Cortex-A15 processor has four 1.60 GHz processor cores. Each processor core has 32 KB L1-cache, and four processor cores has a shared 2 MB L2-cache.

5.3 Performance Evaluation on multi-cores

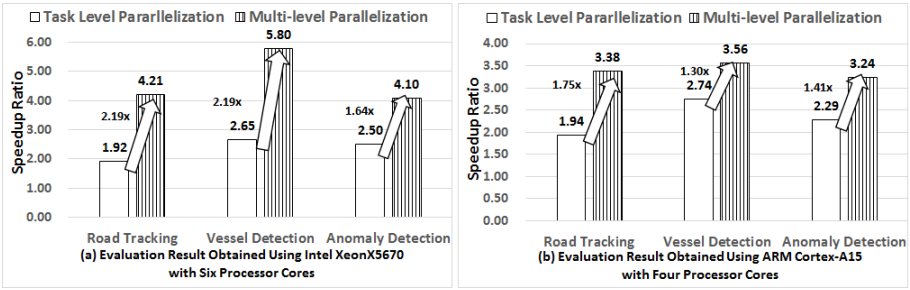

Fig. 6. Speedup ratio for MATLAB/Simulink applications on Intel and ARM multi-cores

Fig.6-(a) and (b) shows average speedup obtained by using only the task level parallelization technique that is similar to the single level parallelization technique in [9] and the multigrain parallelization technique corresponds to proposed method on Intel Xeon X5670 and ARM Cortex-A15. The speedups in Fig.6-(a) and (b) are relative to sequential execution using only one core of each

application. 1.92 times speedup for the road tracking application, 2.65 times speedup for vessel detection application and 2.50 times speedup for the anomaly detection application can be achieved using the task level parallelization technique on Intel Xeon X5670 with six processor cores. On ARM cortex-A15 with four processor cores, 1.94 times speedup for the road tracking application, 2.74 times speedup for the vessel detection application and 2.29 times speedup for the anomaly detection application can be achieved using the task level parallelization technique.

In speedup of the proposed method, 4.21 times speedup for the road tracking application, 5.80 times speedup for the vessel detection application and 4.10 times speedup for the anomaly detection application can be achieved using the multigrain parallelization technique on Intel Xeon X5670 with six processor cores. On ARM cortex-A15 with four processor cores, 3.38 times speedup for the road tracking application, 3.56 times speedup for the vessel detection application and 3.24 times speedup for the anomaly detection application can be achieved using the multigrain parallelization technique. Therefore, the proposed method attains 2.19 times speedup for the road tracking application, 2.19 times speedup for the vessel detection application and 1.64 times speedup for the anomaly detection application compared with the execution using the task level parallelization technique on Intel Xeon X5670 using six processor cores. On ARM Cortex-A15 with four processor cores, 1.75 times speedup for the road tracking application, 1.30 times speedup for the vessel detection application and 1.41 times speedup for the anomaly detection application compared with the execution using the task level parallelization technique.

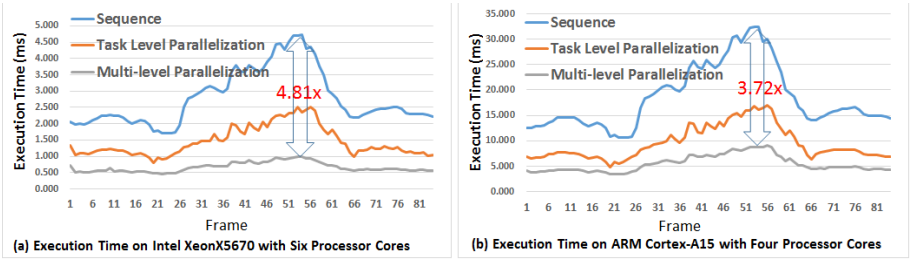

Fig. 7. Execution time per a frame for the road tracking application on Intel and ARM multi-cores

Further, this paper describes execution time per a frame for the road tracking application with a scatter per an input image. Fig.7-(a) and (b) shows execution time per a frame on Intel Xeon X5670 and ARM Cortex-A15 for the road tracking application. The upper lines in Fig.7-(a) and (b) show execution time of ordinary execution. Each of the execution fluctuates from 1.705 ms to 4.728 ms on Intel Xeon X5670 and from 10.58 ms to 32.36 ms on ARM Cortex-A15. The middle lines in Fig.7-(a) and (b) show execution time using the task level parallelization

technique. Each of the execution fluctuates from 0.815 ms to 2.510 ms on Intel Xeon X5670 with six processor cores and from 4.88 ms to 17.02 ms on ARM Cortex-A15 with four processor cores. The lower lines in Fig.7-(a) and (b) show the execution time using the multigrain parallelization technique. Each of the execution fluctuates from 0.459 ms to 0.983 ms on Intel Xeon X5670 with six processor cores and from 3.38 ms to 9.04 ms on ARM Cortex-A15 with four processor cores. Clearly, each variance of the execution time of the multigrain parallelized program is much lower than that of each sequential program on each processor. Therefore, the proposed method allows us to perform stable execution regardless of on input image. In worst case of sequential execution time on each processor, proposed method gives us 4.81 times speedup on Intel Xeon X5670 with six processor cores, and 3.72 times speedup on ARM Cortex-A15 with four processor cores using the multigrain parallelization technique compared with the sequential execution on each processor.

6 Conclusions

This paper has proposed the automatic multigrain parallelization scheme using the OSCAR compiler for embedded applications developed by MATLAB/ Simulink. This scheme exploits multigrain parallelism from auto-generated C code by Embedded Coder and optimizes this code. The proposed method includes three techniques of the inline expansion, the macro task fusion of conditional branches and the converting loop level parallelism into task level parallelism. The inline expansion is used to exploit more parallelism over hierarchies or nested levels. The macro task fusion is used to generate the statically scheduled parallel code without the runtime overhead. The converting loop level parallelism into task level parallelism is used to improve parallelism without the overhead of loop level parallelization. Additionally, the proposed method also includes the automatic profiling framework to improve performance of the statically scheduled parallel code.

Using the proposed method, this paper parallelized three important applications such as road tracking for self-driving cars, vessel detection for medical image recognition, and anomaly detection for pattern recognition really used an industry. In the performance evaluation, the OSCAR compiler with proposed method gave us 4.21 times speedup for the road tracking application, 5.80 times speedup for the vessel detection application and 4.10 times speedup for the anomaly detection application on Intel Xeon X5670 with six processor cores. Moreover, 3.38 times speedup for road tracking, 3.56 times speedup for the vessel detection application and 3.24 times speedup for the anomaly detection application on ARM Cortex-A15 with four processor cores. Comparing with the execution using the task level parallelization technique that is similar to the previous method for MATLAB/Simulink applications, the proposed method attained from 1.30 to 2.19 times speedup on different multi-cores such as Intel or ARM. The proposed method has successfully improved performance applications developed by MATLAB/Simulink on multi-core processors.

Acknowledgment

This work has been partly supported by A&D CO., LTD. for providing the anomaly detection model. We would like to express appreciation to A&D CO., LTD..

References

1. MATLAB/Simulink, http://www.mathworks.com/products/simulink/
2. Kamma, D. and Sasi, G. : Effect of Model Based Software Development on Productivity of Enhancement Tasks - An Industrial Study. Software Engineering Conference (2014)
3. Hanselmann, H., Kiffmeier, U., Köster, L., Meyer, M. and Rükgauer, A. : Production Quality Code Generation from Sirnulink Block Diagrams. Proceedings of the International Symposium on Computer Aided Control System Design (1999)
4. Gluch, D. and Kornecki, A. : Automated Code Generation for Safety Related Applications: A Case Study. Proceedings of the International Multiconference on Computer Science and Information Technology (2006)
5. Embedded Coder, http://www.mathworks.com/products/embedded-coder/
6. Parallel Computing Toolbox, http://www.mathworks.com/products/parallel-computing/
7. RTI-MP, http://www.dspace.com/en/inc/home/products/sw/impsw/rtimpblo.cfm
8. Canedo, A., Yoshizawa, T. and Komatsu, H. : Automatic Parallelization of Simulink Applications. Proceedings of the International Symposium on Code Generation and Optimization (2010)
9. Kumura, T., Nakamura, Y., ISHIURA, N., TAKEUCHI, Y. and IMAI, M. : Model Based Parallelization from the Simulink Models and Their Sequential C Code. SASIMI 2012
10. Obata, M., Kaminaga, H., Ishizaka, K. and Hironori, Kasahara. : Hierarchical Parallelism Control for Multigrain Parallel Processing. LCPC 2002
11. MEX Function, http://www.mathworks.com/help/matlab/apiref/mexfunction.html
12. Kasahara, H. : Parallel Processing Technology. CORONA PUBLISHING CO., LTD (1991)
13. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J. and Kasahara, H. : OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers. LCPC 2009
14. Shirako, J., Nagasawa, K., Ishizaka, K., Obata, M. and Kasahara, H. : Selective Inline Expansion for Improvement of Multi Grain Parallelism. The IASTED International Conference on PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS (2004)
15. Umeda, D., Kanehagi, Y., Mikami, H., Hayashi, A,. Kimura, K. and Kasahara, H. : Automatic Parallelization of Hand Written Automotive Engine Control Codes Using OSCAR Compiler. CPC 2013
16. Road Tracking, http://www.mathworks.com/help/vision/examples.html
17. Yen-Kuang C., Li, E., Jianguo L. and Tao, Wang. : Novel parallel Hough Transform on multi-core processors, Acoustics, Speech and Signal Processing (2008)
18. Bhadauria, H., Bisht, S. and Singh A. : Vessels Extraction from Retinal Images, IOSR Journal of Electronics and Communication Engineering (2013)