情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2014-SLDM-165 No.14
Vol.2014-EMB-32 No.14
2014/3/15

動的タスク切り替えを考慮した 組込みメニーコア SoC 向けタスクマッピング手法の拡張

甲斐田純也　谷口一徹　冨山宏之

立命館大学　〒525-8577　滋賀県草津市野路東 1 丁目 1-1

E-mail:　junya.kaida@tomiyama-lab.org, {i-tanigu, ht}@fc.ritsumei.ac.jp

本研究では、多数のプロセッサコアを搭載するメニーコア SoC を対象とした静的タスクマッピング手法を提案する。提案するタスクマッピング手法は、コア数の制約のもとで、タスクに対して適切な数のコアを割り当てる。本手法はタスク並列性とデータ並列性を考慮しており、メニーコア SoC に存在する多数のコアを効率よく使用することができる。先行研究では、１つのタスクをマッピングすることができるコアの組み合わせに制限を設けていた。本研究では既存手法を拡張しタスクをマッピングする方法に幅を持たせることによりマッピング解の改善を目指す。評価実験では、整数線形計画問題に基づく既存のタスクマッピング手法との比較を行い、拡張手法によって得られるマッピング解の質を評価する。

Extension of a Static Task Mapping Technique with Dynamic Task Switching for Embedded Many-core SoCs

Junya Kaida　Ittetsu Taniguchi　Hiroyuki Tomiyama

Ritsumeikan University, 1-1-1 Nojihigashi, Kusatsu-shi, Shiga, 525-8577 Japan

E-mail:　junya.kaida@tomiyama-lab.org, {i-tanigu, ht}@fc.ritsumei.ac.jp

This paper studies a static task mapping algorithm with dynamic task switching for embedded many-core SoCs. The task mapping technique proposed in this paper takes into account both inter-application and intra-application parallelisms in order to fully utilize the potential parallelism of the many-core architecture. We set a limit to the pair of cores that a single task can be mapped on in the previous research. In this research, we extend the previous technique to make the mapping pattern have variety and improve the mapping results. We evaluate the mapping results obtained by proposed technique by comparing them with those of existing technique which is based on ILP.

1. Introduction

Embedded System-on-Chip (SoC) architecture has shifted from single-core to multi-core paradigm because of improved power/performance efficiency, and it is now heading towards the many-core era. In order to fully utilize the high parallelism in the many-core architecture, mapping of application software onto cores is one of the important technologies. Especially in embedded SoCs, application mapping needs to take into account not only application-level parallelism (inter-application parallelism) but also data parallelism within applications (intra-application parallelism). One reason is that, unlike scientific applications, the amount of data parallelism inherent in individual embedded applications is limited. Another reason is that many embedded applications are inherently parallel.

This paper proposes extension of a static task mapping technique for homogeneous many-core SoCs proposed in [6]. The proposed technique considers both data and task parallelisms of applications, and maps tasks to the cores. The proposed technique maps tasks to the same cores if the tasks do not have to run in parallel. The tasks mapped onto the same cores are switched to each other at runtime. However, in [6], we set a limit to the pair of cores that a single task can be mapped on. In this paper, we make the mapping pattern have variety to improve the mapping results.

The rest of this paper is structured as follows. Related works are reviewed in Section 2. A task mapping technique with dynamic task switching, which is our previous work is explained in Section 3. Extension of the existing mapping technique is done in Section 4 and experimental results are shown in Section 5. Finally, Section 6 concludes this paper.

2. Related Work

Application mapping for multi/many-core architectures has been an important research topic for many years. Recent studies include [1] which proposes a heuristic algorithm for static task mapping on multi-core embedded systems. The work supports task mapping to hardware accelerators as well as CPU cores, but data parallelism is not considered. In other words, a task is assigned a single core. Techniques presented in [2][3][4][5] take into account data parallelism within tasks as well as task parallelism. Their methods take a task graph as input and perform task scheduling and mapping simultaneously, aiming at minimization of schedule length or pipeline throughput. Our work presented in this paper is similar to their works in a sense that we try to find the optimal number of cores for each task or application. However, our software model is different from theirs in that they take a task graph (i.e., a set of dependent tasks) of a single application as input and try to minimize the execution time of a single activation of the application or to maximize the pipeline throughput, while we target embedded systems where multiple applications run concurrently and repeatedly at different execution rates. The applications may be independent or dependent.

For application mapping for such embedded systems, we proposed static mapping techniques with dynamic task switching [6][7]. The dynamic task switching supports to switch the tasks mapped onto the same cores at runtime. In our previous techniques, tasks can be mapped on a limited number of combinations of the cores. In this paper, we relax the location restrictions of the mapping to aim for the improvement of the mapping result.

3. Previous Work: Static Task Mapping with Dynamic Task Switching

This section explains a static mapping technique with dynamic task switching, which is our previous research[6]. The technique determines, for each task, the number of cores onto which the task is mapped, considering both task and data parallelisms of individual tasks. The mapping technique supports dynamic task switching, and maps tasks to the same cores if the tasks do not have to run in parallel.

3.1 Many-core Architecture and Task Models

In this paper, we assume homogeneous many-core architectures with shared memory such as SMYLEref architecture presented in [8]. It is also assumed that the execution time of an application does not depend on the physical position of the application unless the application is assigned the same number of cores.

We assume embedded systems where multiple applications run in parallel. The applications are repeatedly executed at runtime in a cyclic way. Their execution can be periodic, aperiodic or sporadic, and their execution repetition rates may differ among applications. We implicitly assume that the applications are independent of each other. It is still possible to apply this work to dependent applications, but the obtained mapping results may not be optimal depending on how much the applications communicate with each other.

3.2 Problem Description

In this work, applications are mapped onto cores in a static way. Static mapping means that application mapping decision is made at a design time, and the applications never migrate over the cores at run time. This reduces the runtime overhead (in terms of performance and memory requirement) at the cost of lower CPU utilization compared with dynamic mapping. Also, our mapping supports dynamic task switching, which means the tasks mapped onto the same cores are switched to each other at runtime. Since some tasks may not be executed simultaneously, sharing of cores brings effective CPU utilization among these tasks. Therefore, proposed task mapping algorithm tries to utilize the cores to share with tasks exclusively executed.

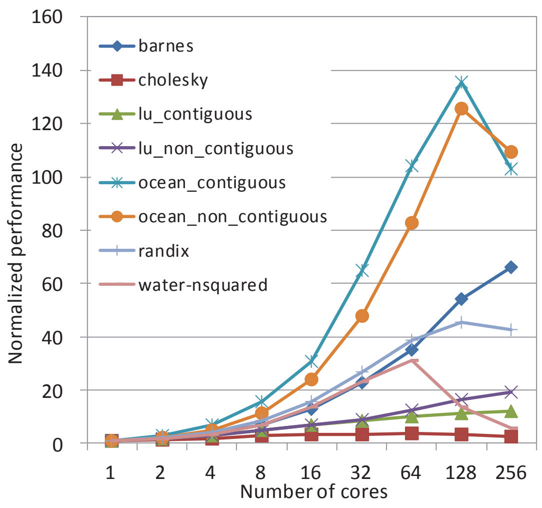

Figure 1. Normalized performance on different number of cores.

In general, the execution time and energy consumption of an application depend on the number of cores which the application uses. Figure 1 shows normalized performance of eight application programs from the SPLASH-2 benchmark suite executed on the Graphite cycle-accurate multi-core simulator [8]. For each program, we changed the number of cores from 1 to 256, and measured the number of execution cycles. The graph shows that eight programs feature different performance scalability curves. For example, the performance of `ocean_contiguous` scales up nicely until 128 cores, but it drops at the point of 256 cores. Barnes continuously scales up to 256 cores, but the performance improvement is relatively lower than `ocean_contiguous`. `Cholesky` does not scale up at all.

As we seen in Figure 1, different applications present different performance scalability curves, meaning that the optimal number of cores to be assigned depends on the application. In addition, we have to remind that the total number of cores is limited. For example, let us consider a scenario where we need to map barnes and randix onto a 64-core SoC. Of course, we cannot allocate 64 cores to both of the two applications because we have only 64 cores in total. In this case, assigning 32 cores to each application is a natural solution.

In the example above, we used the normalized performance as a metric for application mapping, but in practice we need to consider other factors such as energy consumption. Hereafter, for generality, let gain be a metric which indicates not only performance but also energy consumption and other important factors.

In order to describe mapping problem, we introduce a concept of tile. A tile is a set of cores on which a single task can be mapped. Figure 2 shows an SoC with four cores. Fore simplicity without loss of generality, we assume that each task may use one, two or four cores. In this case, the 4-core SoC has the following seven ways of tiling.

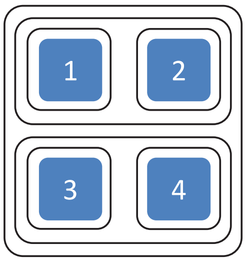

Figure 2. Tiles.

- Tile 1: Core 1
- Tile 2: Core 2
- Tile 3: Core 3
- Tile 4: Core 4
- Tile 5: Cores 1 and 2
- Tile 6: Cores 3 and 4
- Tile 7: Cores 1, 2, 3 and 4

We say that two tiles are overlapped if those tiles have at least one identical core. In case of Figure 2, Tiles 1 and 5 are overlapped, but Tiles 3 and 5 are not overlapped. Apparently, if two tasks need to run in parallel, the tiles of the two tasks must not be overlapped.

3.3 ILP Formulation of the Mapping Problem with Tile Constraints

In order to obtain optimal results of static task mapping with dynamic task switching, this session explain our previous research [6] based on ILP formulation.

Let $gain_{ij}$ be the gain of task i in case it is mapped to tile j. Then we introduce a variable map_{ij}: map_{ij} takes 1 if task i is mapped to tile j, otherwise 0. In order to describe feasibility of the mapping result, we introduce two symbols: $parallel_{i1i2}$ and $overlap_{j1j2}$. $parallel_{i1i2}$ indicates whether two tasks need to be executed in parallel or not. $parallel_{i1i2}$ takes 1 if tasks $i1$ and $i2$ need to be executed in parallel, and 0 otherwise. $overlap_{j1j2}$ indicates whether two tiles are overlapped or not. $overlap_{j1j2}$ takes 1 if tiles $j1$ and $j2$ are overlapped, and 0 otherwise. We assume that values of $gain_{ij}$, $parallel_{i1i2}$ and $overlap_{j1j2}$ are given.

Then, the task mapping problem is formulated as objective function (1) under the constraints (2)-(3).

maximize:
$$\sum_i \sum_j map_{ij} \times gain_{ij} \tag{1}$$

subject to:
$$\forall i, \sum_j map_{ij} = 1 \tag{2}$$

$$\forall (i1, i2, j1, j2), parallel_{i1i2} = 1 \rightarrow$$
$$map_{i1j1} \times overlap_{j1j2} + map_{i2j2} \times overlap_{j1j2} \leq 1 \tag{3}$$

Formula (2) expresses that if the task $i1$ and task $i2$ need to run in parallel, two tiles, i.e., tile $j1$ on which task *i1* is mapped and tile $j2$ on which task *i2* is mapped, cannot be overlapped. If both $parallel_{i1i2}$ and $overlap_{j1j2}$ are 1, either map_{i1j1} or map_{i2j2} should be 0. Formula (3) expresses that if the task $i1$ and task $i2$ need to run in parallel, two tiles, i.e., tile $j1$ on which task $i1$ is mapped and tile $j2$ on which task $i2$ is mapped, cannot be overlapped. If both $parallel_{i1i2}$ and $overlap_{j1j2}$ are 1, either map_{i1j1} or map_{i2j2} should be 0.

4. Extension of Mapping with Dynamic Task Switching

In the previous research, we supposed that a single task is mapped on the tiles prepared on the SoC. In other words, the way of mapping for a single task was limited to neighboring 2^n cores. In this paper, we extend the task mapping technique proposed in the previous research to make the mapping pattern have variety and improve the mapping results. The task mapping problem dealt in this paper only considers a constraint of the number of cores and remove the location restrictions, while previous research considered the both constraints.

4.1 Relaxing Location Restrictions

In this section, we remove the restrictions on a position of the cores that single task can be mapped onto to find out how the location restrictions have an impact on the mapping results. We supposed that single task should be mapped to any tile (neighboring 2^n cores) in the previous research. However, in this paper, we suppose that the tasks can use the cores in any combination. For example, the way of using two cores for a single task on SoC of 16 cores was limited to eight patterns as shown in Figure 3. In this paper, we do not put location limitation on the set of cores. Therefore, the way of using 2 cores on this SoC is ${}_{16}C_2 = 120$.

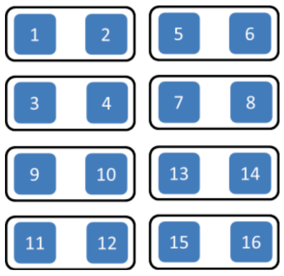

Figure 3. The way to use two cores on SoC of 16 cores

4.2 ILP Formulation of Tile Free Mapping Problem

The problem is formally defined as an ILP problem. By solving the ILP problem, optimal task mapping is obtained.

$gain_{ik}$ denotes the gain of task i in case it is mapped with core usage pattern k. Core usage pattern k is the index of the number of cores the task uses. For example, if a single task is allow to use 2^n cores from 1 to 256 cores, total number of core usage patterns are nine (1, 2, 4, 8, 16, 32, 64, 128 and 256 cores). In this example, $gain_{14}$ indicates the gain value when task 1 is executed on eight cores. map_{ij} is 1 if task i is mapped to core j, and 0 otherwise. $parallel_{i1i2}$ indicates whether two tasks need to be executed in parallel or not, similar to task mapping problem in the previous work. In addition, we define three symbols, O_{ik}, $MapPattern_k$ and $Core_i$. O_{ik} is 1 if task i is mapped with core usage pattern k, otherwise 0. $MapPattern_k$ indicates the number of cores of the usage pattern k. In the example above, $MapPattern_4$ is eight. $Core_i$ is the sum of the number of cores used by task i. We assume that values of $gain_{ij}$, $MapPattern_k$ are given.

Then, the task mapping problem is formulated as objective function (4) under the constraints (5)-(9).

maximize:
$$\sum_i \sum_k O_{ik} \times gain_{ik} \tag{4}$$

Subject to:
$$\forall i \sum_k O_{ik} \times MapPattern_k = Core_i \tag{5}$$

$$\forall i \sum_j map_{ij} = Core_i \tag{6}$$

$$\forall i \sum_k O_{ik} = 1 \tag{7}$$

$$\forall i \;\; Core_i \geq 1 \tag{8}$$

$$\forall i,j \;\; parallel_{i1i2} = 1$$
$$\rightarrow \quad map_{i1j} + map_{i2j} \leq 1 \tag{9}$$

Objective function (4) expresses the total gain of the tasks and is the objective function to be maximized. Formulas (5) and (6) defines the valuable $Core_i$ which indicates the sum of the number of cores used by task i. Formula (7) says that each task should be mapped by using any core usage pattern. Formula (8) says that each task should be mapped on at least one core. Formula (9) expresses that if the task $i1$ and task $i2$ need to run in parallel, those two tasks cannot use the same core on the SoC. If $parallel_{i1i2}$ is 1, either map_{i1j} or map_{i2j} should be 0.

5. Experiments

Task mapping results are obtained by solving the ILP problem with an ILP solver. We compared the proposed technique with the ILP based technique explained in Section 3. The ILP problem was solved with IBM CPLEX12.5 [10], and all experiment was performed on Intel Xeon 2.0GHz and 128GB memory machine. We performed three experiments changing the core usage pattern of a single task. In the first experiment, a single task is allowed to use 2^n cores ranging from 1 to 256 cores. In the second experiment, we add some new core usage patterns for a single task. In the third experiment, we allow the tasks to use from 1 to 256 cores at an interval of 1 core. We performed the experiment on 2^n cores ranging from the number of tasks to 1024 cores. We prepared three sets of application programs based on the SPLASH-2 benchmark suite. Task sets 1, 2, and 3 include 8, 4, and 16 tasks (application programs), respectively. Task set 1 includes eight application programs shown in Figure 1, and task set 2 includes `lu_non_contiguous`, `ocean_contiguous`, `ocean_non_contiguous`, and `water-nsquared` from these eight application programs. Task set 3 includes the same two programs for each application program. Values of the two-dimensional matrix $parallel_{i1i2}$ are randomly decided based on density d, which indicates the percentage of value 1. d=100% indicates $parallel_{i1i2}$=1 for any two tasks, meaning that all tasks should run in parallel. On the other hand, d=0% indicates that all cores are available to every task.

5.1 Experiment to Investigate the Effects of Tile Constraints

In this experiment, a single task will be mapped on 2^n cores ranging from 1 to 256 cores. Therefore, the number of core usage patterns for a single task will be nine (1, 2, 4, 8, 16, 32, 64, 128 and 256 cores). Task mapping results are compared with the technique we proposed in previous work to see how the tile limitation has an impact on task mapping results.

We varied the density d from 0% to 100%, and the number of cores. Figures 4-8 show the comparison of total gain to the previous work. We normalized the total gain value and the baseline is the total gain value obtained by task mapping technique proposed in previous work [6]. Each figure shows the comparison for different d.

As shown in Figures 4-8, the total gain barely changed in all of the cases and the improvement was only around 1-6%. Similar tendency were shown in the results of task set 3. The results of task set 2 did not change at all. It can be said that increasing the number of tiles with different shapes does not affect the mapping result a lot when the core usage patterns for a single task are not prepared enough.

5.2 Increase the Core Usage Pattern

In the experiment so far, a single task can be mapped on 2^n cores from 1 to 256 and the total number of core usage patterns for a single task were nine (1, 2, 4, 8, 16, 32, 64, 128 and 256 cores). In this experiment, we add some new core usage patterns whose number of cores in use is intermediate value of existing patterns. Therefore, a single core can be mapped with 16 different patterns (1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192 and 256 cores).

We compared the mapping results with the results obtained by ILP based technique we proposed in the previous research in similar to experience 5.1. When we look at Figure 4-8, we can see larger improvement in the mapping results compared to that of previous experiment. Similar tendency are shown in the results of task sets 2 and 3. It can be said that preparing the variety of core usage patterns is effective to improve the efficiency of the whole system.

5.3 Let Task Use Any Number of Cores

In this experiment, we let the tasks to use any number of cores from 1 to 256. Therefore, the core usage patterns for a single task on this mapping problem are 256. From the results of experiments in section 5.1 and 5.2, the larger improvement can be expected. The value of the gain correspond to each number of cores are obtained by using statistical analysis software "R" [11]. We applied the spline interpolation to each task set to interpolate the gain data.

As shown in Figure 4-8, the mapping results improved larger than experiment in section 5.2 and some result shows improvement of about 40%. However, in some cases, results are smaller than those of experiment 5.2. This happens because of the error of the interpolation of the gain value.

The core usage of the mapping result which showed the large improvement (Task set 1, d=25%, 64 cores on SoC) is shown in Table 1. We compare the core usage of experiment in section 5.1 and 5.3. Each value expresses the number of cores the task uses in the mapping obtained with each experiment.

When we look at Table 1, we can see that the core usage is quite different between experience 5.1 and experience 5.3. We can see that core usage of each task in experience 5.1 is similar to each other, while the way of using cores varies in the result of experience 5.3. We found that tasks can use suitable amount of cores by preparing various patterns of core usage.

Table 1. Comparison of core usage of two techniques

Task	Experiment A	Experiment C
`water-nsquared`	32cores	33cores
`randix`	32cores	9cores
`ocean_non_contiguous`	32cores	63cores
`ocean_contiguous`	32cores	55cores
`lu_non_contiguous`	32cores	63cores
`lu_contiguous`	16cores	32cores
`cholesky`	16cores	1core
`barnes`	32cores	31cores

6. Conclusions

In this paper, we have proposed static task mapping technique with dynamic task switching for embedded many-core SoCs. We extended the existing task mapping technique to make the mapping pattern have variety. The proposed technique takes into account both inter-application and intra-application parallelisms in order to efficiently utilize the potential parallelism of the many-core architecture. Experimental results show that preparing various task usage patterns for each task is effective to improve the efficiency of the whole system.

At present, this work does not assume that applications have deadline constraints. In future, we will take into account deadline constraints of individual applications.

Acknowledgment

The authors would like to thank Professor Hiroshi Sasaki for his supports to conduct the experiments. The authors would also like to thank Professor Koji Inoue of Kyushu University and Associate Professor Yuko Hara-Azumi of Tokyo Institute of Technology for their support for this work. This work was in part supported by NEDO.

References

[1] Y. Ando, S. Shibata, S. Honda, H. Tomiyama, and H. Takada, "Fast design space exploration for mixed hardware-software embedded systems," *International SoC Design Conference (ISOCC)*, 2011.

[2] S. Ramaswamy, S. Sapatnekar and P. Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1098-1116, Nov. 1997.

[3] H. Yang and S. Ha, "ILP based data parallel multi-task mapping/scheduling technique for MPSoC," *International SoC Design Conference (ISOCC)*, 2008.

[4] H. Yang and S. Ha, "Pipelined data parallel task mapping/scheduling technique for MPSoC," *Design Automation and Test in Europe (DATE)*, 2009.

[5] N. Vydyanathan, et al., "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," *IEEE Trans. on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1158-1172, Aug. 2009.

[6] J. Kaida, T. Hieda, I. Taniguchi, H. Tomiyama, Y. Hara-Azumi, and K. Inoue, "Task Mapping Techniques for Embedded Many-core SoCs," *International SoC Design Conference (ISOCC)*, 2012.

[7] J. Kaida, I. Taniguchi, T. Hieda, and H. Tomiyama, "A Static Task Mapping Algorithm with Dynamic Task Switching for Embedded Many-core SoCs," *International Symposium on Communications and Information Technologies(ISCIT)*, 2013.

[8] M. Kondo, et al., "SMYLEref: A reference architecture for manycore-processor SoCs," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013.

[9] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," *High Performance Computer Architecture (HPCA)*, 2010.

[10] IBM CPLEX Optimizer, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/. [Access: 2014/02/10].

[11] The R Project for Statistical Computing http://www.r-project.org/. [Access: 2014/02/10].

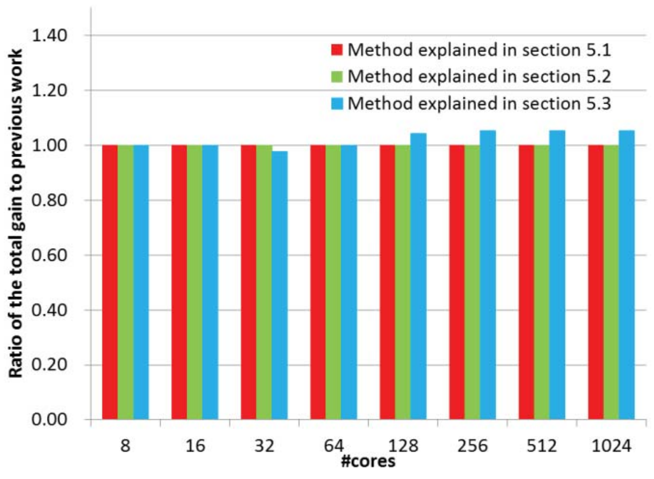

Figure 4. Comparison of gain (*d*=0%)

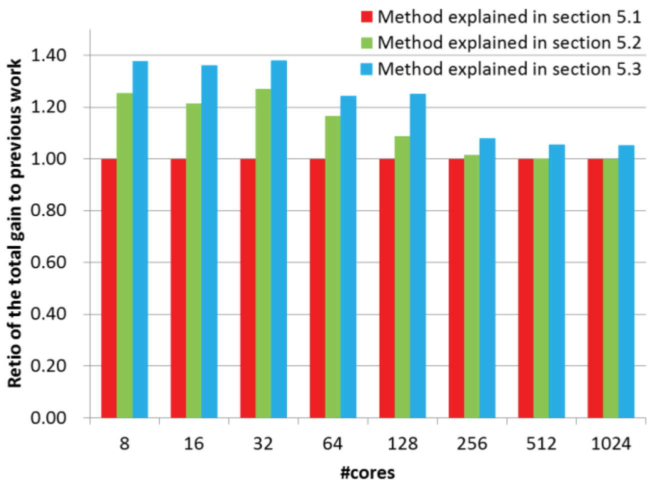

Figure 5. Comparison of gain (*d*=25%)

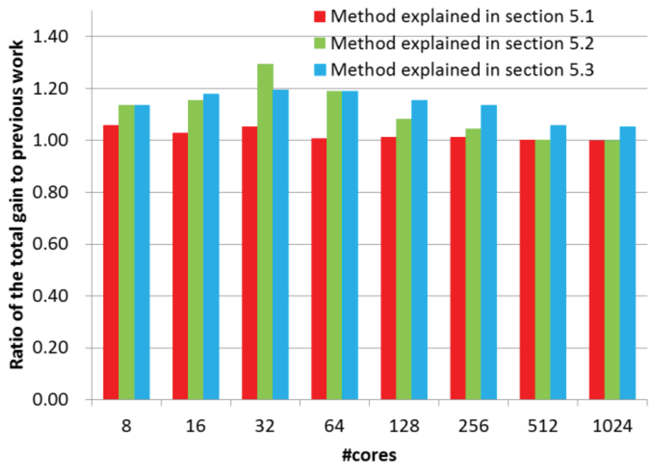

Figure 6. Comparison of gain (*d*=50%)

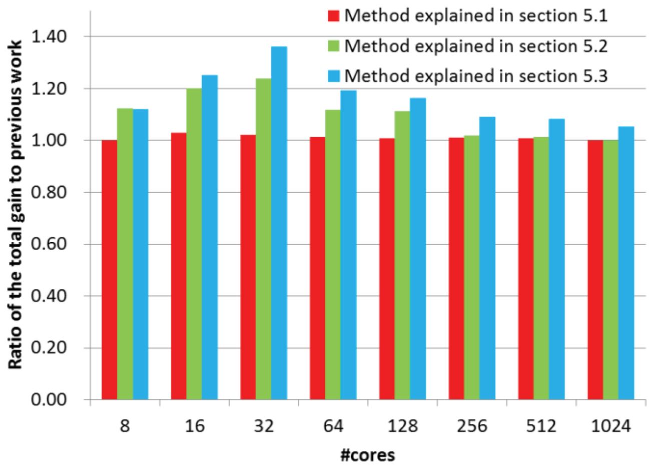

Figure 7. Comparison of gain (*d*=75%)

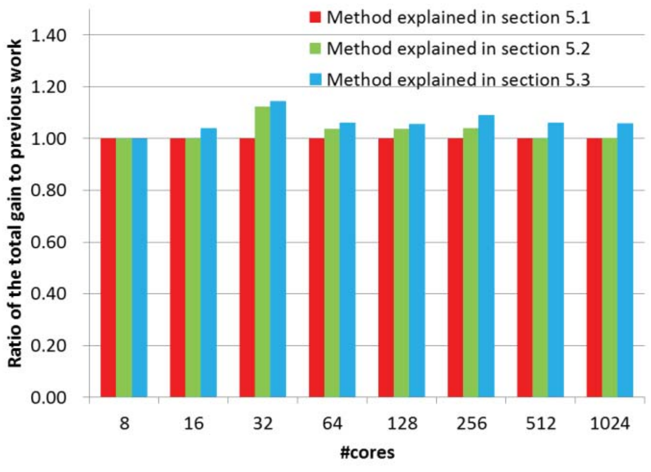

Figure 8. Comparison of gain (*d*=100%)