IPSJ Transactions on System LSI Design Methodology Vol.12 74–77 (Aug. 2019)

[DOI: 10.2197/ipsjtsldm.12.74]

Short Paper

A Genetic Algorithm for Scheduling of Data-parallel Tasks on Multicore Architectures

Yang Liu[1,a)] Lin Meng[1,b)] Hiroyuki Tomiyama[1,c)]

Received: November 29, 2018, Revised: March 8, 2019,
Accepted: April 22, 2019

Abstract: This paper proposes a genetic algorithm for scheduling of multiple data-parallel tasks on multicores. Unlike traditional task scheduling, this work allows individual tasks to run on multiple cores in a data-parallel fashion. Experimental results show the effectiveness of the proposed algorithm over state-of-the-art algorithms.

Keywords: task scheduling, multicore architecture, task parallelism, data parallelism, genetic algorithm

1. Introduction

This paper addresses task scheduling on multicore architectures, with a goal of minimum schedule length under constraints on inter-task dependency and the number of cores. In general, task scheduling is an NP-hard problem, and finding exact solutions is proven to be very complex and consumes a large amount of memory and computing resources [1], [2], [3]. Therefore, many heuristic approaches to task scheduling have been proposed [4], [5], [6], [7]. Recently, genetic algorithms (GAs) have been widely studied as useful methods for obtaining high-quality solutions for task scheduling problems [8], [9], [10], [11], [12]. Unfortunately, previous works on GA-based task scheduling consider task parallelism only. Many studies such as Refs. [13], [14], [15], [16], [17], [18], [19] and [20] have shown that, for a large class of scientific and multimedia applications, exploiting both task and data parallelisms yields better speedups compared to either pure task parallelism or pure data parallelism. In Ref. [13], the authors developed a language, compiler and runtime system for task- and data-parallel systems. The authors of Refs. [14] and [15] studied scheduling of data-parallel tasks where the degree of data parallelism for each task is flexible. In Refs. [16], [17], [18], [19] and [20], on the other hand, the authors studied scheduling of data-parallel tasks where the degree of data parallelism for each task is fixed prior to scheduling. The authors of Refs. [16], [17] and [18] proposed heuristic algorithms based on list scheduling, while the work in Ref. [19] proposed an exact banch-and-bound algorithm. In Ref. [20], the authors presented an integer linear programming approach to scheduling of data-parallel tasks which considers communication overheads among dependent tasks.

This paper proposes a genetic algorithm for task scheduling with both task and data parallelisms where the degree of data parallelism is fixed and communication overheads are not taken into account. Specifically, we propose a novel chromosome representation for task scheduling problem and efficient genetic operators (i.e., selection, crossover and mutation) for the chromosome representation.

[1] Graduate School of Science and Engineering, Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[a)] yang.liu@tomiyama-lab.org
[b)] menglin@fc.ritsumei.ac.jp
[c)] ht@fc.ritsumei.ac.jp

2. Problem Description

The task scheduling problem addressed in this paper is the same as the one in Refs. [17], [18] and [19]. This paper assumes homogeneous multicore architectures. An application is modeled as a directed acyclic graph, so called a task graph, where a node represents a task and a directed edge represents a flow dependency between two tasks. **Figure 1** shows an example of a task graph and its optimal schedule on four cores. Tasks labeled "S" and "E" are dummy tasks, denoting an entry point and an exit point of the application, respectively. Two integer values are associated with each task. The first number denotes the degree of data parallelism of the task. In other words, the number denotes the number of cores which are necessary to run the task. We assume that the degree of data parallelism is decided by programmers, and how to decide it is out of the scope of this paper. The latter number on each node denotes the execution time of the task. For example, task 1 runs on 4 cores, and it takes 10 time units to complete its execution.

Given a task graph, task scheduling decides when and on which cores each task is executed in such a way that the overall schedule length (a.k.a. makespan) is minimized while meeting constraints on inter-task dependency and the number of available cores.

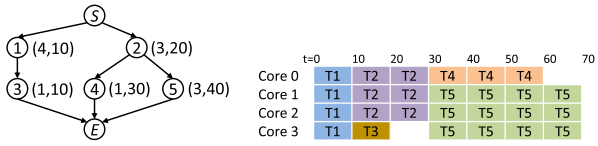

Fig. 1 A task graph with data parallelism (left) and its optimal schedule on four cores (right).

3. The Proposed Genetic Algorithm

3.1 An Overall Procedure

Genetic algorithms are a kind of meta-heuristic algorithms inspired by the processes observed in natural selection [21]. Our genetic algorithm is based on a standard procedure as follows:

(1) Initialization: Generate initial population.
(2) Calculation of the fitness: Calculate the fitness value for each individual.
(3) Selection: Select individuals as parents for the next generation.
(4) Crossover: Vary the programming of a chromosome (or chromosomes) from one generation to the next generation.
(5) Mutation: Alter genes in chromosomes.
(6) Termination: Go back to step (2) until a certain criteria is reached.

3.2 Representation of a Chromosome

A *chromosome* is a set of strings, and represents a potential solution (also called as an *individual*) for the problem. Adequate definition of the chromosome is one of the most important issues in genetic algorithms. In our genetic algorithm, a chromosome is defined as an array of N elements where N represents the number of tasks. This array determines the sequence of task execution. **Figure 2** shows an example of our chromosome for the task graph in Fig. 1. The chromosome indicates that task 1 (depicted as T1 in the figure) is scheduled first, task 2 is the next, and so on.

A chromosome is called *valid* if the scheduling solution represented by the chromosome satisfies the precedence dependencies among the tasks. The chromosome in Fig. 2 is valid since T1 is scheduled before T3, and T2 is scheduled before T4 and T5.

3.3 Initialization

Our algorithm begins with generating a set of chromosomes. A chromosome is generated by randomly ordering the tasks from the leftmost gene to the right, still meeting precedence dependencies. In case of the task graph in Fig. 1, the leftmost gene is randomly selected from T1 or T2. If T2 is selected as the first gene, the second gene is randomly selected from T1, T4 or T5.

3.4 Fitness Function and Selection

The fitness function defines the quality of the chromosome. In our genetic algorithm, the fitness function returns the shortest length of the schedule represented by the chromosome. Briefly speaking, our fitness function runs list-based scheduling with the priority represented by the chromosome.

Our genetic algorithm employs a classic selection technique based on roulette wheel. On a virtual roulette wheel, our algorithm assigns each chromosome a segment of a size proportional to its fitness. Hence, chromosomes with higher fitness values are more likely to be selected for the next generation.

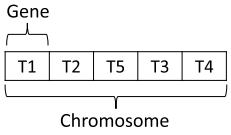

Fig. 2 A chromosome example.

3.5 Crossover and Mutation

In genetic algorithms, crossover creates a new chromosome by exchanging part of genes between two chromosomes, and mutation randomly alters genes in a chromosome. **Figure 3** illustrates how our crossover and mutation operations are performed in order to generate valid chromosomes.

Our crossover operation first randomly selects two chromosomes A and B from the population, next randomly selects a crossover point in chromosome A, then copies the left part of chromosome A to child chromosome C, and finally copies the remaining genes from chromosome B to chromosome C.

Our mutation operation first randomly selects a chromosome from the population, next randomly selects a gene (i.e., T4 in Fig. 3), and then randomly places the gene between its immediate predecessor(s) and successor(s). The task graph in Fig. 1 shows that the predecessor of T4 is T2, and T4 has no successor. Therefore, T4 is placed randomely but after T2.

3.6 Termination

Our genetic algorithm stops when the number of generations reaches a user-specified number.

4. Experiments

The proposed genetic algorithm was implemented in C++, and was evaluated against three state-of-the-art algorithms. One is an exact branch-and-bound algorithm [19], and the other two are heuristic ones based on list scheduling, i.e., the PCS algorithm [17] and the dual-mode algorithm [18]. Since the branch-and-bound algorithm is computationally expensive, we limited the runtime of the algorithm up to 12 hours, and the best solution found by that time was used for our evaluation. As benchmark programs, 20 task graphs with 50 tasks each were selected from Standard Task Graph (STG) [22] [*1]. The experiments were conducted on Intel Core i7-4790K with 32 GB memory. In our genetic algorithm, the population size was set to 16,384, and the number of generations was limited to 50.

Figures 4 and **5** show the quality of results (i.e., normalized schedule length) on four cores and eight cores, respectively. The results clearly demonstrate the effectiveness of our genetic algorithm over the heuristic algorithms.

The runtime of the four scheduling algorithms are compared in **Table 1**. The runtime of the branch-and-bound algorithm significantly depends on the task graph. The PCS and dual-mode algorithms ran in the order of milliseconds, while our genetic al-

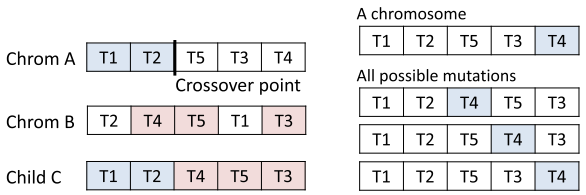

Fig. 3 Examples of crossover (left) and mutation (right).

[*1] Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks.

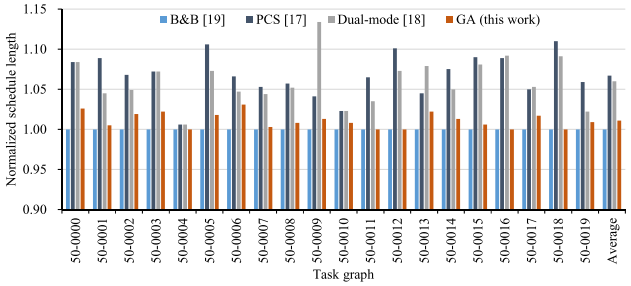

Fig. 4 Comparison of four algorithms on four cores.

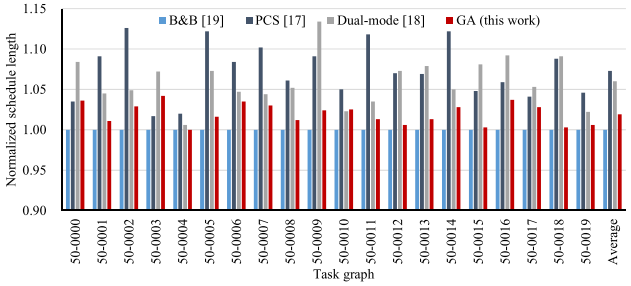

Fig. 5 Comparison of four algorithms on eight cores.

Table 1 Runtimes of the scheduling algorithms (seconds).

	Four cores	Eight cores
B&B [19]	0.89–43,200 (suspended)	0.93–43,200 (suspended)
PCS [17]	< 0.01	< 0.01
Dual-mode [18]	< 0.01	< 0.01
GA (this work)	3.81–4.37	4.21–4.84

gorithm ran in the order of seconds.

The results in Fig. 4, Fig. 5 and Table 1 clearly show the trade-off between the quality of results and the algorithm runtime. In principle, the branch-and-bound algorithm broadly explores the solution space to find a truely optimal solution. However, due to its exponential complexity of computation, it sometimes takes an unacceptably long time to find the solution. The PCS and dual-mode algorithms are based on list scheduling. Once they find a solution in a greedy manner, they stop without trying to find better solutions. Therefore, they run fast, but the quality of results is not high. The genetic algorithm iteratively explores the solution space by repeating selection, crossover and mutation operations. With our parameter settings in the experiments, the genetic algorithm searched approximately 800,000 valid solutions for each task graph. This is the main reason why the genetic algorithm outperforms the PCS and dual-mode algorithms which search only one solution.

5. Conclusions

In this paper, we proposed a genetic algorithm for the task scheduling problem which takes into account both task parallelism and data parallelism. Our experiments using a set of standard task sets show that the proposed genetic algorithm efficiently finds near-optimal schedules in a short runtime. In future, we plan to extend our scheduling algorithm so that inter-task communication and resource conflicts are taken into account.

Acknowledgments This work is in part supported by KAKENHI 15H02680.

References

[1] Kasahara, H. and Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing, *IEEE Trans. Computers*, Vol.C-33, No,11, pp.1023–1029 (1984).
[2] Fujita, S.: A branch-and-bound algorithm for solving the multiprocessor scheduling problem with improved lower bounding techniques, *IEEE Trans. Computers*, Vol.60, No.7, pp.1006–1016 (2011).
[3] Sinnen, O., Kozlov, A.V. and Shahul, A.Z.S.: Optimal scheduling of task graphs on parallel systems, *Proc. 9th International Conference on Parallel and Distributed Computing, Applications and Technologies* (*PDCAT*), pp.323–328 (2008).
[4] Kwok, Y.K. and Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors, *ACM Computing Surveys* (*CSUR*), Vol.31, No.6, pp.406–471 (1999).
[5] Hwang, J.J., Chow, Y.C., Anger, F.D. and Lee, C.Y.: Scheduling precedence graph in systems with interprocessor communication times, *SIAM Journal of Computing*, Vol.18, No.2, pp.244–257 (1989).
[6] Sih, G.C. and Lee, E.A.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architecture, *IEEE Trans. Parallel and Distributed Systems*, Vol.4, No.2, pp.175–187 (1993).
[7] Hagras, T. and Janecek, J.: A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environment, *Proc. 2013 International Conference on Parallel Processing Workshops*, pp.149–155 (2003).
[8] Page, A.J. and Naughton, T.J.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing, *Proc. 19th International Parallel and Distributed Processing Symposium*, p.189a (2005).
[9] Hou, E.S.H., Ansari, N. and Ren, H.: A genetic algorithm for multiprocessor scheduling, *IEEE Trans. Parallel and Distributed Systems*, Vol.5, No.2, pp.113–120 (1994).
[10] Omara, F.A. and Arafa, M.M.: Genetic algorithms for task scheduling problem, *Journal of Parallel and Distributed Computing*, Vol.70, No.1, pp.13–22 (2010).
[11] Roy, P., Alam, M.M. and Das, N.: Heuristic based task scheduling in multiprocessor systems with genetic algorithm by choosing the eligible processor, *International Journal of Distributed and Parallel Systems* (*IJDPS*), Vol.3, No.4, pp.111–121 (2012).
[12] Entezari-Maleki, R. and Movaghar, A.: A genetic-based scheduling algorithm to minimize the makespan of the grid applications, *Proc. International Conferences on Grid and Distributed Computing* (*GDC*) *and Control and Automation* (*CA*), pp.22–31 (2010).
[13] Hassen, S.B., Bal, H.E. and Jacobs, C.J.H.: A task- and data-parallel programming language based on shared objects, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.20, No.6, pp.1131–1170 (1998).
[14] Radulescu, A., Nicolescu, C., van Gemund, A.J.C. and Jonker, P.P.: CPR: Mixed task and data parallel scheduling for distributed systems, *Proc. 15th International Parallel and Distributed Processing Symposium* (*IPDPS*) (2000).
[15] Shimada, K., Kitano, S., Taniguchi, I. and Tomiyama, H.: ILP-based scheduling for parallelizable tasks, *IEICE Trans. Fundamentals*, Vol.E100-A, No.7, pp.1503–1505 (2017).
[16] Ramaswamy, S., Sapatnekar, S. and Banerjee, P.: 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 (1997).
[17] Liu, Y., Meng, L., Taniguchi, I. and Tomiyama, H.: Novel list scheduling strategies for data parallelism task graphs, *International Journal on Networking and Computing*, Vol.4, No.2, pp.279–290 (2014).
[18] Liu, Y., Meng, L., Taniguchi, I. and Tomiyama, H.: A dual-mode scheduling approach for task graphs with data parallelism, *International Journal of Embedded Systems*, Vol.9, No.2, pp.147–156 (2017).
[19] Liu, Y., Meng, L., Taniguchi, I. and Tomiyama, H.: A branch-and-bound algorithm for scheduling of data-parallel tasks, *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies* (*SASIMI*), pp.96–100 (2016).
[20] Shimada, K., Taniguchi, I. and Tomiyama, H.: Communication-aware scheduling of data-parallel tasks on multicore architectures, *IPSJ Trans. System LSI Design Methodology*, Vol.12, pp.65–73 (2019).
[21] Holland, J.H.: Genetic algorithms, *Scientific American*, Vol.267, No.1, pp.66–73 (1992).
[22] Tobita, T. and Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms, *Journal of Scheduling*, Vol.5, No.5, pp.379–394 (2002).

Yang Liu received his M.E. and Ph.D. degrees in science and engineering from Ritsumeikan University in 2015 and 2018, respectively. He currently works for Panasonic Advanced Technology Development Co., Ltd. His research interests include multicore task scheduling, embedded systems, and simultaneous localization and mapping.

Lin Meng received his B.S., M.S., and Ph.D. degrees in science and engineering from Ritsumeikan University in 2006, 2008 and 2012, respectively. He was a Research Associate with the Department of Electronic and Computer Engineering, Ritsumeikan University from 2011 to 2013, where he was also an Assistant Professor from 2013 to 2018, and a Lecturer from 2018 to 2019. From 2015 to 2016, he was a Visiting Scholar with the Department of Computer Science and Engineering, University of Minnesota at Twin Cities. In 2019, he became an Associate Professor with the Department of Electronic and Computer Engineering, Ritsumeikan University. His research interests include computer architecture, parallel processing and deep learning based image recognition. He is a member of IEEE, IEICE and IPSJ.

Hiroyuki Tomiyama received his B.E., M.E. and D.E. degrees in computer science from Kyushu University in 1994, 1996 and 1999, respectively. He worked as a visiting researcher at UC Irvine, as a researcher at ISIT/Kyushu, and as an associate professor at Nagoya University. Since 2010, he has been a full professor with College of Science and Engineering, Ritsumeikan University. He has served on program and organizing committees for a number of premier conferences including DAC, ICCAD, DATE, ASP-DAC, CODES+ISSS, CASES, ISLPED, RTCSA, FPL and MPSoC. He has also served as editor-in-chief for IPSJ TSLDM, as an associate editor for ACM TODAES, IEEE ESL and Springer DAEM, and as chair for IEEE CS Kansai Chapter and IEEE CEDA Japan Chapter. His research interests include, but not limited to, design methodologies for embedded and cyber-physical systems. He is a member of ACM, IEEE, IEICE and IPSJ.

(Recommended by Associate Editor: *Yasuhiro Takashima*)