Genetic algorithm for Scheduling of Data-Parallel Tasks

Yang Liu, Hiroyuki Tomiyama

Department of Electronic and Computer Engineering Ritsumeikan University

1-1-1 Noji-Higashi, Kusatsu, Shiga 525-8577, Japan



(a) A task graph



(b) An optimal schedule

Figure 1. A scheduling example

**Abstract — This paper studies a task scheduling problem which schedules a set of data-parallel tasks on multiple cores. Unlike most of previous literature where each task is assumed to run on a single core, this work allows individual tasks to run on multiple cores in a data-parallel fashion. Since the scheduling problem is NP-hard, that is, to obtain the optimal solutions require search of the overall solution space, which is very time consuming. Therefore, heuristic methods are reasonable choose to obtain suboptimal results. In this paper, an efficient method based on genetic algorithm is proposed to solve this problem. Different with traditional Genetic algorithms for task scheduling, we proposed a new representation for the chromosome of tasks scheduling and corresponding genetic operators, aim to reduce the search space and improve the computing speed. In addition to normal implementation, we also implemented our algorithm with OpenMP, show how to speeded up our algorithm further.**

**Keywords — task scheduling; multicore; data parallelism; branch-and-bound Genetic algorithm OpenMP**

# Introduction

The problem of task scheduling can be simply described as scheduling a set of tasks onto a multiprocessor system, find the minimum scheduling length under the given constraint conditions. Due to the wide deployment of multicore architecture not only in general-purpose processors but also in embedded processors, this problem has now become a more important problem than ever.

In general, task scheduling is considered a NP-hard problem. The effect of finding exact results is proved very complex, and consuming a large amount of memory and computing resources.

Therefore, Many heuristic approaches for task scheduling have been proposed（随便举例）. Recently, Genetic Algorithms (GAs) have been widely studied as useful heuristics for obtaining high quality solutions for task scheduling problem （随便举例）.

Unfortunately, Majority of the works deal task scheduling with genetic algorithms only considered about task parallelism. Many studies has shown that, for a large class of large computational applications, exploiting both task and data parallelism yields better speedups compared to either pure task parallelism or pure data parallelism.

This paper presents an approach for task scheduling based on genetic algorithm, to solve the scheduling problem with task and data parallelism. Not only with different problem define, we also proposed a new representation for the chromosome of this problem. Our chromosome only encode information about the ordering of task execution, ignore those tasks are mapped on which cores. It is reduces greatly the size of search space and improves the performance of algorithm. The efficient genetic operators (select, crossover, mutation) corresponding to this chromosome also are presented. In additional, we implemented our algorithm with OpenMP; show how to speed-up our algorithm further.

The contributions of this paper are as follows:

* This paper presents an approach for task scheduling based on genetic algorithm, to solve the scheduling problem with task and data parallelism
* This paper proposed a new chromosome representation to scheduling problem and corresponding crossover and mutation strategies are used to minimize the makespan.
* The implementation with OpenMP for accelerating our proposed algorithm is discussed.

This paper is organized as follows. Section II formally describes a scheduling problem addressed in this paper, and Section III proposes a scheduling algorithm. Experiments are presented in Section IV.

# Problem Definition

This section defines a task scheduling problem addressed in this paper.

## Problem Description

This work assumes homogeneous multicore processors. An application is modeled as an acyclic directed graph (DAG), so called a task graph, where a node represents a task and a directed edge represents a flow dependency between two tasks. Figure 1 (a) shows an example of a task graph. In this graph, tasks labeled “*S*” and “*E*” are dummy tasks which do not perform any meaningful computation. Tasks *S* and *E* denote 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 scope of this paper. The latter number on each node denotes the execution time of the task. For example, task 1 runs on 3 cores, and it takes 30 time units to complete the task.



Figure 2. A branching tree

Given a task graph, task scheduling decides when and on which core each task is executed in such a way that the overall schedule length is minimized, while meeting constraints on flow dependency among tasks and the number of available cores. Figure 1 (b) shows one of optimal schedules on four cores for the task graph in Figure 1 (a).

# GENETIC ALGORITHM

Genetic algorithm was first invented by Holland [13], it is a meta-heuristic inspired by the processes observed in natural evolution. Recently, genetic algorithms have been successfully applied to A genetic algorithm is an evolutionary algorithm which generates near optimal solution of a problem by a guided random search method where elements (called individuals) in a given set of solutions (called population) are randomly combined and modified until some termination condition is achieved. The population evolves iteratively in order to improve a given cost function or fitness function of its individual [4].

## Depth-First Search

Our algorithm uses a branching tree to systematically enumerate all possible schedules. For example, Figure 2 shows a branching tree for the task graph in Figure 1 (a). In the tree, each node represents a task, and a branch between two nodes denotes that the parent task is scheduled no later than the child task. A path from the root to a leaf denotes a schedule. For example, a path in Figure 2 denotes the schedule shown in Figure 1 (b) [[1]](#footnote-1).

Our algorithm travels the branching tree from the root to leaves in a depth-first order. However, traveling all nodes in the branching tree has time complexity of O(*𝑛*!), which is not practical for large task graphs. The rest of this section present four rules to prune unnecessary branches.

## Pruning Partial Schedules with Same Tasks

Let us consider the branching tree in Figure 2. Assume that our algorithm already visited partial schedule and now we have reached . Note that the two partial schedules contain the same tasks with different orders. If we compare the two partial schedules, we can figure out that cannot be better than , and thus, we can prune further branches under .

How to compare the two partial schedules is as follows.

Figure 3 (a) and (b) show time charts of partial schedules and , respectively. In Figure 3 (a), one of the four cores is available at time 10, and then, task 3 is schedulable. Here, a task is schedulable if both of the following two conditions hold:

* All flow dependencies are solved.
* The number of available cores is enough to run the task.

Similarly, tasks 3, 4 and 5 are schedulable at time 30 in Figure 3 (a). In Figure 3 (b), tasks 3, 4 and 5 are schedulable at time 30. Before time 30, no task is schedulable since no core is available.



(a) Partial schedule



(b) Partial schedule

Figure 3. Partial schedules with same tasks

Now, we see that, at any time point, a set of schedulable tasks in partial schedule is a subset of that in partial schedule . For example, at time 10, a set of schedulable tasks in partial schedule is empty, which is a subset of {3}. Then, it is guaranteed that no schedule under partial schedule is better than the best schedule under , and therefore, branches under can be pruned.

In our algorithm, when we visit a new partial schedule, in other words, when we visit a new node in the branching tree, we look-up previously-visited partial schedules with same tasks, and compare their schedulable task sets. If the schedulable task set of one partial schedule is always a subset of the other, we prune the former partial schedule.

## Scheduling Exclusive Task First

Let us consider the task graph in Figure 1 again. Initially, either task 1 or 2 is schedulable at time 0. In this case, scheduling task 1 first leads to an optimal schedule in the following reason.

Since task 1 requires all of four cores, this task cannot be executed in parallel with any other tasks. We refer to a task as an *exclusive* task if the task cannot run in parallel with any other tasks which are not yet scheduled. Task 1 is an exclusive task. On the other hand, task 2 is not exclusive since task 2 can run in parallel with task 3.

Delaying execution of exclusive tasks which can be scheduled at the earliest cannot minimize the schedule length. Our algorithm schedules exclusive tasks as early as possible. When visiting a node, and if one of the branches goes to an exclusive task with the earliest start time, branches to the other tasks are pruned.

## Reducing Meaningless Idle Time

Let us consider partial schedule in the branching tree shown in Figure 2. There are three branches from task 2, going to tasks 3, 4 and 5. If we look at the time chart in Figure 3 (a), it is obvious that the branch to task 3 is the best among the three. The earliest start time of task 4 and that of task 5 are both time 30 because of the flow dependencies. On the other hand, the earliest finish time of task 3 is time 20, which is earlier than the earliest start time of the other tasks. Therefore, delaying execution of task 3 produces meaningless idle time.

During traveling a branching tree, if the earliest finish time of a child task is earlier than or equal to the earliest start time of the other children, only the former task is visited and the other branches are pruned.

## Pruning based on Lower Bound

Similar to typical branch-and-bound algorithms, our algorithm keeps a temporarily-optimal schedule and updates it when a better schedule is found. When branching to a child, our algorithm calculates the lower bound of schedule length. If the lower bound is longer than the length of the temporarily-optimal schedule, the branch is pruned.

When our algorithm visits a new node in the branching tree, we use two simple formulas as follows, in order to check the lower bound of the schedule under the node.

(6)

(7)

In the formulas, denotes the available time of core *j*. For example, in Figure 3 (a), is 30 for , and . is a set of tasks which are not yet scheduled. and denote the degree of data parallelism and execution time of task *i*, respectively. is the number of cores, and is the length of the temporarily-optimal schedule. If formula (6) holds, the schedule length under this node cannot be shorter than , and therefore further branches are pruned.

In formula (7), denotes a set of tasks which have already been scheduled. represents the total idle time in the temporarily-optimal schedule, and is defined as follows.

(8)

Formula (7) checks if the total idle time of the current partial schedule is larger than or not. If yes, further branches under the partial schedule are pruned.

## Selection of Branch

So far, four rules to prune branches are described. Another important issue in the depth-first branch-and-bound search is how to select a task to go first when multiple child tasks exist.

Out of the children, our algorithm selects the child task which has the earliest start time. In case there exist multiple tasks with the same start time, we select a task based on the PCS strategy which was presented in [4].

# Experiments

Table 1. Results for task graphs with 10 tasks on 4 cores

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task graph ID | Schedule Length | | Runtime (sec) | |
| ILP | B&B | ILP | B&B |
| rand0000 | 32 | 32 | 6,823 | <1 |
| rand0001 | 43 | 43 | 21,788 | <1 |
| rand0002 | 26 | 26 | 60,012 | <1 |
| rand0003 | 30 | 30 | 71,678 | <1 |
| rand0004 | 36 | 36 | 2,588 | <1 |
| rand0005 | 75 | 75 | 40,054 | <1 |
| rand0006 | 70 | 70 | 46,245 | <1 |
| rand0007 | 94 | 94 | 50,019 | <1 |
| rand0008 | 121 | 121 | 6,115 | <1 |
| rand0009 | 79 | 79 | 58,830 | <1 |
| rand0010 | 23 | 23 | 55,539 | <1 |
| rand0011 | 33 | 33 | 55,068 | <1 |
| rand0012 | 33 | 33 | 15,171 | <1 |
| rand0013 | 31 | 31 | 42,571 | <1 |
| rand0014 | 53 | 53 | 44,250 | <1 |
| rand0015 | 81 | 81 | <1 | <1 |
| rand0016 | 77 | 77 | <1 | <1 |
| rand0017 | 100 | 100 | 41,675 | <1 |
| rand0018 | 72 | 72 | <1 | <1 |
| rand0019 | 70 | 70 | 220,650 | <1 |

We implemented our proposed scheduling algorithm in C++, and conducted two sets of experiments to test the effectiveness of the proposed algorithm.

In the first experiments, we use 20 sets of 10 tasks, derived from Standard Task Graph (STG) [9]. An integer linear programming (ILP) technique (see Section II) was used as a counterpart to our algorithm. Although the ILP technique is guaranteed to yield optimal schedules, it takes a long time which is often unacceptable. In order to solve the ILP problems, IBM ILOG CPLEX 12.5 was used. The experiments were conducted on dual Xeon processors (E5-2650, 2.00Hz) with 128GB memory.

Table 1 shows scheduling results for 20 task graphs with 10 tasks on 4 cores. ILP and B&B denote the ILP technique using CPLEX and our branch-and-bound algorithm, respectively. The results in the table show that our algorithm yields the same schedule length as the ILP techniques in any case. Although we have not mathematically proved the correctness of our algorithm yet, our algorithm always found the optimal schedule as long as we tested.

As shown in Table 1, in any cases of 10 tasks, our branch-and-bound algorithm found optimal schedules within a second. On the other hand, the runtime of CPLEX significantly varied depending on the task graph. In the worst case, it took more than 60 fours for CPLEX to find the optimal schedule for 10 tasks.

In the next set of experiments, we compared our branch-and-bound algorithms with two existing heuristic ones. One is the PCS algorithm [4] and the other is the dual-mode algorithm [5]. We used 20 sets of 50 tasks and [another](file:///C:\Users\scomu\AppData\Local\Youdao\Dict\Application\7.0.1.0214\resultui\dict\result.html?keyword=another) 20 sets of 100 tasks from [9]. The number of cores was changed from two to eight. Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks.

The results of task sets with 50 tasks are shown in Table 3 and 100 tasks are shown in Table 3. In Table 2 and 3, the best solution is marked in red. The “X” means that B&B cannot find the optimal result within 12 hours. In this case, the results are surrounded by parentheses.

As the Table 2 and 3 shows, although for some task sets B&B cannot find the optimal result in a practical time due to the huge computational cost, B&B always yields the best schedule results compare with heuristic algorithms.

The runtimes of the PCS and dual-mode algorithms were always less than 1 second. On the other hand, the runtime of our branch-and-bound algorithm significantly varied depending on the task graph. For task sets with 50 tasks on 2 or 4 cores(Table 2A and 2B), the runtime of our algorithm is less than 1 minute in most cases. Nevertheless, along with the increasing numbers of tasks and core numbers, the average runtimes also increased. For task sets with 100 tasks on 16 cores(Table 3D), 50% task sets are failed to yield an optimal result in a practical time.

# Conclusions

In this paper, we proposed a branch-and-bound algorithm for a task scheduling problem which takes into account both task-parallelism and data-parallelism. We presented four rules to prune non-optimal branches. The experiments show that our algorithm could find best schedules in a practical time for most task sets with tasks under 50.

However, for task sets with more tasks, for example, with 100 tasks, our algorithm may fail to obtain optimal results in a reasonable runtime due to the huge computational cost. In future, we plan to use parallel computing to accelerate our algorithm, in order to solve tasks sets with more tasks in a shorter runtime. we also plan to formally prove the correctness of our algorithm.

Table 2B.

Results for task graphs with 50 tasks on 4 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 155 | 168 | 167 | 8 | <1 | <1 |
| rand0001 | 202 | 220 | 211 | <1 | <1 | <1 |
| rand0002 | 162 | 173 | 170 | <1 | <1 | <1 |
| rand0003 | 181 | 194 | 194 | 114 | <1 | <1 |
| rand0004 | 166 | 167 | 167 | <1 | <1 | <1 |
| rand0005 | 397 | 439 | 426 | <1 | <1 | <1 |
| rand0006 | 258 | 275 | 270 | 6 | <1 | <1 |
| rand0007 | (339) | 357 | 354 | X | <1 | <1 |
| rand0008 | 387 | 409 | 407 | <1 | <1 | <1 |
| rand0009 | 314 | 327 | 356 | 3 | <1 | <1 |
| rand0010 | 128 | 131 | 131 | 50 | <1 | <1 |
| rand0011 | 170 | 181 | 176 | <1 | <1 | <1 |
| rand0012 | 179 | 197 | 192 | 2 | <1 | <1 |
| rand0013 | 178 | 186 | 192 | 7 | <1 | <1 |
| rand0014 | 159 | 171 | 167 | 462 | <1 | <1 |
| rand0015 | 345 | 376 | 373 | <1 | <1 | <1 |
| rand0016 | 292 | 318 | 319 | <1 | <1 | <1 |
| rand0017 | 359 | 377 | 378 | 6,800 | <1 | <1 |
| rand0018 | 363 | 403 | 396 | <1 | <1 | <1 |
| rand0019 | 323 | 342 | 330 | <1 | <1 | <1 |

Table 3A.

Results for task graphs with 50 tasks on 2 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 196 | 203 | 203 | 2.46 | <1 | <1 |
| rand0001 | 222 | 232 | 232 | <1 | <1 | <1 |
| rand0002 | 186 | 188 | 188 | <1 | <1 | <1 |
| rand0003 | 224 | 224 | 224 | 15.8 | <1 | <1 |
| rand0004 | 174 | 177 | 177 | <1 | <1 | <1 |
| rand0005 | 465 | 495 | 495 | <1 | <1 | <1 |
| rand0006 | 338 | 351 | 351 | <1 | <1 | <1 |
| rand0007 | 384 | 384 | 384 | 37.9 | <1 | <1 |
| rand0008 | 428 | 434 | 434 | <1 | <1 | <1 |
| rand0009 | 382 | 386 | 386 | <1 | <1 | <1 |
| rand0010 | 153 | 153 | 153 | 4.15 | <1 | <1 |
| rand0011 | 190 | 205 | 205 | <1 | <1 | <1 |
| rand0012 | 192 | 208 | 208 | <1 | <1 | <1 |
| rand0013 | 234 | 238 | 238 | <1 | <1 | <1 |
| rand0014 | 195 | 195 | 195 | <1 | <1 | <1 |
| rand0015 | 402 | 425 | 425 | <1 | <1 | <1 |
| rand0016 | 366 | 374 | 374 | <1 | <1 | <1 |
| rand0017 | 434 | 439 | 439 | 19.8 | <1 | <1 |
| rand0018 | 421 | 428 | 428 | <1 | <1 | <1 |
| rand0019 | 376 | 393 | 393 | <1 | <1 | <1 |

Table 4D.

Results for task graphs with 50 tasks on 16 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 136 | 156 | 151 | 4680 | <1 | <1 |
| rand0001 | 192 | 195 | 198 | 1.67 | <1 | <1 |
| rand0002 | 128 | 150 | 146 | 9.09 | <1 | <1 |
| rand0003 | (158) | 169 | 165 | X | <1 | <1 |
| rand0004 | 144 | 158 | 157 | <1 | <1 | <1 |
| rand0005 | 360 | 406 | 388 | 8.51 | <1 | <1 |
| rand0006 | 243 | 268 | 249 | 354 | <1 | <1 |
| rand0007 | (260) | 301 | 279 | X | <1 | <1 |
| rand0008 | 319 | 360 | 345 | <1 | <1 | <1 |
| rand0009 | 260 | 289 | 292 | 149 | <1 | <1 |
| rand0010 | (122) | 126 | 127 | X | <1 | <1 |
| rand0011 | 129 | 135 | 146 | 1.13 | <1 | <1 |
| rand0012 | 164 | 174 | 169 | 26.9 | <1 | <1 |
| rand0013 | 144 | 154 | 155 | 251 | <1 | <1 |
| rand0014 | (143) | 160 | 147 | X | <1 | <1 |
| rand0015 | 309 | 325 | 347 | <1 | <1 | <1 |
| rand0016 | 254 | 286 | 293 | 37.9 | <1 | <1 |
| rand0017 | (308) | 333 | 312 | X | <1 | <1 |
| rand0018 | 326 | 342 | 344 | 1.1 | <1 | <1 |
| rand0019 | 299 | 334 | 336 | 5.02 | <1 | <1 |

Table 5C.

Results for task graphs with 50 tasks on 8 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 139 | 149 | 148 | 1250 | <1 | <1 |
| rand0001 | 184 | 203 | 201 | 2.41 | <1 | <1 |
| rand0002 | 139 | 161 | 153 | 1.28 | <1 | <1 |
| rand0003 | 165 | 175 | 180 | 7210 | <1 | <1 |
| rand0004 | 147 | 150 | 155 | <1 | <1 | <1 |
| rand0005 | 379 | 432 | 406 | <1 | <1 | <1 |
| rand0006 | 231 | 259 | 246 | 306 | <1 | <1 |
| rand0007 | 296 | 336 | 312 | 13700 | <1 | <1 |
| rand0008 | 333 | 366 | 354 | 1.53 | <1 | <1 |
| rand0009 | 289 | 323 | 326 | 12.5 | <1 | <1 |
| rand0010 | 118 | 127 | 125 | 1380 | <1 | <1 |
| rand0011 | 159 | 180 | 172 | <1 | <1 | <1 |
| rand0012 | 170 | 183 | 178 | 15.4 | <1 | <1 |
| rand0013 | 158 | 171 | 171 | 294 | <1 | <1 |
| rand0014 | 144 | 166 | 163 | 1860 | <1 | <1 |
| rand0015 | 289 | 304 | 307 | 1.61 | <1 | <1 |
| rand0016 | 245 | 269 | 266 | 24.4 | <1 | <1 |
| rand0017 | (286) | 306 | 314 | X | <1 | <1 |
| rand0018 | 328 | 358 | 354 | <1 | <1 | <1 |
| rand0019 | 343 | 361 | 365 | 5.63 | <1 | <1 |

Table 3B.

Results for task graphs with 100 tasks on 4 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 358 | 388 | 376 | 3610 | <1 | <1 |
| rand0001 | 335 | 348 | 343 | 11500 | <1 | <1 |
| rand0002 | 390 | 413 | 424 | 9.41 | <1 | <1 |
| rand0003 | 325 | 341 | 338 | 33.2 | <1 | <1 |
| rand0004 | 340 | 354 | 366 | 8470 | <1 | <1 |
| rand0005 | 655 | 704 | 682 | 2.37 | <1 | <1 |
| rand0006 | 706 | 785 | 737 | 23.6 | <1 | <1 |
| rand0007 | 711 | 760 | 735 | 9310 | <1 | <1 |
| rand0008 | (694) | 701 | 706 | X | <1 | <1 |
| rand0009 | 747 | 783 | 779 | 88.9 | <1 | <1 |
| rand0010 | 363 | 385 | 392 | 72.2 | <1 | <1 |
| rand0011 | 364 | 394 | 377 | 165 | <1 | <1 |
| rand0012 | 405 | 432 | 434 | <1 | <1 | <1 |
| rand0013 | 390 | 404 | 427 | 31.8 | <1 | <1 |
| rand0014 | 316 | 354 | 334 | 33.8 | <1 | <1 |
| rand0015 | 650 | 706 | 683 | 102 | <1 | <1 |
| rand0016 | 603 | 667 | 646 | 8.93 | <1 | <1 |
| rand0017 | 705 | 746 | 755 | 135 | <1 | <1 |
| rand0018 | 571 | 628 | 626 | 19700 | <1 | <1 |
| rand0019 | 659 | 700 | 709 | 4.32 | <1 | <1 |

Table 3A.

Results for task graphs with 100 tasks on 2 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 431 | 431 | 431 | 18.1 | <1 | <1 |
| rand0001 | 396 | 401 | 401 | <1 | <1 | <1 |
| rand0002 | 446 | 459 | 459 | <1 | <1 | <1 |
| rand0003 | 391 | 406 | 406 | <1 | <1 | <1 |
| rand0004 | 393 | 393 | 393 | 73.4 | <1 | <1 |
| rand0005 | 774 | 814 | 814 | <1 | <1 | <1 |
| rand0006 | 820 | 868 | 868 | <1 | <1 | <1 |
| rand0007 | 845 | 861 | 861 | 6.98 | <1 | <1 |
| rand0008 | 792 | 796 | 796 | <1 | <1 | <1 |
| rand0009 | 910 | 947 | 947 | <1 | <1 | <1 |
| rand0010 | 445 | 464 | 464 | 1.2 | <1 | <1 |
| rand0011 | 440 | 445 | 445 | 227 | <1 | <1 |
| rand0012 | 456 | 469 | 469 | <1 | <1 | <1 |
| rand0013 | 472 | 480 | 480 | 0.496 | <1 | <1 |
| rand0014 | 386 | 391 | 391 | 36.1 | <1 | <1 |
| rand0015 | 763 | 781 | 781 | <1 | <1 | <1 |
| rand0016 | 748 | 764 | 764 | <1 | <1 | <1 |
| rand0017 | 857 | 860 | 860 | 5.61 | <1 | <1 |
| rand0018 | 720 | 724 | 724 | 28.5 | <1 | <1 |
| rand0019 | 747 | 749 | 749 | <1 | <1 | <1 |

Table 3C.

Results for task graphs with 100 tasks on 8 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | 316 | 356 | 337 | 1330 | <1 | <1 |
| rand0001 | (317) | 326 | 330 | X | <1 | <1 |
| rand0002 | 346 | 380 | 372 | 44.5 | <1 | <1 |
| rand0003 | 320 | 338 | 342 | 211 | <1 | <1 |
| rand0004 | (306) | 340 | 327 | X | <1 | <1 |
| rand0005 | 644 | 713 | 698 | 7.94 | <1 | <1 |
| rand0006 | 659 | 712 | 703 | 190 | <1 | <1 |
| rand0007 | (622) | 675 | 657 | X | <1 | <1 |
| rand0008 | (614) | 637 | 638 | X | <1 | <1 |
| rand0009 | 684 | 785 | 737 | 135 | <1 | <1 |
| rand0010 | 315 | 338 | 327 | 131 | <1 | <1 |
| rand0011 | (346) | 353 | 349 | X | <1 | <1 |
| rand0012 | 380 | 431 | 423 | <1 | <1 | <1 |
| rand0013 | 363 | 382 | 385 | 1270 | <1 | <1 |
| rand0014 | (314) | 327 | 319 | X | <1 | <1 |
| rand0015 | (621) | 697 | 646 | X | <1 | <1 |
| rand0016 | 599 | 625 | 657 | 87.7 | <1 | <1 |
| rand0017 | 684 | 730 | 730 | 4750 | <1 | <1 |
| rand0018 | (604) | 657 | 642 | X | <1 | <1 |
| rand0019 | 631 | 679 | 679 | 9.39 | <1 | <1 |

Table 3D.

Results for task graphs with 100 tasks on 16 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task  graph ID | Schedule Length | | | Runtime (sec) | | |
| B&B | PCS | Dual-mode | B&B | PCS | Dual-mode |
| rand0000 | (311) | 335 | 333 | X | <1 | <1 |
| rand0001 | (302) | 307 | 304 | X | <1 | <1 |
| rand0002 | 335 | 365 | 355 | 11 | <1 | <1 |
| rand0003 | 289 | 314 | 309 | 712 | <1 | <1 |
| rand0004 | (297) | 317 | 307 | X | <1 | <1 |
| rand0005 | 623 | 668 | 676 | 13.6 | <1 | <1 |
| rand0006 | 629 | 687 | 666 | 562 | <1 | <1 |
| rand0007 | (604) | 665 | 637 | X | <1 | <1 |
| rand0008 | (597) | 607 | 610 | X | <1 | <1 |
| rand0009 | 663 | 728 | 713 | 394 | <1 | <1 |
| rand0010 | 315 | 362 | 354 | 581 | <1 | <1 |
| rand0011 | (299) | 336 | 331 | X | <1 | <1 |
| rand0012 | 387 | 410 | 421 | <1 | <1 | <1 |
| rand0013 | 353 | 375 | 372 | 2060 | <1 | <1 |
| rand0014 | (305) | 313 | 306 | X | <1 | <1 |
| rand0015 | (540) | 606 | 557 | X | <1 | <1 |
| rand0016 | 594 | 648 | 645 | 73.6 | <1 | <1 |
| rand0017 | 632 | 677 | 692 | 23400 | <1 | <1 |
| rand0018 | (563) | 591 | 595 | X | <1 | <1 |
| rand0019 | 631 | 676 | 672 | 5.1 | <1 | <1 |

# Acknowledgment

This work is in part supported by KAKENHI 15H02680.

# References

1. S. B. Hassen, H. E. Bal, and C. J. H. Jacobs, “A task and data-parallel programming language based on shared objects,” *ACM Trans. on Programming Languages and Systems.*, vol. 20, no. 6, 1998.
2. S. Ramaswamy, S. Sapatnekar, and P. Banerjee, “A framework for exploiting task and data parallelism on distributed memory multicomputers,” *IEEE Trans. on Parallel and Distributed Systems.*, vol. 8, no. 11, 1997.
3. E. G. Coffman, *Computer and Job-shop Scheduling Theory*, Wiley, 1976.
4. Y. Liu, L.Meng, I. Taniguchi, and H. Tomiyama, “Novel list scheduling strategies for task graphs with data parallelism," *International Journal on Networking and Computing*, vol. 4, no. 2, 2014.
5. Y. Liu, L. Meng, I. Taniguchi and H. Tomiyama, “A dual-mode scheduling algorithm for task graphs with data parallelism,” *Asia Pacific Conference on Circuits and Systems*, 2014.
6. H. Kasahara and S. Narita, “Practical multiprocessor scheduling algorithms for efficient parallel processing,” *IEEE Trans. on Computers*, vol. C-33, no. 11, 1984.
7. S. Fujita, “A branch-and-bound algorithm for solving the multiprocessor scheduling problem with improved lower bounding techniques,” *IEEE Trans. on Computers*, vol. 60, no. 7, 2011.
8. O. Sinnen, A. V. Kozlov, and A. Z. S. Shahul, “Optimal scheduling of task graphs on parallel systems,” *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2008
9. http://www.kasahara.elec.waseda.ac.jp/schedule/   
   (Last accessed: June 2016)

1. Paths also result in the same schedule as shown in Figure 1 (b). [↑](#footnote-ref-1)