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

**Abstract — This paper studies a task scheduling problem which schedules a set of data-parallel tasks on multiple cores. Unlike most of the 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 choices to obtain suboptimal results. In this paper, an efficient method based on genetic algorithm is proposed to solve this problem. Different from 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 speed 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 an 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 (example). Recently, Genetic Algorithms have been widely studied as useful heuristics for obtaining high-quality solutions for task scheduling problem (example).

Unfortunately, Majority of the works deal task scheduling with genetic algorithms only considered about task parallelism. Many studies have 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 reduces greatly the size of search space and improves the performance of the algorithm. The efficient genetic operators (select, crossover, mutation) corresponding to this chromosome also are presented. In addition, 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 for scheduling problem and corresponding crossover and mutation strategies are used to minimize the execution time.
* 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. a brief introduction of genetic algorithm is given in section III. Our proposed algorithm is presented in section IV & V. Experiments are presented in Section VI.



(a) A task graph



(b) An optimal schedule

Figure 1. A scheduling example

Table 1. Basic terms of genetic algorithm

|  |  |
| --- | --- |
| Terms | Meaning |
| *Environment* | Problem |
| *Individual* | Solution to a problem |
| *Chromosome* | Representation for a solution |
| *Population* | Set of solutions represented by chromosome |
| *Gene* | The basic element for a chromosome |
| *Fitness* | The degree of adaptation for individual to his environment |
| *Selection* | The operation of choosing the to be parents |
| *Crossover* | The operation of producing child |
| *Mutation* | The operation of randomly alter genes |

# Problem Definition

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 the 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.

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

The genetic algorithm was first invented by Holland [ref], it is a meta-heuristic inspired by the processes observed in natural selection. This algorithm thinks of a set of candidate solutions for a problem as biological population, the fitness of each individual is evaluated, according to Darwin's theory: "Survival of the fittest". The fitter ones are more likely selected and produce next generations. During this breeding process, the spontaneous mutations occur, may create individuals better adapt the environment. The basic terms of genetic algorithm used in this paper are shown and defined in Table 1.

Typically, a genetic algorithm consists of the following steps.

1. *Initialization: Generate the initial population*
2. *Calculation of the fitness*: The fitness of each individual is calculated according to the definition of the problem.
3. *Selection*: Select the adapted individuals as parents for the next generation
4. *crossover*: New individuals are produced.
5. *Mutation*: Alter genes for individual
6. Go step 2 until the stopping criteria reached.

# PROPOSED ALGORITHM

In this section, the proposed algorithm was presented in detail. Our algorithm based on genetic algorithm described in Section III. In the following, we examine each step of genetic algorithm for the scheduling problem with task and data parallelism.

## Chromosome representation

In genetic algorithm, the chromosome is a set of strings, which represent a potential solution for a problem. Finding an adequate chromosome is one of the most important issues for a successful application of genetic algorithms. Due to all genetic operators are defined on the chromosome, a good chromosome representation will make the genetic operators easier to implement, and limit the unnecessary search space. Several different types of chromosomes are used in previous works for task scheduling problem, all of them contain the task scheduling and mapping information, means that both the ordering of tasks execution and tasks are mapped on which cores are encoded in chromosome.

This kind of chromosome may not very efficient for task scheduling with task and data parallelism, because the tasks can be mapped on multiple cores, therefore, the length of chromosome may tend to very long. We intend to find a more condensed representing of chromosome. Our proposed chromosome only encodes information about the ordering of task execution, ignores tasks are mapped on which cores, this representation also reduce greatly the size of search space and improves the performance of the algorithm.

The proposed chromosome representation is an array of N elements. N represents the number of tasks. this array determines the sequence of the processing of the tasks. Figure 2 shows an example of the proposed chromosome. In Figure 2, task1 (T1) will be scheduled first, the next one is task2 (T2), and so on.

Another very important thing is the precedence relation between tasks must be kept in our chromosome. It means that in a valid chromosome, tasks must be placed in before their descendants.

## Initialization

Our algorithm begins with a randomly generate a set of candidate solutions represented by chromosome which defined in (1). Our algorithm of initialization guarantees that all the generated chromosomes are valid.

The pseudocode of initialization is shown as below:

C is a random chromosome. The Ti\_parent indicates the parent tasks for task i. The important assumption here is: a task with a larger id is not a parent for tasks with small id than its. we need to reorder the tasks if the task graph does not satisfy this assumption before scheduling.

## Fitness function

The fitness function is used to decode a chromosome and assigns it a fitness value, we use a deterministic algorithm to schedule the tasks according to the chromosome and the task graph. This algorithm also restores the mapping information, that is, tasks are mapped on which cores. The detail of fitness function is explained in the below:

1. Denote the first gene in the chromosome as Ti (the task need to be scheduled) and remove it from the chromosome.
2. The start time of Ti is calculated as follows:
   1. a = The maximum of the finished time of all parent tasks of Ti
   2. b = The earliest time when enough unoccupied cores are ready for executing Ti
3. The start time of Ti = MAX(a, b)
4. Update the finished timeofTi: The start time of Ti + the execution time of Ti.
5. Go step 1 until all tasks were scheduled.

After the finished time for all tasks were calculated, the fitness function can be defined as follow:

Figure 2. A chromosome example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T1 | T2 | T4 | T5 | T3 |

*f* is the fitness value of this chromosome, Because the task scheduling problem aims to minimize the overall scheduling length, the smaller fitness value is better.

## Selection

The selection operator is guided by the fitness value of each chromosome calculated by C. Chromosomes with better fitness value have a larger probability to survive. Different approaches were used in the selection operators such as roulette wheel selection, rank selection, and steady-state selection. Our algorithm uses roulette wheel.

For i to N //N is number of tasks in task graph

min = MAX(C[Ti\_parent])+1

index = RADOM\_BETWEEN(min, i)

order[i] = index

For j to (i-1)

If(order[j] >= index)

order[i] += 1

For i to N

C[order[i]] = i

In roulette wheel selection, each chromosome in the population is allocated a segment on a virtual roulette wheel of a size proportional to its fitness.

The adapter chromosomes have a larger segment, it means such chromosomes more likely to be selected when the wheel is spin.

This size of the segment for each task was calculated as below:

is the better fitness value in population, is the fitness value of current chromosome. The part of denominator is a normalization factor.

Table 2. The list of GAs parameters

|  |  |
| --- | --- |
| Terms | value |
| *Population size* | 16384 |
| *α (selection)* | 0.6 |
| *m (mutation)* | 0.05 |
| *Max generations* | 50 |

## Crossover

The crossover operator is analogous to the biological crossover. Two chromosomes are chosen from the population by *selection* described in D, the child chromosomes are produced from them.

Due to our chromosome represents the order of task execution, simply exchanges part of gene between two chromosomes may produce invalid chromosomes, we use the following method to ensure the generated chromosomes is valid:

1. Select two chromosomes.

The *Chromosome A* and the *Chromosome B*

1. A random crossover point in *Chromosome A* is chosen.
2. Copy the left segment of crossover point in *Chromosome A* to the *child.*
3. Remove genes in *Chromosome B,* which have been passed tothe *child.*
4. Append remaining *Chromosome B* to the end of the *child.*

## Mutation

Mutation operator will randomly alter one or more gene. In genetic algorithms, selection operator will remove bad chromosomes, but lose the diversity in the population. Mutation is a very important mechanism to recover it. Hence, the mutation operator gives us the possibility of producing better child than their parents. Our mutation operator also guarantees that after mutation the chromosomes are valid.

In the proposed chromosome, the value of ith gene indicate a task id, and the execution sequence of this task is i. our *mutation* will change the order of execution of tasks by following algorithm.

1. The chromosome is subjected to mutation with a probability *m*. Generate a random number *p* (from 0 to 1) for each gene. If *p* > *m* go step 2.
2. Calculation the new execution sequence of current task as follows:
   1. *upper* = current execution sequence.
   2. *lower =* the maximum ofthe execution sequence of the parents of current task + 1.
3. New execution sequence of this task = RANDOM\_BETWEEN(*lower, upper*)

# PARALLELIZATION OF ALGORITHM

The genetic algorithm may require a large execution time because a large number of chromosomes must be generated and evaluated. Therefore, we use the parallel technique to improve computational efficiency. There are various types of parallelization techniques i.e., Pthreads, C++11 STL threads, OpenMP, Intel TBB, CUDA and OpenCL. We concentrate on OpenMP, because it provides an easy and flexible way to implement. OpenMP is an API for writing multi-threaded applications on shared memory multi-processor architecture.

In our algorithm, A data dependency occurs when calculating normalization factor in the selection operator, but otherwise, all of the genetic operators are completely performed independently, we can easily parallelize our algorithm.

# Experiments

The proposed algorithm was implemented in C++. We evaluated on arbitrary task graphs of 50 tasks. The task graphs we used are derived from Standard Task Graph (STG) [9]. Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks. We conducted all experiments on Intel i7 (i7-4790K, 4 cores) and 32GB memory on Ubuntu 14.04.

In the above discussion, we have presented a set of important parameters. The parameters have strong effects on the execution time and the quality of results. Finding the optimal set of parameters is another important and hard mission, but these are not included in the scope of this article. We simply set the parameters in Table 2.

To evaluate our proposed genetic algorithms, we compared it with an exact B&B algorithm (Liu, 2017) and two heuristic algorithms based on list scheduling. One is the PCS algorithm (Liu, 2014) and the other is the dual-mode algorithm (Liu, 2017)

The results are shown in Tables 2.The schedule lengths obtained by the four algorithms are compared. The values in parentheses are schedule lengths normalized to the B&B algorithm. The closer this value to 1, the nearer to global optimal result. The global optimal results were shaded in red.

Table 3.   
Results for task graphs with 50 tasks on 4 cores

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task graph ID | Schedule Length | | | |
| B&B  (Liu 2017) | PCS (Liu 2014) | Dual-mode (Liu 2017) | GAs |
| 50-0000 | 155  (1.000) | 168  (1.084) | 167  (1.084) | 159  (1.026) |
| 50-0001 | 202  (1.000) | 220  (1.089) | 211  (1.045) | 203  (1.005) |
| 50-0002 | 162  (1.000) | 173  (1.068) | 170  (1.049) | 165  (1.019) |
| 50-0003 | 181  (1.000) | 194  (1.072) | 194  (1.072) | 185  (1.022) |
| 50-0004 | 166  (1.000) | 167  (1.006) | 167  (1.006) | 166  (1.000) |
| 50-0005 | 397  (1.000) | 439  (1.106) | 426  (1.073) | 404  (1.018) |
| 50-0006 | 258  (1.000) | 275  (1.066) | 270  (1.047) | 266  (1.031) |
| 50-0007 | 339  (1.000) | 357  (1.053) | 354  (1.044) | 340  (1.003) |
| 50-0008 | 387  (1.000) | 409  (1.057) | 407  (1.052) | 390  (1.008) |
| 50-0009 | 314  (1.000) | 327  (1.041) | 356  (1.134) | 318  (1.013) |
| 50-0010 | 128  (1.000) | 131  (1.023) | 131  (1.023) | 129  ] (1.008) |
| 50-0011 | 170  (1.000) | 181  (1.065) | 176  (1.035) | 170  (1.000) |
| 50-0012 | 179  (1.000) | 197  (1.101) | 192  (1.073) | 179  (1.000) |
| 50-0013 | 178  (1.000) | 186  ((1.045) | 192  (1.079) | 182  (1.022) |
| 50-0014 | 159  (1.000) | 171  (1.075) | 167  (1.050) | 161  (1.013) |
| 50-0015 | 345  (1.000) | 376  (1.090) | 373  (1.081) | 347  (1.006) |
| 50-0016 | 292  (1.000) | 318  ((1.089) | 319  (1.092) | 292  (1.000) |
| 50-0017 | 359  (1.000) | 377  (1.050) | 378  (1.053) | 365  (1.017) |
| 50-0018 | 363  (1.000) | 403  (1.110) | 396  (1.091) | 363  (1.000) |
| 50-0019 | 342  (1.000) | 342  (1.059) | 330  (1.022) | 326  (1.009) |
| Average | 1.000 | 1.067 | 1.060 | 1.011 |

Table 4. Execution time comparison

|  |  |
| --- | --- |
| Algorithm | Execution time  (millisecond) |
| B&B | 892~88,100,000 |
| PCS | <1 |
| Dual-mode | <1 |
| GAs | 3985 |
| GAs (openMP) | 553 |

We can find the proposed algorithm achieves global optimization with a high probability (25%). Even in the case where genetic algorithm was not found the best result, the result is also very close to global optimization Compared with the dual-mode or PCS algorithm, the GAs found shorter schedule length with an average of 5%.

Table 4 shows the execution time taken by each algorithm solving a task graph with 50 tasks on 4 cores. The runtime of B&B algorithm significantly varied depending on the task graph. Therefore, we show the best and worst execution time in 20 task graphs. As for other algorithms, the average execution time was shown

Although the execution time of GAs is longer than that of list-scheduling algorithms, however, it should be noted that a near-linear speedup achieved by using OpenMP (our CPU has 4 cores).

# Conclusions

In this paper, we proposed an efficient method based on genetic algorithm for task scheduling problem which takes into account both task-parallelism and data-parallelism. We also proposed a new chromosome representation and corresponding genetic operators aimed to reduce the search space and improve the computing speed.

The experiments show that our algorithm found the balance between computation time and solution quality. We can near-linear speed up our algorithm by using parallel computing method. In future, we plan to use GPU to accelerate our algorithm.

# 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)