Research on Scheduling with Dependencies

Chen Yuxuan

Computer Science and Engineering, Waseda University
Kasahara Laboratory
* E-mail: sandy2008@toki.waseda.jp.

This paper presents multiple algorithms on scheduling with dependencies, and compare them using the random Standard Task Graph. The complexity and efficiency is also analyzed.

1 Introduction

Scheduling is a problem of assigning tasks to processing machines to complete their work. The main objective in scheduling of dependent tasks is to find out the minimize time of processing the tasks. The scheduling problem is NP-hard, exact solutions are hard to be generate in polynomial time. When the tasks have dependencies with each other, the algorithm is different from the other scheduling problems. In this paper, our goal is to find the optimum solution or get close to the optimum solution, and analyze the complexity and efficiency of different methods.

We applied 4 methods to our scheduling with dependencies problem: Greedy Algorithm, Critical Path/Most Immediate Successor First (CP/MISF) Algorithm, Genetic Algorithm, Depth First/Implicit Heuristic Search (DF/IHS) Algorithm.

To test our proposed Algorithms, we used the Standard Task Graph Set (STG) which is a kind of benchmark for evaluation of multiprocessor scheduling algorithms. It represented our tasks and their dependencies, and we implemented our algorithm using Python and C language.

We developed an evaluation algorithm to calculate the completion time using only the order of tasks and PEs.

For every tests, we recorded the orders of the tasks and PE and we used the to evaluate the spent time of the specific order.

2 Algorithms

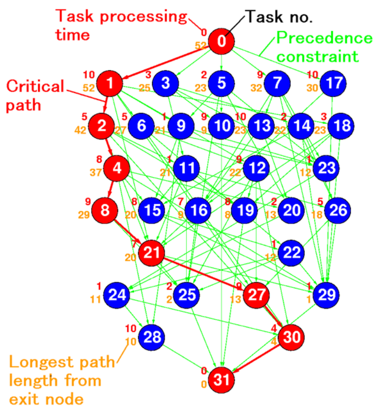

Figure 1: Standard Task Graph

Figure.1 is the Standard Task Graph which shows the relationships between the tasks. And our algorithms will be implemented according to these graphs.

2.1 Evaluation

This section will introduce an algorithm to evaluate the time of a single solution.

We derive this equation to calculate the starting time of a task:

$$starting\ time(task) = max(max(completion\ time\ of\ parents), available\ time\ of\ PE) \tag{1}$$

and we derive another equation to calculate the completion time of each tasks:

$$completion\ time(task) = starting\ time(task) + execution\ time(on\ particular\ PE) \tag{2}$$

And the maximum completion time of the task is the completion time of the whole scheduling problem.

2.2 Greedy Algorithm

The logic of the Greedy Algorithm is very simple: assign the task with the longest time to the PE with minimum completion time.

The complexity using Big O Analysis:

$$O(n) \tag{3}$$

where n is the number of tasks.

2.3 CP/MISF Algorithm

In Critical Path/Most Immediate Successor First (CP/MISF), we define a route called Critical Path which describes the longest sequence of tasks in the scheduling problem.

List Scheduling:

1) Find the length of the Critical Path of all the tasks.

2) According to the initial conditions of the tasks, find all the ready tasks.

3) List the PE lists, then find out all the PE that can be assigned with the tasks.

4) Sort the task list as a priority list, the key is the length of the Critical Path. The longer the Critical Path is, the higher the priority order is.

5) Find the list that can assign all the tasks to the PEs we get in 3) and 4), then we can return to 2), we end the scheduling if there are no tasks remained.

The complexity using Big O Analysis:

$$O(n^2) \tag{4}$$

2.4 GA Algorithm

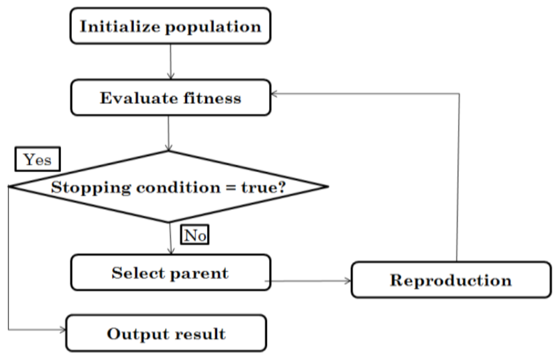

Figure 2: Standard Task Graph

Genetic Algorithm is a metaheuristic inspired by the process of natural selection. And the algorithm can be shown in this graph:

1) We randomly generate number of solutions as its population, and each solution is defined as a chromosome. And we use our

2) We check whether the stopping condition is achieved or not, and select number of couples for producing new generations.

3) We develop two methods to produce new generations: crossover and mutation.

Crossover: swap a selected portion or bunch of genes from a couple of chromosomes to each other and generate new solutions.

Mutation: randomly select two points in a chromosome and swap their places.

The time of population and iterations will also be decided by the number of the tasks.

2.5 GA Algorithm with selected population

The principle is same as the GA Algorithm, except we added the result of Greedy Algorithm and CP/MISF as the initial population.

2.6 DF/IHS

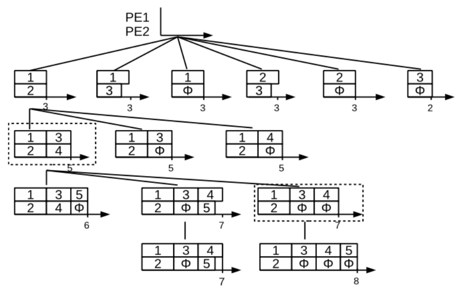

Figure 3: The search tree of DF/IHS Algorithm

Depth First/Implicit Heuristic Search is the Algorithm to find the exact solution of the problem, and it is based on depth first search.

1) As shown in Figure.3, we will traverse tree from the root, and explores as far as possible along each branch before backtracking.

2) Use the Evaluation Algorithm to evaluate every nodes, and record the node in the Hash Table. For example, when task 1 and task 3 are assigned, we record the time in the Table with the key, '10100' (if there are 5 tasks in total).

And we are using three strategies to reduce the number of the nodes:

A. Stop searching the node if we reached a node with length of Critical Path.

B. If the completion time of a node is larger than the longest completion time of the searched node, we cut the branch of the first node.

C. If the completion time of the PEs of the nodes, is larger than the completion time of all the PEs of another node with the same tasks, we cut the branch of the first node.

The complexity using Big O Analysis:

$$O(n^n) \tag{5}$$

where n stands for the number of the tasks.

3 Results

3.1 Conditions

The programs are implemented in Python, and all the codes are open sourced in Appendix for testing.

The hardware conditions are shown below:

MacBook (Retina, 12-inch, Early 2016)

CPU: 1.1 GHz Intel Core m3

Memory: 8 GB 1867 MHz LPDDR3

GPU: Intel HD Graphics 515 1536 MB

Since the number of the tasks were set to 50, it is easy to calculate all the possible solutions, we did not set a limitation of wall-clock time. Which means, we will not stop running our programs until we get the result we wanted to get.

3.2 Comparision

Table 1: Relation between Algorithms and Time

	rand0000(5PE)	rand0001(4PE)	rand0002(4PE)	rand0003(6PE)	rand0004(4PE)
Greedy	60	120	86	66	85
CP/MISF	66	97	77	58	73
Genetic	55	90	71	52	71
DF/IHS	55	89	71	52	71

In this section, we will compare the results of different algorithms. For rand0000, rand0001, rand0002, rand0003, rand0004, we compared all the algorithms and the result is shown in Table.1. We can see that both Genetic Algorithm and DF/IHS can get the optimized results.

And by comparing these results, we can have the following conclusions:

1) The results of Greedy Algorithm and CP/MISF are worse than Genetic and DF/IHS, and in many of the cases, CP/MISF is better than Greedy Algorithm, we can take the result of the less time when we are aiming to get the minimum time.

2) The results of Genetic Algorithm and DF/IHS both can get optimized results, but the time to get these results are very different, we will compare them in the following part.

3) In some of the cases, Genetic Algorithm did not work as the optimized results, this was caused by the wrong selection of the seeds, or the lack of the population.

4 Conclusions

For a most economic solution, we propose the combination of Greedy Algorithm and CP/MISF, to find the minimum value of both of them. And for a solution which require higher accuracy, we propose the GA Algorithm which selected population. When we need to calculate the exact solution, it is necessary for us to use DF/IHS to calculate the results.

5 References

References and Notes

1. Bean, James C. ”Genetic algorithms and random keys for sequencing and optimization.” ORSA journal on computing 6.2 (1994): 154-160.

2. Tobita, Takao, and Hironori Kasahara. ”A standard task graph set for fair evaluation of multiprocessor scheduling algorithms.” Journal of Scheduling 5.5 (2002): 379-394.

3. Nakamura Asuka, and Maekawa Hitaka ”Algorithm for reducing the number of search nodes in the exact solution solution of task scheduling problem” Programming (PRO) 7.1 (2014): 1-9.

6 Appendix

6.1 Python Code to Calculate Critical Path

```
def FindCP(inst):
        CP = [0]*len(inst[0])
        TaskNext =  [[] for i in range(len(inst[0]))]
        for t in range(len(inst[0])):
                for k in inst[1][t]:
                        if k != 0 and t != 0:
                                TaskNext[k-1].append(t)

        CPboolean = [False]*(len(inst[0])-1)
        CPboolean.append(True)

        while CPboolean != [True]*len(inst[0]):
                for t in range(len(inst[0])-1):
                        if (sum(CPboolean[g] for g in TaskNext[t]) == len(
                                if TaskNext[t] != []:
                                        CP[t] = max(CP[g] for g in TaskNex
                                else:
                                        CP[t] = inst[0][t]
                        CPboolean[t] = True
        print (max(CP))
        return CP
```

6.2 Python Code to Evaluate

```
def CalcFit(inst, gene):
        petime = [0]*PEnum
        starttime = [0]*(len(inst[0])-1)
        endtime = [0]*(len(inst[0])-1)
        parents = [[]for i in range((len(inst[0]))-1)]
        for i in range ((len(inst[0]))-1):
                parents[i] = inst[1][i]


        genelist = []
        now = 0
        for s in range(len(gene)):
                genelist.append(gene[s][1])

        for t in genelist:
                for p in parents[t-1]:
                        if p!= 0:
                                starttime[t-1] = max(starttime[t-1],endtime
                        else:
                                starttime[t-1] = petime[gene[now][0]-1]
                endtime[t-1] = starttime[t-1] + inst[0][t-1]
                petime[gene[now][0]-1] = endtime[t-1]
                now += 1
        return(petime)
```

6.3 Python Code of GA Algorithm

```
def Genetic(inst, ps):
        population = []
        time = 100000
        for k in range (ps): #Generate Init. Population
                undonetask= list(range(1,len(inst[0])))
                readytask = []
                finishtask = [0]
                gene = [] #First Digit is Task, 2nd is Machine
                while len(undonetask) > 0:
                    for j in undonetask:
                        if set(inst[1][j-1]).issubset(set(finishtas
                            readytask.append(j)
                            readytask = list(set(readytask))
                            shuffle(readytask)
                    gene.append([randint(1,PEnum),readytask[0]])
                    undonetask.remove(readytask[0])
                    finishtask.append(readytask[0])
                    readytask.remove(readytask[0])
                genewithtime = Gene(gene,CalcFit(inst,gene))
                population.append(genewithtime)
        for t in range(ps):
                population.append(Gene(Crossover(inst, population[t].info,
                population.append(Gene(Crossover(inst, population[t].info,
                population.append(Gene(Crossover(inst, population[t].info,
                population.append(Gene(Crossover(inst, population[t].info,
        population.sort(key = lambda x:x.time)
        print(max(population[0].time))



def CalcFit(inst, gene):
        petime = [0]*PEnum
        starttime = [0]*(len(inst[0])-1)
        endtime = [0]*(len(inst[0])-1)
        parents = [[]for i in range((len(inst[0]))-1)]
        for i in range ((len(inst[0]))-1):
                parents[i] = inst[1][i]



        genelist = []
        now = 0
        for s in range(len(gene)):
                genelist.append(gene[s][1])

        for t in genelist:
                for p in parents[t-1]:
                        if p!= 0:
                                starttime[t-1] = max(starttime[t-1],endtime
                        else:
                                starttime[t-1] = petime[gene[now][0]-1]
```

```
            endtime[t-1] = starttime[t-1] + inst[0][t-1]
            petime[gene[now][0]-1] = endtime[t-1]
            now += 1
        return(petime)

def Crossover(inst, gene1, gene2):
        length = len(inst[0])-1
        t = randint(0,length-1)
        geneson1 = []
        geneson2 = []
        geneson3 = []
        geneson4 = []
        for i in range(length):
                if i<t:
                        geneson1.append(gene1[i])
                        geneson2.append(gene2[i])
                        geneson3.append([gene2[i][0],gene1[i][1]])
                        geneson4.append([gene1[i][0],gene2[i][1]])
                elif i>=t:
                        geneson3.append(gene1[i])
                        geneson4.append(gene2[i])
                        geneson1.append([gene2[i][0],gene1[i][1]])
                        geneson2.append([gene1[i][0],gene2[i][1]])
        return [geneson1,geneson2,geneson3,geneson4]
```