Codelet Scheduling by Genetic Algorithm

Songwen Pei, Jinkai Wang, Wenyang Cui, Linhua Jiang
Shanghai Key Lab of Modern Optical Systems
University of Shanghai for Science and Technology, Shanghai 200093, China
E-mail: swpei@usst.edu.cn

Tongsheng Geng, Jean-Luc Gaudiot
Parallel Systems and Computer Architecture Lab
University of California
Irvine, CA 92697, USA
Email: {tgeng, gaudiot}@uci.edu

Stéphane Zuckerman
Computer Architecture & Parallel Systems Laboratory
University of Delaware
140 Evans Hall Newark, DE 19716
Email: szuckerm@udel.edu

***Abstract*—Codelet model is a fine-grained, event-driven hybrid parallel model inspired by dataflow, whose performance depends on the scheduling policy. How to design optimal codelet scheduling policy based on the features of tasks is important to the codelet-based system performance. In this paper, we propose an adaptive codelet scheduling policy by combing "pure" genetic algorithm for tasks with complex dependencies. It is verified that the policy is effective based on bunches of experimental results.**

Keyword—dataflow, Codelet model, task scheduling, genetic algorithm

I. Introduction

Dataflow model is a natural, data-driven parallel execution model [1, 2], featured by good elasticity for algorithm, strong scalability for program, and high performance for machine. It attracts great attentions from academia and industry, and is considered as an upgrading scheme of traditional parallel computing models based on the sequential von Neumann model, such as OpenMP [3] and POSIX threading models [4]. The dataflow model offers a simple solution to achieve high performance by isolating execution and expressing exact producer-consumer relations. However, the fine-grained executing way of dataflow model is prone to poor temporal and spatial locality. It will occur to extra overhead of computing and memory access, and will consume much energy due to frequently data communication. Then, a hybrid von Neumann/dataflow execution model [5, 6] is emerging in the field of dataflow model. Codelet model is one of the most important hybrid dataflow models [7, 8].

Suettlerlein et al. have developed the Codelet model, which is a fine-grained, event-driven parallel execution model based on the firing rules of dataflow [9]. In the codelet model, codelet is a basic compute unit which contains a set of machine instructions. Similar to a daflow actor, a codelet can be fired once all its events (i.e. dependencies) have occurred. The Codelet model uses Threaded Procedures (TPs) acted as the container of codelets to make up for the poor locality of dataflow model. Since the Codelet model inherits the advantages of its ancestors, it is applicable to deal with high performance parallel computing tasks in the field of big data. Codelet-based runtime system [10-12] is a software system designed to fill in the gap between the hardware and the Codelet model. The primary mission of runtime system is distributing and scheduling codelets. In this paper, we use DARTS (Delaware Runtime System) [13] as the runtime system implementation of the codelet model.

Scheduling policy determines the computing performance of an application in the parallel system [14]. In order to enhance the execution efficiency of the parallel tasks by Codelet model, we propose a codelet scheduling policy named as Genetic Policy (GP) mainly focus on task's execution time. GP is based on pure genetic algorithm (PGA) [15] which uses a two-dimensional matrix to implement a schedule [16]. The PGA is a heuristic method for finding an approximate optimal schedule in the multiprocessor scheduling problem. While in this paper, PGA is further improved in the generation of initial population and the operation of genetic operators, and combines with the Codelet model. Experiments will be in form of simulation, using randomly generated directed acyclic task graph to simulate the process of the task distribution and scheduling in the Codelet model.

This paper is organized as follows. Section II provides the necessary background including Codelet model, DARTS and the genetic algorithm. Section III defines the scheduling problem of the Codelet model. Section IV describes the content and the structure of the algorithm. Section V and section VI introduce the key steps of the algorithm in detail. Section VII presents the experimental content and the simulation results. Finally, section VIII summarizes the whole article.

II. Background

A. Codelet model

Codelet is a basic execution unit composed of a serial of machine instructions. A task in the Codelet model can be partitioned into many interconnected codelets[17]. Each codelet is controlled by a synchronous dependency tracking interface. A codelet becomes enable if all the events of the codelet have been satisfied. Then, the codelet will be distributed to a target processing unit according to scheduling policy, and it will be executed until hardware resources are available.

The codelets connect to each other by events, which forms a codelet graph (CDG) similar to the dataflow graph (DFG) [18, 19]. As the container of codelets, TP holds multiple codelets and allocates space for shared data. The CDG shown

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.231

in Fig. 1 can be seen as a directed acyclic graph (DAG) that consists of codelets, events and tokens. Codelets in the graph are linked together based on their data dependencies.

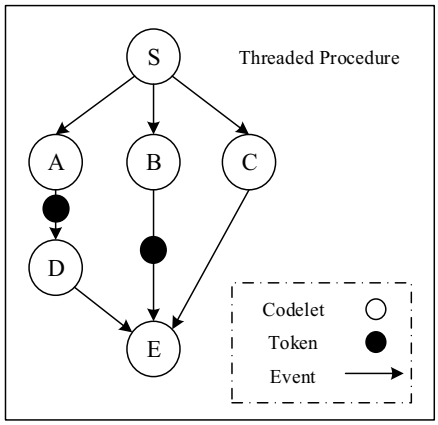

Fig. 1. A codelet graph (CDG)

B. DARTS runtime

The DARTS runtime is an implementation of the codelet execution model. The main functions of the DARTS runtime are allocating hardware resources, loading and executing the codelets based on the input of system configuration. Schedulers are the most vital components of the DARTS runtime, responsible for scheduling codelets and TPs, signaling events to the specified codelet and firing it. As shown in Fig. 2, each core in the system has a scheduler (hyper-threaded cores may have more than one)[13]. Moreover, schedulers in a cluster can be divided into threaded procedure scheduler (TPS) and codelet scheduler (CDS). TPS is mainly responsible for loading balance of TPs and codelets, and executing codelets in spare time. Each TPS contains two pools, namely TP Pool (TPP) and Ready Pool (RP), used to store TPs and enabled codelets separately. CDS acts as a computing unit, responsible for performing the enabled codelet code. Each CDS contains a RP for storing enabled codelets. In general, a cluster consists of a TPS and several CDS, and the communication between different clusters is performed by TPS. This paper only considers the codelet scheduling in a single cluster.

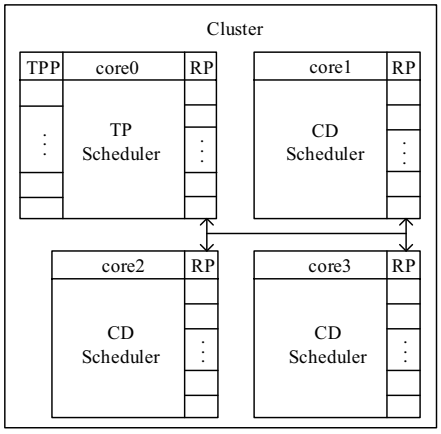

Fig. 2. Abstract machine model [13]

The mechanism of carrying out a task in the Codelet model is based on the scheduling policy. DARTS implements three main policies including static, dynamic and work stealing [13]. Default static policy distributes enabled codelets to each scheduler successively with round robin. However, the static policy can set the codelet metadata by programmer to specify which scheduler executes the codelet. Dynamic policy puts all the enabled codelets into the RP of the TPS, and the schedulers in free time take codelets from the TPS independently. Steal policy is similar to dynamic policy. All scheduler with steal policy in a cluster share the enabled codelets, while the difference is that the idle schedulers attempt to steal a codelet from other schedulers randomly, rather than to acquire work from the TPS uniformly in the style of dynamic policy. And these three scheduling policies are as experimental comparisons in the Section VIII.

C. Genetic Algorithm

Genetic algorithm (GA) is a search technique used in computing to find true or approximate solutions to optimization and search problems [20]. GA can find the optimal solution from the solution space quickly owing to its good global search ability, and provide a simple but effective solution to solve complex problems that traditional mathematics are difficult to solve, especially for the optimization problem of task allocating and scheduling [21].

Hou et al. [15] realized the task scheduling by PGA. PGA is an efficient method solving the problem of multiprocessors, and is also appropriate to the codelet scheduling for its intuitive representation of the search node and the simple genetic operators based on the task graph. This paper presents a novel codelet scheduling policy GP, upon the improved PGA.

III. Model and Definition

Since the Codelet model is expressed by CDG, a task executed by the model can be regarded as a set of interconnected codelets [22]. In DARTS, enabled codelets are stored in the RPs of schedulers in the form of queues, so codelets in a CDG can be distributed and scheduled into several queues. Each queue of a scheduler presents the execution order of codelets. To cope with the problems of codelet schedule, we will give the flowing definitions:

(1) G is a CDG that can be expressed by $G = <V, E, Rt, d, id>$, where are:

a) V represents the set of nodes, $V = \{v_0, v_1, \ldots, v_{n-1}\}$, where n is the number of nodes. A node in the CDG represents a codelet;

b) E represents the set of directed dependencies among nodes. $<v_i, v_j>$ represents the dependency between node v_i and v_j, which means node v_i must be completed before v_j can be initiated;

c) Rt represents the runtime of a node. The time cost by running the node v_i can be written as $Rt(v_i)$;

d) d represents the depth of a node in the graph, which is the largest number of edges from an entry node to node itself [23]. The depth of the node v_i can be defined as:

$$d(v_i) = \begin{cases} 0, \mathrm{PRED}(v_i) = \emptyset \\ \mathrm{MAX}(\mathrm{PRED}(v_i)) + 1, \mathrm{ELSE} \end{cases}, 0 <= d <= d_{max},$$

where d_{max} is the depth of the final node and is also the same as the maximum of all nodes' depth. Depth is an important parameter to generate the initial population and judge the legitimacy of a schedule (See Section VI).

e) *id* represents the metadata of a node, corresponding to the id of the scheduler which stores the codelet in the cluster.

As shown in Fig. 3 (a), a task containing multiple subtasks is expressed as a CDG containing the same number of nodes. Each node has its private information including runtime, depth and metadata. The arrows represent the dependencies among the nodes, and indicate the fire method of codelets. For example, the node *v6* will be enabled only if all its precursor nodes *v3*, *v4* and *v5* have been fired.

(2) S is a codelet schedule (e.g. a solution in GA) of a CDG and can be represented by $S = <C, Q>$, where are:

a) C represents the set of schedulers, $C = \{c_0, c_1, \ldots, c_{m-1}\}$, where m is the number of schedulers in the cluster;

b) Q represents the set of codelet queues, $Q = \{q_0, q_1, \ldots, q_{m-1}\}$;

c) q_i represents the execution ordering of the codelets in scheduler c_i, $q_i = \{ v_{i0}, v_{i1}, \ldots, v_{i(k-1)} \}$, where k is the length of queue q_i;

d) A schedule can be converted into a two-dimensional structure that each schedule has serval queues and each queue contains a number of codelets.

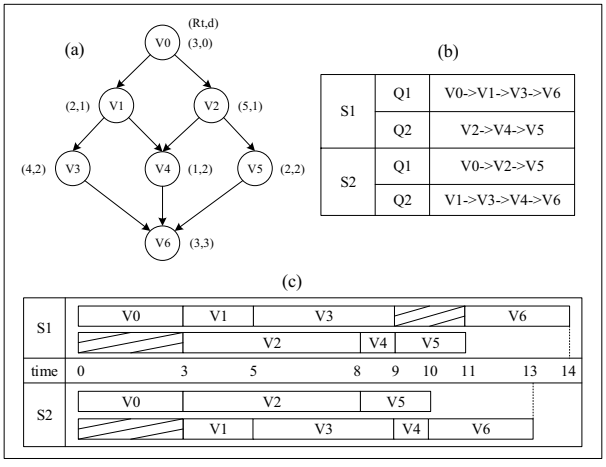

Fig. 3. Codelet scheduling

Each scheduler executes codelets by the ordering of its codelet queue. But this ordering may disturb the original dependencies of the CDG and even cause the task can't be completed. For example, in Fig. 3 (a), the task couldn't be performed in the Codelet model if a scheduler executes codelet queue {V0, V3, V1, V6} due to the mutual reverse dependency between node V1 and V3. According to the basic ideas from PGA[15], restricting the encoding scheme by depth can produce valid codelet queue. So we define that a codelet queue is legal when its codelets are sorted by ascending order based on depth. Moreover, a schedule can be defined legally as all its codelet queues are satisfied with the depth-ordering condition.

As shown in Fig. 3 (b), there are two schedules S1 and S2, each of which has two codelet queues indicating the execution ordering of codelets. Schedule S1 can be expressed as {{V0, V1, V3, V6}, {V2, V4, V5}}, while schedule S2 can be expressed as {{V0, V2, V5}, {V1, V3, V4, V6}}. These two schedules are both legal because their codelets' depth in each codelet queue increases gradually.

(3) $T(x)$ is the finishing time that all codelets in schedule x have been ran, and $T_x(q_i)$ is the total time spent on executing all codelets in codelet queue q_i. There are $T(x) = \mathrm{MAX}(T_x(q_i))$, $\forall i$ ($0 \leqslant i \leqslant m$-1). The finishing time of schedule S1 and S2 are shown in Fig. 3 (c) as T(S1) = 14, T(S2) = 13.

The goal of the codelet scheduling is to find an optimal schedule that makes codelets execute parallel in multiple schedulers in terms of the established order to minimize the task execution time.

IV. Algorithm Framework

Genetic algorithm based on task scheduling should have three principal elements, namely fitness function, population initialization, and evolution method.

We elect the finishing time of the task performed in the Codelet model as the metric to measure the quality of a schedule. The finishing time is also the parameter used for calculating the fitness value.

According to the representation of the schedule expressed in section III, several random generated queues contained the codelets arrayed in ascending order of their depth make up an initial solution (e.g. a codelet schedule). Furthermore, a specific number of initial solutions make up the initial population of the GA .The optimal solution is evolved from the initial population by genetic operators.

The genetic operators in the evolution method are selection, crossover and mutation. In an iteration of the GA, parent solutions perform several genetic operators to generate child solutions. After number of evolutions, the final optimal solution will be obtained until the convergent criterion of the algorithm is met.

Fig. 4 shows the structure of the algorithm, and all specific steps are as follows:

(1) Convert a task into a CDG and identify the depth of each codelet;

(2) Generate the initial population POP(t), $t = 0$, where t is the number of evolutional generation (e.g. the number of iterations);

(3) Compute the fitness value of each solution in the initial population;

(4) Do steps (5)-(9) until the algorithm is convergent;

(5) Select parent solutions by the roulette wheel;

(6) Perform external and internal crossover with probability P_e and 1-P_{ic} respectively, and the sum of P_{ec} and P_{ic} is equal to 1;

(7) Perform mutation with probability P_m;

(8) Retain the optimal solution in the old population by the elite strategy;

(9) Generate a new population as POP(t), $t = t + 1$;

(10) Output the approximate optimal solution whose fitness value is the highest in the last generated population.

V. Fitness Function

Fitness function is the evolutionary motivation of the GA and is used to select the superior solutions and eliminating the inferior ones. The structure of the fitness function not only determines whether the optimal solution generates successfully, but also affects the convergence speed of the algorithm.

The fitness value calculated by the fitness function is the principal index of a solution, and is used in the generation of the initial population and the control of the genetic operators. The fitness value must be a non-negative number. The larger the fitness value is, the better the solution will be.

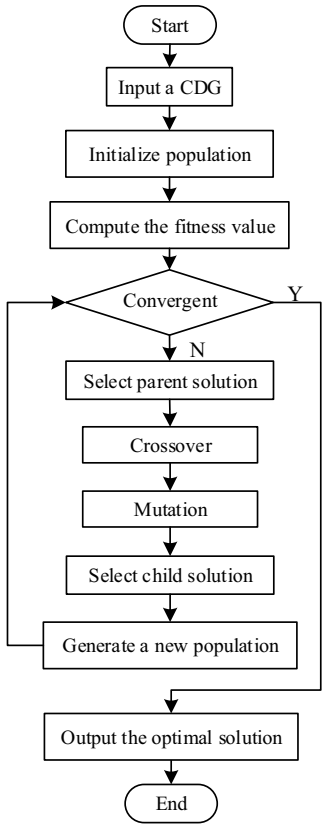

Fig. 4. Algorithm framework

For the problem of the codelet scheduling, there are many factors such as finishing time, throughout, locality and energy efficiency that should be considered as the parameters of the fitness function. In this paper, the fitness function is based on the finishing time of a schedule. The definition of the finishing time is presented in the section III. Since the finishing time of the optimal schedule should be the minimum, we define the fitness value1 as follows:

$$F(x) = c - T(x),$$

where c is a large enough constant, and $T(x)$ is the finishing time of schedule x. c is defined as the sum of all codelets' runtime, e.g. $c = \sum_{i=0}^{n-1} Rt(v_i)$. The finishing time $T(x)$ of optimal schedule is smaller than that of other schedules. Therefore, the fitness value of the optimal schedule is the largest of all schedules.

As shown in the Fig. 3, the finishing time of schedule *S1* is 14, and the constant c is 20. As a result, the fitness value of schedule S1 is 6.

VI. Initial Population

Because the Codelet model is a parallel model, distributing codelets to schedulers in a cluster evenly can reduce the task execution time. Moreover, limited by the dependencies among the codelets, all the codelet queues in the initial solution should be proved legal. In order to deal with the distributing problem of the codelets, we will give the following definitions as follows:

(1) m is the number of schedulers;

(2) $n(i)$ is the number of codelets where their depth are i, then all the codelets in the CDG are partitioned into d_{max} +1 subsets;

(3) $min(i)$ is the minimum number of codelets where their depth are i, and they are assigned to a scheduler, $min(i) = \lfloor n(i) / m \rfloor$;

(4) Every scheduler assigns [$min(i)$, $min(i)$+1] codelets whose depth are i, and the total number of all schedulers' codelets whose depth are i is equal to $n(i)$;

(5) Every codelet appears only once in a schedule.

The method of generating the initial solution is putting the codelets of each subset to m schedulers uniformly, and then sorting the codelets in each codelet queue in ascending order by their depth. For generating a native legal schedule, codelets are assigned to codelet queues successively with the growth of the depth.

Furthermore, the quality of the initial population determines the convergence rate of the genetic algorithm. Thus, we select M excellent solutions from N ($M < N$) generated solutions according to the fitness value as the initial population, to increase the solving speed and prevent premature convergence.

Accordingly, the process of generating the initial population has the flowing steps:

(1) Take the start codelet to a random scheduler, and set depth i as i whose initial value is 1;

(2) Do steps (3)-(5) until i is equal to d_{max};

(3) Count the number of codelets whose depth are i, and

compute the value of *min*(*i*);

(4) Distribute *min*(*i*) codelets whose depth are *i* to each scheduler randomly;

(5) Put remain codelets to schedulers randomly that make the number of distributed codelets in each scheduler no more than *min*(*i*)+1, *i* = *i*+1;

(6) Do steps (1)-(5) until *N* initial solutions are generated;

(7) Compute the fitness value of *N* initial solutions

(8) Pick *M* solutions with higher fitness value from *N* initial solutions as the initial population.

VII. Genetic Operator

In GA, the evolution of population is performed by a serial of genetic operators including selection, crossover and mutation. The task of these genetic operators is performing certain operations for solutions of a population by evaluating the fitness value, so as to realize the evolutionary process and generate better solutions. From the viewpoint of optimization search, genetic operators can make the solutions of the scheduling problem optimize continually.

A. Select Operator

Selection mechanism is available to keep excellent solutions and eliminate inferior ones by fitness value on every iteration[24]. We adopt the biased roulette wheel to implement the operation of selection, that is, parent solution is picked with the probability proportional to its fitness value. However, the roulette wheel method may cause statistical error because of its random choices. So we use the elite strategy as the retention mechanism to ensure the fitness value of the optimal solution monotonically increases. The key thought of the elite strategy is to keep the most excellent solution in the old population to the next generation. And the specific operations are as follows:

(1) The roulette wheel: firstly, compute the fitness value of each solution from the old population, and set the sum of all the fitness value as SUM; then calculate the ratio of each solution's fitness value and SUM as the select probability of the solution; finally, select a certain number of solutions according to the probability as the objects of the next operator.

(2) The elite strategy: firstly, retain the best solution with the highest fitness value in the old population, and this solution is denoted by B(*t*), where B(t) = Best(POP(*t*)); then find the worst solution with the lowest fitness value in the new population, and set this solution as W(*t*+1), where W(t+1) = Worst(POP(*t*+1)); finally, replace W(*t*+1) with B(*t*).

B. Crossover Operator

Crossover can generate new solutions directly to expand the search field, so it is the most important operator in the GA. The process of the crossover operator is that exchange the portions of two selected parent individuals (schedule or codelet queue) to form two new child individuals. The crossover operator in this algorithm has two steps, namely selection of the crossover point and exchange operation.

Limited by the complex dependencies among codelets, traditional selection mechanism of the crossover point may generate an illegal individual. Therefore, the crossover point whose depth is *d'* which needs to meet the following two conditions:

(1) The depth of the codelets next to the crossover point are different from *d'*;

(2) The depth of the codelets in front of the crossover point should not be larger than *d'*.

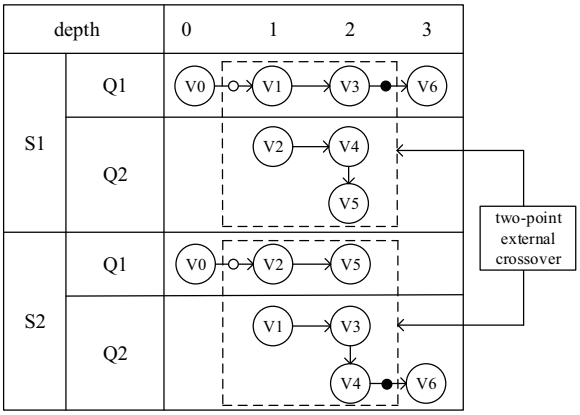

Fig. 5. External crossover ($d'_1 = 0$, $d'_2 = 2$)

To further expand the search field of the GA, we adopt an improved hybrid exchange operation instead of the simple one-point (e.g. one crossover point) exchange operation [24]. Two random crossover points are used in this exchange operation, and whether their depth are same or not determines which exchange method will be executed. The operands of the exchange operation are two parent individuals in the old population. The details of the hybrid exchange operation are:

(1) If the depths of the two random crossover points are the same, perform one-point exchange operation that each operation individual has only one crossover point:

(a) Cut each operation individual into two parts by the crossover point;

(b) Retain the top halves of the two individuals before the crossover pointpoints;

(c) Exchange the bottom halves of the two individuals after the crossover point.

(2) If the depth of the two random crossover points is different, perform two-point exchange operation that each individual has two crossover points:

(a) Cut the each operation individual into three parts by the two different crossover points;

(b) Exchange the middle parts of the two individuals between the crossover points;

(c) Retain the top and bottom parts of the two individuals except of the middle parts.

As shown in Fig. 6, schedule S1 performs the two-point exchange operation because the depth of the two crossover points is different. The operation individuals in this example

are codelet queues. Accordingly, the exchange parts are codelet V3 in codelet queue Q1 and codelet V4, V5 in codelet queue Q2, while the other codelet of the two queues remain unchanged. On the other hand, schedule S2 performs the one-point exchange operation because of the same depth of two crossover points. The exchange parts are codelet V5 in codelet queue Q1 and codelet V3, V4, V6 in codelet queue Q2.

Furthermore, crossover operator can be divided into external crossover operator and internal crossover operator based on the operands.

1) External Crossover Operator

The objects of the external operation are two parent schedules, and the exchange portions are the parts of all codelet queues in two schedules. The algorithm is shown in Algorithm 1:

Algorithm 1: External Crossover

Input: Old population *old_pop*

Output: Two new schedules in the new population *new_pop*

Data: d_{max} is the maximum depth of all codelets

M is the size of population

Begin

EC1. [Select crossover schedules.] Use the roulette wheel to generate two different numbers, *c1* and *c2*, between 0 and *M*, as the indexes of *old_pop* (e.g. an array of schedules), and the corresponding schedules are *sch1* and *sch2* respectively.

EC2. [Select crossover points.] Pick two random numbers d'_1 and d'_2 as the depth of crossover points, $0 <= d'_1, d'_2 <= d_{max}$.

EC3. [Choose exchange operation.] If d'_1 is equal to d'_2, do step EC4, else do step EC5.

EC4. [One-point exchange operation.] Perform the one-point exchange operation that swaps the bottom halves of *sch1* and *sch2* after the crossover points.

EC5. [Two-point exchange operation.] Perform the two-point exchange operation that swaps the middle parts of *sch1* and *sch2* between the crossover points.

End

As shown in Fig. 5, Schedule S1 and S2 execute the two-point external crossover operation, and the two dotted boxes contain parts of the codelet queues which are the portions to be inter-exchanged between S1 and S2.

2) Internal Crossover Operator

The objects of the internal operation are two random codelet queues in a schedule, and the exchange portions are the sections of the two lists. The algorithm is shown in Algorithm 2:

Algorithm 2: Internal Crossover

Input: Old population *old_pop*

Output: A new schedule in the new population *new_pop*

Data: d_{max} is the maximum depth of all codelets

m is the number of schedulers

M is the size of population

Begin

IC1. [Select operation schedule.] Use the roulette wheel to generate a random numbers, *c*, between 0 and *M*, as the selected index of *old_pop*, and the corresponding schedule is *sch*.

IC2. [Select crossover queues.] Randomly generate two different numbers *c1* and *c2*, between 0 and *m*, as the selected indexes of schedule *sch* (e.g. an array of codelet queues), and the corresponding queues are *que1* and *que2* respectively.

IC3. [Select crossover points.] Pick two random numbers d'_1 and d'_2 as the depth of crossover points, $0 <= d'_1, d'_2 <= d_{max}$.

IC4. [Choose exchange operation.] If d'_1 is equal to d'_2, do step EC4, else do step EC5.

IC5. [One-point exchange operation.] Perform the one-point exchange operation that swaps the bottom halves of *que1* and *que2* after the crossover points.

IC6. [Two-point exchange operation.] Perform the two-point exchange operation that swaps the middle parts of *que1* and *que2* between the crossover points.

End

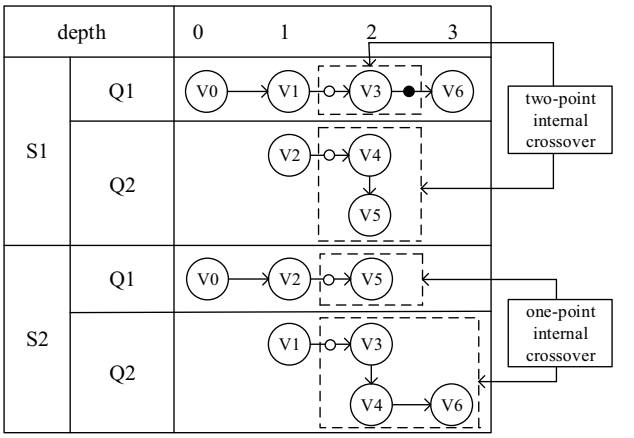

Fig. 6. Internal crossover ($d'_1 = 0$, $d'_2 = 2$ in S1; $d'_1 = d'_2 = 1$ in S2)

Fig. 6 shows two types of internal crossover operation. Schedule S1 executes the one-point internal operation with the upper two dotted boxes as its exchange portions. While schedule S2 executes the two-point internal operation with the lower two dotted boxes as its exchange portions.

C. Mutation Operator

Mutation is the assisted method used to maintain the diversity of population and speed up the convergence rate of the optimal solution. Mutation operator adopts traditional exchange operation that swaps two codelets with same depth in a schedule. In order to guarantee the dependencies of codelets not to be destroyed, the exchange codelets must have

same depth. The specific process is as follow:

(1) Pick a random number d'', $0 < d'' < d_{max}$;

(2) Select two random different codelets where their depth is d'';

(3) Generate a new solution by exchanging the two codelets.

As shown in the Fig. 6, codelet V1 and V2 whose depth are both 1 in schedule S1 can exchange each other when performing mutation operation.

VIII. Experimental Results

Executing a task in the Codelet model can be denoted by a CDG. Thus, the experiment applies a randomly generated task graph to simulate the process of codelets' scheduling and running by the DARTS. In this simulation, we compare the genetic policy proposed in this paper with three scheduling policies introduced in Section II, i.e. static, dynamic and steal policy.

We use Intel Core i7-4720HQ (quad-core and eight-thread) as the simulation platform. The number of nodes in a task graph ranges from 100 to 1000. The out-degree of each node is a random number between 1 and 10. And the runtime of each node is a random number between 1 and 50. The parameters used by the GA are as flow:

(1) population scale POP = 10;

(2) external crossover probability $P_{ec} = 0.2$;

(3) internal crossover probability $P_{ic} = 0.8$;

(4) mutation probability $P_m = 0.05$;

(5) maximal number of iterations MAX = 2000.

Table I through IV summarize the finishing time of the random task graphs, with different number of codelets，which are executed by four scheduling policies such as static, dynamic, steal and genetic on different multiprocessor configurations. From Table I to the Table IV, we find that the task execution efficiency of the genetic policy is superior to the other three ones.

TABLE I. Comparison of 2 schedulers

Codelet Number	Static Policy	Dynamic Policy	Steal Policy	Genetic Policy
100	2616	1415	2615	1378
200	5060	2630	5059	2595
400	10285	5253	10285	5216
600	15408	7841	15404	7790
800	20606	10484	20605	10432

TABLE II. Comparison of 4 schedulers

Codelet Number	Static Policy	Dynamic Policy	Steal Policy	Genetic Policy
100	1098	844	1015	803
200	2045	1571	1928	1497
400	3810	2787	3566	2715
600	5718	4255	5409	4170
800	7427	5494	7010	5418

TABLE III. Comparison of 6 schedulers

Codelet Number	Static Policy	Dynamic Policy	Steal Policy	Genetic Policy
100	861	755	791	712
200	1478	1260	1367	1212
400	2592	2121	2364	2047
600	3708	2979	3347	2914
800	4814	3868	4364	3809

TABLE IV. Comparison of 8 schedulers

Codelet Number	Static Policy	Dynamic Policy	Steal Policy	Genetic Policy
100	767	692	717	656
200	1245	1118	1181	1065
400	2125	1829	1962	1771
600	2921	2447	2788	2369
800	3763	3132	3440	3056

Fig.7, Fig.8, and Fig.9 respectively show the speedup of dynamic, static and steal policies with respect to genetic policy. Fig.10 shows the finishing time of the task graphs that contains 1024 codelets, and each task graph is executed on different numbers of schedulers by four scheduling policies.

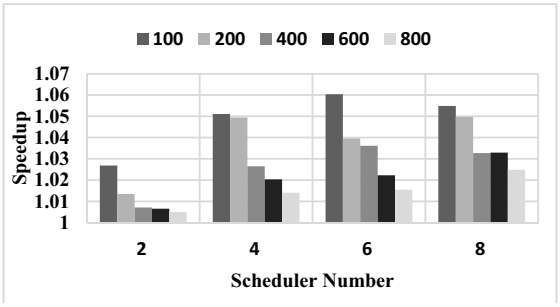

Fig. 7. Speedup (genetic vs. dynamic)

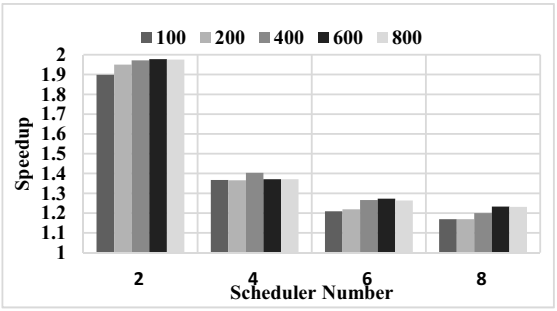

Fig. 8. Speedup (genetic vs. static)

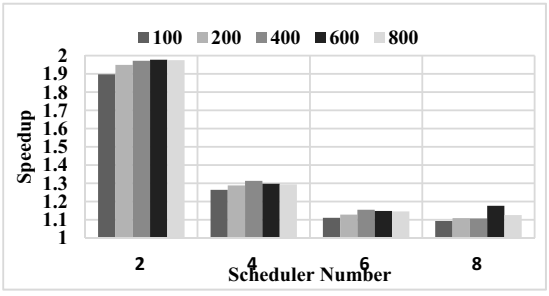

Fig. 9. Speedup (genetic vs. steal)

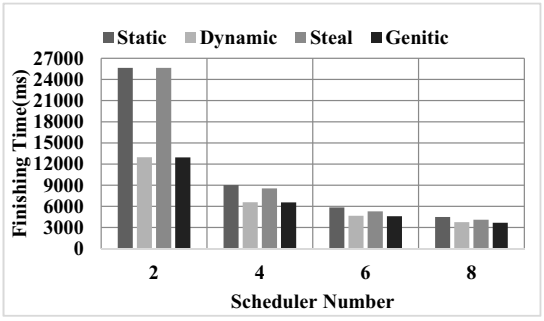

Fig.10. Finish time of executing 1024 codeletsh

As shown in the Fig.7, the speedup of using genetic policy generally increases as the number of scheduler increases. The reason is that, with the increase of the cluster size, the pressure and latency of queuing operations in dynamic policy increases. By contrast, the speedup of the genetic policy compared with the static policy (shown in Fig.8) and the steal policy (shown in Fig.9) decrease as the number of the schedulers increases. The reason is that the static and steal policy have strong randomness to get codelet, which leads to long idle periods of schedulers and poor execution efficiency of the system. However, as the number of schedulers increases, each scheduler will spend less time executing its codelet, and then the total execution time of static and steal policies reduces accordingly. Even so, the finishing time of the genetic policy is still smaller than that of the other policies, which verifies the feasibility and the effectiveness of the genetic policy.

IX. Conclusion

This paper focuses on scheduling policy of the Codelet model. We propose a new codelet scheduling policy, based on the standard genetic algorithm, which can reduce task execution time effectively. Random task graph is employed in the experiment to simulate the execution process of the tasks with different policies in the Codelet model. Simulation results show that the genetic policy introduced in this paper can obtain better efficiency than the other three scheduling policies in the DARTS, and thereby possesses strong superiority to deal with the problem of codelet scheduling.

Acknowledgment

We thank the anonymous reviewers for their invaluable comments. This work was partially funded by the Shanghai Municipal Natural Science Foundation (15ZR1428600), Shanghai Pujiang Program (16PJ1407600), the program for Professor of Special Appointment (Eastern Scholar) at Shanghai Institutions of Higher Learning, USST incubation project (15HJPY-MS02), and the National Science Foundation of United States (CCF-1065448, XPS-1439097).

Reference

[1] Dennis J B. First version of a data flow procedure language[C]//Programming Symposium. Springer Berlin Heidelberg, 1974: 362-376.

[2] Dennis J B. Data flow computer architecture[M]. Springer US, 2011.

[3] Dagum L, Enon R. OpenMP: an industry standard API for shared-memory programming[J]. Computational Science & Engineering, IEEE, 1998, 5(1): 46-55.

[4] Butenhof D R. Programming with POSIX threads[M]. Addison-Wesley Professional, 1997.

[5] Yazdanpanah F, Alvarez-Martinez C, Jimenez-Gonzalez D, et al. Hybrid dataflow/von-Neumann architectures[J]. Parallel and Distributed Systems, IEEE Transactions on, 2014, 25(6): 1489-1509.

[6] Grafe V G, Hoch J E, Davidson G S. Eps' 88: Combining the best features of von Neumann and dataflow computing[R]. Sandia National Labs., Albuquerque, NM (USA), 1989.

[7] Giorgi R, Badia R M, Bodin F, et al. TERAFLUX: Harnessing dataflow in next generation teradevices[J]. Microprocessors and Microsystems, 2014, 38(8): 976-990.

[8] Zuckerman S, Suetterlein J, Knauerhase R, et al. Using a codelet program execution model for exascale machines: position paper[C]//Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. ACM, 2011: 64-69.

[9] Suettlerlein J, Zuckerman S, Gao G R. An implementation of the Codelet model[M]//Euro-Par 2013 Parallel Processing. Springer Berlin Heidelberg, 2013: 633-644.

[10] Dennis J B. Fresh Breeze: a multiprocessor chip architecture guided by modular programming principles[J]. ACM SIGARCH Computer Architecture News, 2003, 31(1): 7-15.

[11] Gautier T, Besseron X, Pigeon L. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors[C]//Proceedings of the 2007 international workshop on Parallel symbolic computation. ACM, 2007: 15-23.

[12] Lauderdale C, Khan R. Towards a codelet-based runtime for exascale computing: position paper[C]//Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. ACM, 2012: 21-26.

[13] Suetterlein J. DARTS: a runtime based on the Codelet execution model[D]. University of Delaware, 2014.

[14] Nguyen T D, Vaswani R, Zahorjan J. Parallel application characterization for multiprocessor scheduling policy design[C]//Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, 1996: 175-199.

[15] Hou E S H, Ansari N, Ren H. A genetic algorithm for multiprocessor scheduling[J]. Parallel and Distributed Systems, IEEE Transactions on, 1994, 5(2): 113-120.

[16] Corrêa R C, Ferreira A, Rebreyend P. Scheduling multiprocessor tasks with genetic algorithms[J]. IEEE Transactions on Parallel and Distributed systems, 1999, 10(8): 825-837.

[17] Benten M S T, Sait S M. Genetic scheduling of task graphs[J]. International Journal of Electronics, 1994, 77(4): 401-415.

[18] Davis A L, Keller R M. Data flow program graphs[J]. 1982.

[19] Purna K M G, Bhatia D. Temporal partitioning and scheduling data flow graphs for reconfigurable computers[J]. Computers, IEEE Transactions on, 1999, 48(6): 579-590.

[20] Tang K S, Man K F, Kwong S, et al. Genetic algorithms and their applications[J]. Signal Processing Magazine, IEEE, 1996, 13(6): 22-37.

[21] Goldberg D E. Genetic algorithms in search optimization and machine learning[M]. Reading Menlo Park: Addison-wesley, 1989.

[22] Sarkar V. Partitioning and scheduling parallel programs for execution on multiprocessors[R]. Stanford Univ., CA (USA), 1987.

[23] Kwok Y K, Ahmad I. Static scheduling algorithms for allocating directed task graphs to multiprocessors[J]. ACM Computing Surveys (CSUR), 1999, 31(4): 406-471.

[24] Potts J C, Giddens T D, Yadav S B. The development and evaluation of an improved genetic algorithm based on migration and artificial selection[J]. Systems, Man and Cybernetics, IEEE Transactions on, 1994, 24(1): 73-86.