A Hybrid Heuristic-Genetic Algorithm with Adaptive Parameters for Static Task Scheduling in Heterogeneous Computing System

Shan DING[1], Jinhui WU[1]
[1]College of Computer Science and Engineering
Northeastern University
Shenyang, China
dingshan@cse.neu.edu.cn

Guoqi XIE[2] and Gang ZENG[3]
[2]College of Computer Science and Electronic Engineering,
Hunan University, Changsha, China, xgqman@hnu.edu.cn
[3]Graduate School of Engineering
Nagoya University, Nagoya, Japan, sogo@ertl.jp

***Abstract*—Task scheduling is critical for obtaining a high performance schedule in heterogeneous computing systems (HCS) and searching an optimal scheduling solution has been shown to be NP-complete. In this paper, a hybrid heuristic-genetic algorithm with adaptive parameter (HGAAP) is proposed by combining a heuristic scheduling algorithm and a genetic algorithm. An existing common heuristic scheduling algorithm is utilized for generating the initial generation to speed up the convergence rate of HGAAP. The parameters of crossover probability and mutation probability are adaptive according to the current evolution status to promote the evolution and find better solution. Moreover, the redundant individuals in each generation are removed to keep the diversity of population during the iteration. The experimental results on randomly generated task sets validated that our algorithm can obtain better scheduling results than existing algorithms.**

Keywords— adaptive parameters; directed acyclic graph; genetic algorithms; heterogeneous computing systems; task scheduling

I. INTRODUCTION

A heterogeneous computing system (HCS) is composed of diverse sets of processors interconnected via a high speed network. Such systems can execute computationally intensive parallel and distributed applications. Generally, an application can be decomposed into a set of tasks with precedence constraints and it is usually modeled as a directed acyclic graph (DAG). In a DAG, the nodes represent tasks and the edges represent precedence constraints between tasks. By assigning the tasks to appropriate processors and executing them as the required order, the total execution time of the application, usually called the makespan, can be reduced. Hence, searching an efficient scheduling of an application on HCS is critical to achieve high performance.

The objective of a task scheduling algorithm is to search an efficient scheduling for an application so that the makespan of the application can be minimized and all precedence constraints between tasks can be met. This problem has been well-studied for many years and it has been proved to be a NP-complete problem[1]. In general, task scheduling algorithms can be divided into two categories: static task scheduling and dynamic task scheduling[2]. In this paper we focus on the static scheduling on HCS. For static scheduling algorithms, all needed information about the application must be known in advance. Although various algorithms have been proposed, they can only produce near-optimal solutions[3]. Hence, there is still room for the development of better task scheduling algorithms.

This paper proposes a hybrid heuristic-genetic algorithm with adaptive parameter (HGAAP) to further improve the performance of existing algorithms. HGAAP uses the existing well-studied heterogeneous earliest finish time (HEFT) heuristic algorithm[3] to generate the initial scheduling solution, which is inserted into the initial population of HGAAP to enhance the convergence rate. For applications with various DAG characteristics, fixed crossover probability and mutation probability cannot always lead to the optimal solutions. Therefore, in HGAAP, the values of both crossover probability and mutation probability are adaptive according to the current evolution status. Moreover, the redundant individuals of the population are removed during the iteration, which can enhance the diversity of the population.

The rest of the paper is organized as follows: Section II reviews the related work. Section III gives an overview of the problem. Section IV presents the proposed algorithm. Section V compares the performance of the proposed algorithm with existing algorithms. We conclude the paper in Section VI.

II. RELATED WORK

The related work on task scheduling algorithms can be classified into three main groups: heuristic scheduling algorithms, meta-heuristic scheduling algorithms, and hybrid scheduling algorithms[3].

Heuristic algorithms usually can search a near optimal solution in polynomial time. These algorithms follow a particular rule, searching one path in the solution space and ignoring others[2]. There are mainly three kinds of heuristic algorithms: list scheduling, cluster scheduling and duplication-based scheduling[2, 3]. List scheduling is the most popular heuristic algorithms. The HEFT algorithm[3] and LDCP algorithm[6] are list scheduling algorithms. These algorithms usually have two phases: task prioritizing and

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.310

processor selection. In task prioritizing phase, each task is assigned a priority and a task list is generated by sorting tasks in descending order of their priorities. In processor selection phase, the task with the highest priority is removed from the list and assigned to a fittest processor. Parallelization methods are used in clustering algorithms to balance communication costs of tasks. Tasks with heavily communicating cost are assigned to the same cluster, and the number of clusters is assumed to be unlimited. Then clusters are mapped to the processors. If the number of clusters is bigger than that of processors, clusters are merged so that its number is equal to or less than that of processors. The execution order of tasks in each processor is decided according to particular criteria. An example of clustering algorithms is introduced in [7]. The duplication-based scheduling algorithms execute the key tasks on more than one processor so that communication costs between tasks can be reduced. These algorithms have two steps. Firstly, a clustering or list scheduling algorithm is used to generate an initial schedule. Then the tasks that have a large number of dependent tasks are selected and assigned to processors in which their dependent tasks have been assigned. An example of duplication-based algorithm is introduced in [8].

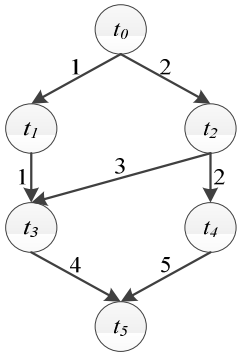
Meta-heuristic scheduling algorithms are guided-random-search-based algorithms that incorporate a combinatorial process for searching schedule solutions. There are many kinds of meta-heuristic algorithms. GA[5] is the most popular. In GA, the possible schedule solutions are encoded into chromosomes. It mimics the principles of evolution and natural genetics to operate the population to evolve a new population. After sufficient number of generations, the best individual is selected from the population as the schedule solution.

Heuristic algorithms search one path in the search space and ignore others. In multimodal problems, it always obtains a near-optimal solution. Meta-heuristic scheduling algorithms overcome this problem by incorporating a combinatoric process in the search for schedule solutions. However, meta-heuristic algorithms usually need more time to obtain the results. Therefore, to speed up the convergence time and obtain more efficient solutions, hybrid scheduling algorithms that combine heuristic and meta-heuristic algorithms have been introduced recently. For example, H2GS[4] is a hybrid algorithm. It combines the Longest Dynamic Critical Path (LDCP)[6] algorithm and the Genetic Algorithm for Scheduling (GAS)[5]. LDCP is used to generate a quality initial schedule solution for GAS to speed up the convergence. GAS operates genetic operators to the population to generate a better schedule solution.

III .PROBLEM DESCRIPTION

The task scheduling problem for an application on HCS is to minimize the makespan by effectively allocating tasks to processors and meeting all precedence constraints.

Generally, an application can be represented by a DAG. A DAG is defined by a tuple ($\boldsymbol{T}$, $\boldsymbol{E}$), where $\boldsymbol{T}$ is a set of $\boldsymbol{n}$ tasks and $\boldsymbol{E}$ is a set of $\boldsymbol{e}$ edges. The $\boldsymbol{i}$ task in the set of tasks is represented by $\boldsymbol{t_i}$. When $\boldsymbol{t_i}$ is allocated to a processor, it must be executed sequentially in this processor without interruption. The edge started by $\boldsymbol{t_i}$ and ended up with $\boldsymbol{t_j}$ is represented by $\boldsymbol{e_{i,j}}$ that represents the precedence constraint between $\boldsymbol{t_i}$ and $\boldsymbol{t_j}$. Each edge $\boldsymbol{e_{i,j}}$ has a value that represents the communication time when transmitting data from $\boldsymbol{t_i}$ to $\boldsymbol{t_j}$. It also indicates that $\boldsymbol{t_j}$ should not be executed until $\boldsymbol{t_i}$ completes its execution. A task may have zero or more input tasks and output tasks. Only if all its input tasks complete, can the task be executed. A task with no input task is an entry task and a task with no output task is an exit task. The communication time between tasks allocated in the same processor is assumed to be zero.



(a) DAG

task	p_0	p_1
t_0	7	8
t_1	6	9
t_2	5	8
t_3	2	3
t_4	3	4
t_5	4	6

(b) Computation cost on two processors

Fig. 1. Example of a DAG application and computation cost table

The HCS is represented by $\boldsymbol{P}$, a set of $\boldsymbol{m}$ processors with different performances. The computation cost of tasks on different processors is stored in an $\boldsymbol{n}\times\boldsymbol{m}$ matrix $\boldsymbol{C}$. Each element $\boldsymbol{c_{i,j}}$ in $\boldsymbol{C}$ represents the computation cost of $\boldsymbol{t_i}$ on processor $\boldsymbol{p_j}$. This paper assumes that the computation cost of tasks in different processors is monotonic. In other words, if the computation cost of $\boldsymbol{t_i}$ on $\boldsymbol{p_j}$ is higher than that on $\boldsymbol{p_k}$, then the computation cost of any tasks on $\boldsymbol{p_j}$ is equal to or higher than that on $\boldsymbol{p_k}$. Fig.1 gives an example of an application consisting of six tasks and a HCS with two processors.

IV. THE PROPOSED ALGORITHM

The proposed HGAAP algorithm is a hybrid genetic scheduling algorithm that combines heuristic algorithm and genetic algorithm. It uses the schedule solution generated by HEFT as an individual of the initial population. Because the solution generated by HEFT is located at an approximate area around the optimal solution, the objective of HGAAP is to improve the performance by searching around the approximate area. The operation of the HGAAP algorithm is described in Fig.2.

A. Schedule Encoding and Chromosome Decoding

In HGAAP, a two-dimensional array is employed to represent chromosomes. The number of rows indicates the number of processors and the number of columns indicates the number of tasks. Each row is a substring that represents a processor in the HCS. If a task is assigned to a processor, the

task will be put into the substring that represents the processor. To make sure that any chromosome can be decoded into a valid schedule solution, the many-to-one method is used in the decoding. It means that different chromosomes can be decoded into the same schedule solution and it can also improve the genetic diversity. For example,

Algorithm: HGAAP
Input: DAGs and related information of applications
Output: the fittest schedule solution

1. Generate the initial population using randomly generated and HEFT generated schedule solutions;
2. Initialize the crossover rate p_c and the mutation rate p_m;
3. while (termination criteria is not met)
   {
4. Evaluate the fitness of the chromosomes in the population;
5. Sort chromosomes in the population in descent order of fitness;
6. if (the fittest individual cannot evolve in the last n iterations)
   {
7. p_m is increased by 10%;
8. if (p_m is equal to or larger than 100%)
   {
9. p_m is set to 10% and p_c is increased by 5%;
   }
   }
10. Copy the best 10% of chromosomes to the elitism set;
11. Select 90% chromosomes to the mating set randomly;
12. Apply the swap crossover operator on the chromosomes in the mating set;
13. Apply the swap mutation operator on the chromosomes in the mating set;
14. Remove redundant chromosomes from the mating set;
15. Sort chromosomes in the mating set in descent order of fitness;
16. Combine the elitism set and the fittest chromosomes in the mating set to generate the new population that has the same size with the initial population;
   }
17. Output the fittest schedule solution from the last generation;

Fig. 2. The HGAAP algorithm

for the application shown in Fig.1, one of the schedule solutions is shown in Fig.3a. The same scheduling result can be encoded into different chromosomes as shown in Fig.3b and Fig.3c.

To evaluate the fitness of chromosomes, we need to decode the chromosome into schedule solutions. For one chromosome, decoding starts from the first task of the first row and ends up with the last task of the last row. During the decoding, tasks are checked sequentially. If a task of a substring is an unscheduled and ready task, then the task is assigned to the processor the substring represents using the insertion-based scheduling policy. If the task does not satisfy the rules, check the next task. If all the tasks in the chromosome are checked, but there are still tasks left, check the first task of the first row to the last task of the last row again. Until all the tasks are assigned to processors, the decoding is over.

B. Initialization of the Population

The initial population of HGAAP is composed of two parts: random-generated schedules and the HEFT-generated schedule. Random-generated schedules can keep the diversity of population and the HEFT-generated schedule can reduce the evolution time. The initial population is created by encoding both kinds of schedules into chromosomes.

The random-generated schedules are generated as follows. Tasks are sorted first, and then are randomly permutated. Finally, tasks are randomly allocated to one of the substring of a chromosome. To maintain the diversity, substrings are selected randomly, and the number of chromosomes or the population size should be equal to or bigger than the number of processors plus one.

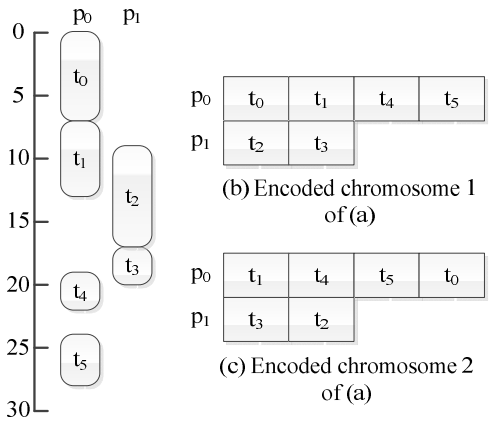


Fig. 3. Examples of task scheduling and chromosome encoding

C. Fitness Evaluation

The fitness evaluation of chromosomes is straightforward. For each chromosome, decode it into a schedule solution first and then execute it. The fitness of a chromosome can be calculated by $1/l$ where l is the makespan of the schedule. As can be seen, the shorter the makespan, the better the chromosome's fitness is.

D. Selection

After evaluating and sorting the fitness of chromosomes, top 10% chromosomes with the highest fitness values are selected as the elitism set. The elitism set guarantees that the best chromosomes are never destroyed. The remaining 90% chromosomes are selected as the mating set by using a linear rank-based selection mechanism [9], see Eq.(1). $P_{linear\_rank}(i)$ is the possibility of the i chromosome selected to the mating set; μ is the size of the population; $rank(i)$ is the calculated ranking of the i chromosome and its value is $\mu - i$; α_{rank} and β_{rank} are the parameters that follows $\alpha_{rank} = 2 - \beta_{rank}$ and $1 \leqslant \beta_{rank} \leqslant 2$. In this paper, β_{rank} is set to 1.7. By using this mechanism, all the chromosomes have the probability to be selected to the mating set. It not only maintains the diversity of the population but also reduces the premature convergence of the population.

$$P_{linear\_rank}(i) = \frac{\alpha_{rank} + \left[\frac{rank(i)}{\mu - 1}\right](\beta_{rank} - \alpha_{rank})}{\mu} \qquad (1)$$

After the selection, the elitism set is reserved without any operation while the mating set is conducted with crossover, mutation, and redundancy deletion operations, which are described in the following sections, respectively.

E. Swap Crossover

Before the crossover operation, two chromosomes are selected from the mating set. For example, if the selected probability p_c is 0.5, then the selected probability of each chromosome is 50%. After the swap crossover operation, they produce two offspring. The chromosomes chosen to produce offspring are parent chromosomes and the offspring are child chromosomes.

In a swap crossover operation, one substring is chosen randomly for each parent chromosome, and two crossover points with the same length are selected randomly for each substring. Then they exchange the selected parts between the two crossover points with each other. After exchange, two mask chromosomes can be obtained. For a mask chromosome, if a task in the other part is duplicated with a task in the exchanged part, the reduplicate task is changed to a mark ***x***. Consider the parent chromosomes shown in Fig.4a and Fig.4b. Cp11, cp12, cp21 and cp22 are the crossover points. Tasks between cp11 and cp12, and tasks between cp21 and cp22 are tasks for exchanging. The mask chromosomes are shown in Fig.4c and Fig.4d. Then child chromosomes can be produced by comparing each substring of the parent chromosome with the same substring of the mask chromosome. If the substring of parent chromosome is not the exchange substring and there is no mark ***x*** in the same substring of the mask chromosome, copy the substring from the parent chromosome to the same substring of the child chromosome. Otherwise, if the substring of the parent chromosome is not the exchange substring and there is mark ***x*** in the substring of the mask chromosome, delete the mark ***x*** and then copy the substring from the mask chromosome to the same substring of the child chromosome.

In case that the substring of the parent chromosome is the exchange substring, tasks in the substring of both mask chromosome and parent chromosome should be traversed one after another. In every traversing step, the rules are as follows. Firstly, if the current task of the mask chromosome is mark ***x***, check the next task for both mask chromosome and parent chromosome. Secondly, if the current task of the mask chromosome is identical to the task in the same location of parent chromosome, delete the task from the mask chromosome and copy the task from the parent chromosome to the same substring of the child chromosome. Thirdly, if the current task of the mask chromosome is different with the task in the same location of parent chromosome, copy the task from the parent chromosome to the same substring of the child chromosome. Finally, copy the left tasks in the substring of mask chromosome to the end of the same substring of the child chromosome with the same order as the left tasks in the substring of mask chromosome.

The child chromosome produced by the parent chromosome in Fig.4a and its mask chromosome in Fig.4c is shown in Fig.4e. Similarly, the child chromosome produced by the parent chromosome in Fig.4b and its mask chromosome in Fig.4d is shown in Fig. 4f.

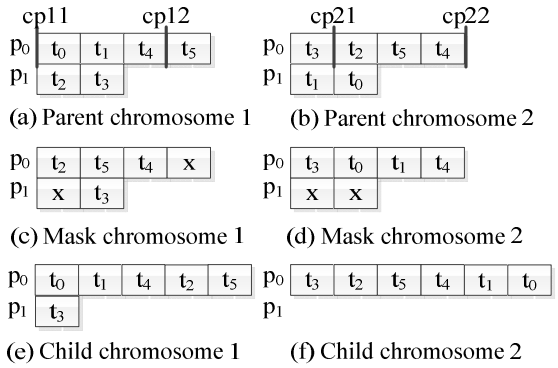


Fig. 4. The swap crossover operator

F. Swap Mutation

After the swap crossover operation, the swap mutation is applied to enhance the diversity of the population further. Two tasks in the chromosome are randomly chosen and swapped. The swap mutation operation is executed to the chromosomes in a probability of p_m. If p_m is 0.5, then 50% of the chromosomes are chosen to execute the swap mutation operation on average.

G. Deleting the Redundant Chromosomes

After the swap crossover and swap mutation operations are finished, a redundancy checking is applied to the mating set where all the schedules decoded by chromosomes are checked. If there are ***n*** decoded schedules having the same scheduling results, only one of them is preserved and the other are removed from the mating set to enhance the diversity of the population. In some cases, removing redundant chromosomes may lead to the number of the mating set is smaller than 90% of the population. To solve this problem, we insert some new chromosomes into the mating set to keep the population size unchanged. These chromosomes are generated by implementing the swap mutation to the fittest chromosome in the mating set. Finally, the chromosomes in the mating set are combined with the chromosomes in the elitism set to create the next generation.

H. Parameters Adaptation and Termination Criterion

Because the characteristics of DAGs are various, using fixed p_m and p_c cannot always lead to the optimal solution. Hence, the p_c and p_m of HGAAP are not fixed. The algorithm starts with an initial status where p_c and p_m are set to 5% and 10%, respectively. During the iteration of the generation, if the evolution occurs, the current parameters will be used in the next ***n*** iterations. Otherwise, if the fittest individual cannot evolve in the last ***n*** iterations, p_m will be increased by 10%. In case that p_m is over 100%, it returns to 10% and p_c will be increased by 5%. The parameter adaptation works in such a way to promote the possible evolutions and to result in more efficient solutions at the cost of more computation time

When p_m and p_c are both 100% and there is no evolution occurs, the algorithm is over. The solution of the algorithm is the schedule decoded by the fittest chromosome of the population.

V. EXPERIMENTAL RESULT AND ANALYSIS

To evaluate the proposed algorithm, we compared it with existing HEFT and H2GS by using a large number of randomly generated DAGs in the experiments.

A. Performance Metrics

The Normalized Schedule Length (NSL)[9] and the speedup[3] are used to compare the performance of the algorithms. The NSL is defined as the normalized makespan to the lower bound of the makespan, as shown in Eq.(2). The $\boldsymbol{CP_{lower}}$ is the Critical Path of the DAG executed on the fastest processor $\boldsymbol{p_\alpha}$. A critical path is the path from an entry task to an exit task that has the greatest sum of computation costs of tasks and communication costs of edges. The denominator is the sum of computation costs of tasks located on the $\boldsymbol{CP_{lower}}$.

$$\mathrm{NSL} = \frac{\text{makespan}}{\sum_{t_i \in CP_{lower}} c_{i,a}} \qquad (2)$$

The speedup of a task schedule is defined as the ratio of the minimal sequential execution time to the makespan, see Eq.(3). The minimal sequential execution time is calculated by assigning all the tasks to the processor where the cumulative computation cost is minimum.

$$\mathrm{speedup} = \frac{\min_{p_j \in P}\{\sum_{t_i \in T} c_{i,j}\}}{\text{makespan}} \qquad (3)$$

B. Randomly Generated DAGs

The DAGs are generated by a random DAG generator that has a set of input parameters and can generate DAGs with various characteristics. The input parameters are described as below:

- Number of tasks, $\boldsymbol{n}$; number of processors, $\boldsymbol{p}$.
- Communication to computation ratio, ***CCR***: the ratio that the average communication cost divided by the average computation cost of all the tasks in the DAG.
- Shape of the DAG, $\boldsymbol{\alpha}$: the height and width of a DAG are randomly generated, using uniform distribution with values of $\frac{\sqrt{n}}{\alpha}$ and $\alpha \times \sqrt{n}$, respectively. If the value is not an integer, the smallest integer not less than the value is selected.
- Average computation cost, ***ACC***: the average computation cost of all the tasks in the DAG.
- Computation cost heterogeneity factor, $\boldsymbol{h}$: this value indicates the variance of the computation costs of a task in different processors. If the value is high, the variance of the computation cost of a task in different processors is high and vice versa. If the value is zero, the computation cost of a task in different processors are assumed to be the same. The average computation cost of a task $\boldsymbol{w_i}$ is randomly generated, using a uniform distribution with a mean value of ***ACC***. The computation cost of a task for each processor can be set by randomly generating from the range $[\boldsymbol{w_i} \times (1\text{-}h/2), \boldsymbol{w_i} \times (1+h/2)]$. During the generating of the computation costs, the monotonic of the computation costs in different processors must be maintained.

In our experiments, the generated 1500 DAGs consist of three different shape values: 0.5, 1.0 and 2.0. For each kind of shape, we use four kinds of HCS varying from 2 to 8 processors with an increment of 2; five kinds of task numbers varying from 20 to 100 tasks with an increment of 20; five kinds of ***CCR***: 0.1, 0.5, 1.0, 2.0 and 5.0; and five kinds of heterogeneity: 0.1, 0.2, 0.4, 0.6 and 0.8. For all the DAGs, the value of ***ACC*** is 100.

C. Tuning the HGAAP algorithm

For HGAAP, the parameters need to be set are the size of population and the evolution iteration time $\boldsymbol{n}$. 100 DAGs are randomly selected from the generated DAG set and are used to tune HGAAP and find out the best parameters. The size of population is varied from 9 to 45 with an increment of 4 and the evolution iteration time $\boldsymbol{n}$ is varied from 1 to 11 with an increment of 2. Our preliminary experiments indicated that when the size of population is 17 and the evolution iteration number $\boldsymbol{n}$ is 7, the HGAAP algorithm can achieve its best performance. Therefore, the above parameters are utilized in the following experiments for performance evaluations

D. Performance Results

The DAGs are scheduled by HEFT, H2GS and HGAAP algorithms, respectively. For all generated scheduling solutions, the NSL and speedup values are calculated and compared with respect to different number of tasks and CCR values.

All the above experiments were conducted on a windows7 desktop running on a 3.70GHz Intel Core i3 Dual-Core CPU processor with 4GB main memory. The total running time of HGAAP for scheduling 1500 DAGs is 65 hours. Thus, one DAG needs about 156s to get its scheduling solution by the HGAAP.

We divided the 1500 DAGs into 5 categories with different CCR or number of tasks, and each category has 300 DAGs, and then we calculate the average scheduling results of the 300 DAGs as follows.

The average NSL values generated by HEFT, H2GS and HGAAP with respect to CCR are shown in Fig.5a. The average NSL values of HGAAP are shorter than those of HEFT and H2GS by: (1.47%, 0.75%), (2.43%, 0.87%), (3.52%, 1.18%), (6.11%, 1.61%) and (6.67%, 1.63%), for CCR of 0.1, 0.5, 1.0, 2.0, and 5.0, respectively. The speedup values generated by HEFT, H2GS and HGAAP with respect to CCR are shown in Fig.5b. The average speedup values of HGAAP are higher than those of HEFT and H2GS by: (1.47%, 0.99%), (2.39%, 1.02%), (3.57%, 1.26%), (6.11%, 1.58%) and (6.57%, 1.41%), for CCR of 0.1, 0.5, 1.0, 2.0, and 5.0, respectively;

The average NSL values generated by HEFT, H2GS and HGAAP algorithms with respect to the number of tasks are shown in Fig.5c.The average NSL values of HGAAP algorithm are shorter than those of HEFT and H2GS by: (5.96%, 1.51%), (5.69%, 1.36%), (4.81%, 1.35%), (4.39%, 1.45%) and (4.10%, 1.43%), for the number of tasks of 20, 40, 60, 80, and 100, respectively. The speedup values

generated by HEFT, H2GS and HGAAP with respect to the number of tasks are shown in Fig.5d. The average speedup values of HGAAP are higher than those of HEFT and H2GS by: (4.42%, 0.94%), (4.22%, 1.18%), (3.52%, 1.40%), (2.82%, 1.50%) and (2.71%, 1.42%), for DAG size of 20, 40, 60, 80, and 100, respectively.

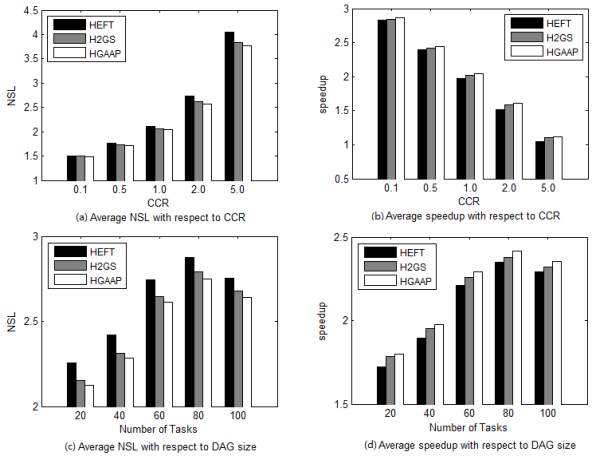


Fig. 5. Average NSL and speedup on randomly generated DAGs

In all the experiments with varied CCR and varied number of tasks, the HGAAP algorithm outperforms the HEFT and H2GS algorithms in terms of both NSL and speedup.

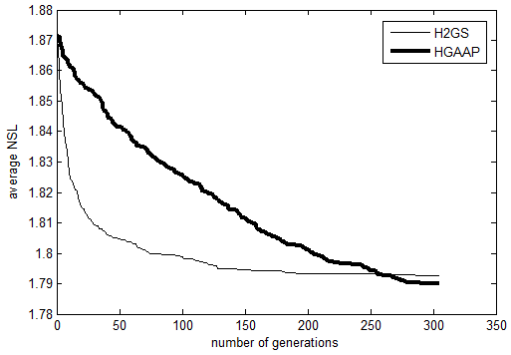


Fig. 6. The convergence trace of the average NSL

The convergence curve is usually used to compare the performance of genetic algorithms. Because characteristics of DAGs are various, we choose one kind of DAGs for comparing the convergence rate of H2GS and HGAAP. In the experiment, the inputs of the DAGs are as follows: the number of tasks is 50; the number of processors is 4; the CCR is 1.0; the shape value α is 1; the ACC is 100, and the heterogeneity is 0.5. 50 randomly generated DAGs are used to compare the convergence performance of the two algorithms. The average result of 50 DAGs is shown in Fig.6. The HGAAP algorithm searches all the possible p_c and p_m and if the current p_c and p_m can lead to the evolution, the HGAAP algorithm uses them to generate the next generation. Because the H2GS algorithm uses the fixed p_c and p_m that are calculated before the iteration, it has a faster convergence rate. However, the final result obtained by the HGAAP algorithm is better than that of the H2GS algorithm.

VI. Conclusion

In this paper, we proposed a hybrid heuristic genetic algorithm HGAAP for static task scheduling on HCS to further improve the performance of existing algorithms. The crossover probability and mutation probability of the HGAAP are adaptive according to the current evolution status, and redundant individuals are removed during the iteration of generations. Experimental results on a large number of randomly generated DAGs validated that the proposed HGAAP algorithm can achieve better scheduling results than existing HEFT and H2GS algorithms.

In future work, we are planning to extend the HGAAP algorithm to partially-connected networks of HCS. This will make the proposed algorithm be suitable for more cases.

VII. References

[1] Sih, Gilbert C., and Edward A. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." *IEEE transactions on Parallel and Distributed systems* 4.2 (1993): 175-187.

[2] Zomaya, Albert Y., Chris Ward, and Ben Macey. "Genetic scheduling for parallel processor systems: comparative studies and performance issues."*IEEE Transactions on Parallel and Distributed systems* 10.8 (1999): 795-812.

[3] Topcuoglu, Haluk, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." *IEEE transactions on parallel and distributed systems* 13.3 (2002): 260-274.

[4] Daoud, Mohammad I., and Nawwaf Kharma. "A hybrid heuristic–genetic algorithm for task scheduling in heterogeneous processor networks." *Journal of Parallel and Distributed Computing* 71.11 (2011): 1518-1531.

[5] Daoud, Mohammad I., and Nawwaf Kharma. "An efficient genetic algorithm for task scheduling in heterogeneous distributed computing systems."*Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*. IEEE, 2006.

[6] Daoud, Mohammad I., and Nawwaf Kharma. "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems."*Journal of Parallel and distributed computing* 68.4 (2008): 399-409.

[7] Cheng, Hui. "A high efficient task scheduling algorithm based on heterogeneous multi-core processor." *Database Technology and Applications (DBTA), 2010 2nd International Workshop on*. IEEE, 2010.

[8] Bajaj, Rashmi, and Dharma P. Agrawal. "Improving scheduling of tasks in a heterogeneous environment." *IEEE Transactions on Parallel and Distributed Systems* 15.2 (2004): 107-118.

[9] Bansal, Savina, Padam Kumar, and Kuldip Singh. "An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems." *IEEE Transactions on Parallel and Distributed Systems* 14.6 (2003): 533-544.