Reliability-Oriented Genetic Algorithm for Workflow Applications Using Max-Min Strategy

Xiaofeng Wang[#1], Rajkumar Buyya*, Jinshu Su[#2]

[#]*College of Computer, National University of Defense Technology*
Changsha, 410073, Hunan, China
{xf_wang[1], sjs[2]}@nudt.edu.cn

**GRIDS Laboratory, Department of Computer Science and Software Engineering*
The University of Melbourne, VIC 3010, Australia
raj@csse.unimelb.edu.au

***Abstract*—** To optimize makespan and reliability for workflow applications, most existing works use list heuristics rather than genetic algorithms (GAs) which can usually give better solutions. In addition, most existing GAs evolve a scheduling solution randomly, which may give invalid solutions or lead to slow convergence of the algorithm. In this paper, we define three heuristics for GAs to decide the priorities for a resource and a task dynamically. We propose Look-Ahead Genetic Algorithm (LAGA) to optimize both makespan and reliability for workflow applications. It uses a novel evolution and evaluation mechanism: the genetic operators evolve the task-resource mapping for a scheduling solution, while the solution's task order is determined in the evaluation step using our new max-min strategy, which is specifically proposed for GAs. Our experiments show that LAGA can provide better solutions than existing list heuristics and evolve to better solutions more quickly than a traditional genetic algorithm.

1. Introduction

Distributed computing systems, such as peer-to-peer systems and Grids, have been widely deployed for executing computationally intensive applications. These systems usually comprise large number of geographically distributed resources which are more susceptible to unreliability. It seems likely that as distributed systems become larger and more widely dispersed, the reliability of an application running on these systems decreases due to the system's inherent unreliability. Hence, the scheduling of an application in such environments must take into account the reliability of the application besides the execution time (makespan) which is usually considered.

For a workflow application, which can be modelled by a Directed Acyclic Graph (DAG), optimizing makespan and reliability simultaneously is known to be a NP-hard problem. Many list heuristics have been proposed for DAG modelled applications. Most of them tried to give makespan [9,10,14] or reliability [1,16,18] suboptimal solutions, whose optimality cannot be guaranteed [4]. However, Genetic Algorithms (GAs) can usually provide better quality solutions than list heuristics [7,12]. Although GA is more time consuming than list heuristics, it is acceptable for applications with long runtime. In addition, the speed of GA can be accelerated by using parallel genetic algorithm technology [2].

Currently, Bi-objective Genetic Algorithm (BGA) [17] is the only GA we know that can give both makespan and reliability optimized scheduling solutions for workflow applications. However, BGA can give invalid solutions which violate the dependency between tasks. To address this problem, the scheduling of a workflow application can be divided into two components: task-resource mapping and task execution order [4]. Most existing GAs evolve these two components randomly [4,8,17], which may lead to slow convergence of the algorithm. In fact, the GAs can be improved by using heuristics to evolve solutions more intelligently. However, very few heuristics were specifically proposed for GAs. Although some two phase heuristics have been proposed, which are reported to be more efficient than other heuristics [7], they cannot work with GAs because of the evolution mechanism.

In this paper, we propose Look-Ahead Genetic Algorithm (LAGA) to intelligently optimize both makespan and reliability for a workflow application. We define three new heuristics for LAGA to decide the priorities for a resource and a task. LAGA has two characteristics: (i) it optimizes the typical GA by a new mutation operator according to our resource priority heuristic. (ii) it uses a novel evolution and evaluation mechanism: the genetic operators (crossover and mutation) evolve the task-resource mapping for a solution, while the solution's task execution order is determined in the evaluation step using our proposed max-min strategy, which is the first two phase strategy that can work with GAs. LAGA can avoid the invalid solution problem in BGA [17] by using the max-min strategy to evolve the task execution order. More importantly, LAGA can accelerate the evolution of solutions more intelligently by using our evolution and evaluation mechanism.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 defines the scheduling problem. Section 4 proposes the priority heuristics for resources and tasks. Our novel LAGA is

978-0-7695-3622-4/09 $25.00 © 2009 IEEE
DOI 10.1109/CCGRID.2009.14

presented in Section 5, while experimental results are presented in Section 6, followed by the conclusion and future work in Section 7.

2. Related Work

For a workflow application in unreliable distributed systems, makespan and reliability should be optimized simultaneously. This problem is known to be NP-hard [19], hence many list heuristics [3,10,11,13] have been proposed. To optimize the makespan, Dynamic Level Scheduling (DLS) [10] assigns the task with a higher static level and an earlier start time to the fastest resource. To optimize the reliability, Dongarra et al. [1] proved that the tasks should be assigned to the node with the minimum multiplication value of instruction execution time γ and reliability λ. Dogan et al. [18] proposed a bi-criteria heuristic rule called RDLS based on DLS. RDLS evaluated the priority of a task-resource assignment according to the task's size, its starting time, the resource's computing power, and the reliability cost. A two phase min-min heuristic was reported to be the best tested list heuristics [17]. It works as follows: i) for each task, select its assumed resource which can start the task earliest. ii) from all the tasks with the assumed resource, it selects the task with the minimum ending time to be scheduled. This heuristic cannot be used by GAs, because the task-resource mapping is fixed at the evolution time. However, our max-min strategy is specifically proposed for GAs based on our new task priority heuristics.

Usually, a genetic algorithm (GA) can give better scheduling solutions than a list heuristic [7]. Currently, BGA [17] is the only GA that we know can optimize both makespan and reliability for a workflow application. But it evolved the scheduling solutions randomly, which may give invalid solutions violating the dependence between tasks. To keep the task dependence in the evolution, two methods have been proposed. CorreA et al. [14] define a partition V_1, V_2 of the tasks such that there is no dependency from a task in V_1 to a task in V_2, and their crossover operation only exchanges the order of the tasks in set V_2. Wang et al. [4] represented a scheduling solution as two strings: the task-resource mapping string and the task execution order string, and they evolved the two strings separately. Although these two methods can solve the invalid solution problem, they did not take into account the reliability of an application. In addition, most existing GAs [4,8,17] evolve the task-resource mapping and the task execution order randomly, which may lead to slow convergence of the algorithm. In our look-ahead genetic algorithm, a solution's task execution order is determined by our max-min strategy, so that the algorithm will not give invalid solutions. Moreover, our new evolution and evaluation mechanism can accelerate the evolution of solutions by applying our resource and task priority heuristics.

3. Scheduling Problem Model

A scheduling system model consists of an application, a specific computing environment and the scheduling criteria. We model a workflow application as a DAG: $App = (V, E)$. V is the set of task nodes $v_i (1 \le i \le n)$, which denote the tasks of an application. E is the set of edges $e(i, j)(1 \le i < j \le n)$ which represents the dependence constraint between tasks v_i and v_j, v_i is the parent task and v_j is the child task. A task with no parents is called an entry task, and a task with no children is called an exit task. For each task node v_i, its weight $|v_i|$ is the number of instructions required to be executed for this task, which is assumed to be known using compiling technology [1]. Like in some other works [1,8,13], we focus on computationally intensive applications, which means the communication time between tasks is not modeled. Extending our model to include the communication time will be our future work.

The computing environment is represented by a set of m resources $R = \{r_1, r_2 \cdots r_m\}$. Each resource r_i is associated with two values: λ_i, the resource's failure rate and γ_i, the resource's computing power illustrated by unitary instruction execution time (i.e. the time to execute one instruction).

In a workflow application, each task could be executed only after all its parent tasks have been completed. Thus the available starting time for a task v_i is:

$$t_i^{avail} = \max_{e(j,i) \in E} t_j^e, \quad (1)$$

where t_j^e is the ending time for task v_j. The available starting time for all entry tasks is 0. Let $M : V \to R$ denotes the task-resource mapping function, and then $M(i) = r_j$ means that task v_i is assigned to resource r_j. The beginning and ending times of task v_i can be defined as:

$$\begin{aligned} t_i^b &= \max\{t_i^{avail}, idle(M(i))\} \\ t_i^e &= t_i^b + |v_i| \cdot \gamma_j \quad where\ M(i) = r_j \end{aligned}, \quad (2)$$

where $idle(M(i))$ is the time when resource $M(i)$ becomes idle. Let t_S^j be the time when resource r_j finishes all the tasks assigned to it in scheduling S, it can be defined to be:

$$t_S^j = \max_{i|M(i)=r_j} \{t_i^e\}. \quad (3)$$

The reliability of an application can be given by the probability that all the resources remain functional until the tasks assigned to them finish. The probability that resource r_i can successfully complete all its tasks in scheduling S is $R_s^i = e^{-t_s^i \cdot \lambda_i}$ [1]. Thus the success probability R_s for an application in scheduling S can be computed as the product of all R_s^i, which is illustrated in Equation 4. We can see that to maximize the reliability, we need to minimize the failure factor $fal(S) = \sum_{i=1}^{m} t_s^i . \lambda_i$.

$$R_s = \prod_{i=1}^{m} R_s^i = e^{-\sum_{i=1}^{m} t_s^i . \lambda_i} . \quad (4)$$

Our scheduling algorithm tries to maximize the reliability and minimize the makespan for an application under the time constraint D. Therefore the scheduling problem can be formalized as following:

$$\begin{aligned} &\text{Minimize} \quad fal(S) = (\sum_{i=1}^{m} t_S^i . \lambda_i) \\ &\text{Minimize} \quad time(S) = \max_{r_i \in R}(t_S^i) \\ &\text{Subject to} \quad time(S) < D \end{aligned} . \quad (5)$$

4. Proposed Heuristics for GA

In this section, we define one resource priority heuristic and two task priority heuristics for our proposed GA. To optimize the reliability for an application, it has been proven that the resource r_i which has the minimal multiplication value of instruction speed (unitary instruction execution time) γ_i and failure rate λ_i should have a higher priority to be chosen in the scheduling [1]. So we can define our resource priority heuristic as:

Resource Priority Heuristic (ResPH)

Let $1/\gamma_i \lambda_i$ be the priority of a resource r_i, and S be a schedule where all the tasks are assigned to a resource with the highest priority. Then any other schedule $S' \neq S$ with reliability of $R_{S'}$ is such that $R_{S'} < R_S$.

To minimize the makespan for an application, we should give higher priority to tasks which can start earlier and tasks which have a bigger influence on the makespan of the application. Thus we can define our fist task priority heuristic as:

Task Priority Heuristic 1 (TaskPH1)

Let the importance of a task v_i be the length of the longest path beginning from the task in the DAG graph, which can be denoted as:

$$impt(i) = \begin{cases} |v_i| & \text{if } v_i \text{ is an exit task} \\ |v_i| + \max\limits_{e(i,j) \in E} impt(j) & otherwise \end{cases} . \quad (6)$$

And the task v_i's priority $p(i)$ is:

$$p(i) = E(\gamma) \cdot impt(i) - \max(t_i^{avail}, idle(M(i))) , \quad (7)$$

where $E(\gamma)$ is the mean instruction speed of all resources. Then, if there are two tasks scheduled to the same resource, the one with the higher priority should be scheduled first.

TaskPH1 uses the mean resource instruction speed to estimate the completion time of the longest path beginning from a task. It is easy and simple to be implemented. Assuming that all the tasks of an application have been assigned to some specific resource, we can have a more precise estimation of the completion time for a path, and thus define the second task priority heuristic as:

Task Priority Heuristic 2 (TaskPH2)

Let the estimated completion time for the longest path beginning from task v_i be:

$$comp(i) = \begin{cases} |v_i| \cdot \gamma_j & \text{if } v_i \text{ is an exit task} \\ |v_i| \cdot \gamma_j + \max\limits_{e(i,k) \in E} comp(k) & otherwise \end{cases} . \quad (8)$$

where $M(i) = r_j$

And task v_i's priority $p(i)$ is:

$$p(i) = comp(i) - \max(t_i^{avail}, idle(M(i))) , \quad (9)$$

Then, if there are two tasks scheduled to the same resource, the one with the higher priority should be scheduled first.

5. Look-Ahead Genetic Algorithm

A typical GA consists of the following steps: (1) create an initial population consisting of randomly generated chromosomes (solutions); (2) evaluate the fitness of each solution and select the solutions for the next population; (3) generate a new generation of solutions by applying two genetic operators namely crossover and mutation; and (4) repeat step 2 and 3 until the population converges. To evolve the solutions intelligently without giving invalid solutions, we design the Look-Ahead Genetic Algorithm (LAGA). Its genetic operators evolve the task-resource mapping for a solution, while the task execution order is determined in the evaluation step using our new max-min strategy based on *TaskPH1* or *TaskPH2*. The details of LAGA are presented in the following subsections.

5.1 Chromosome Encoding

As illustrated in Fig. 1b, we use a two-dimensional string to represent a scheduling solution. One dimension of the string represents the index of resources, which depicts the task- resource mapping; the other dimension denotes the order between tasks. The two-dimensional string can be converted into the task-resource mapping

string M (Fig. 1c) directly, which is a vector of length $|V|$. The task-resource mapping string has the same symbol M with the mapping function, since they have the same meaning. Hence, $M(i) = r_j$ denotes task v_i is scheduled to resource r_j.

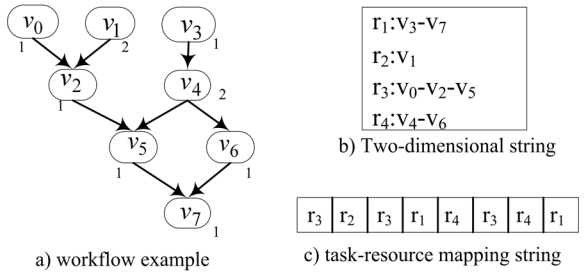


Fig. 1 Encoding Example

5.2 Evolution

GAs use crossover and mutation operations to evolve the solutions for an application. A crossover operation tries to create a better chromosome by exchanging two fittest chromosomes, and a mutation operation usually changes some of the genes in a chromosome randomly. To keep the dependence between tasks, the two operations usually evolve the task-resource mapping and the task execution order of a solution separately and randomly [4,8]. This may result in difficulty for the GA to find a better solution, since a good task execution order for one task-resource mapping does not mean it is also good for another task-resource mapping.

Here, our crossover and mutation operations only evolve the task-resource mapping for the new chromosomes. The task order of the new offspring is to be determined later in the evaluation step. Our crossover operation first randomly chooses some pairs of chromosomes with a probability p_c. For each pair, it randomly generates a cut-off point for the task-resource mapping string M, which divides the strings of the pair into top and bottom parts. Then, the task-resource assignments in each bottom part are exchanged. And two new task-resource mapping offspring is generated.

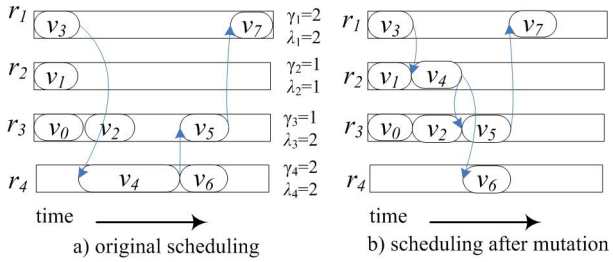


Fig. 2 Mutation Operation

Our mutation operation mutates a solution intelligently based on *ResPH*. It selects a solution with a probability p_m. Then, it randomly chooses one task in the solution and reassigns it to any resource which has lower $\gamma_i \lambda_i$. As shown in Fig. 2a, task v_4 is originally scheduled to resource r_4 whose $\gamma_i \lambda_i$ is 4, so the mutation reassigns it to resource r_2 with a lower $\gamma_i \lambda_i$ of 1. Fig. 2b shows the new scheduling, in which both the makespan and the reliability of the application have been improved.

Algorithm 1 The Evaluation Algorithm

```
1  input: task-resource mapping string M
2  output: {t_S^i, que_i} for each resource r_i
3  for each entry task v_j
4      add v_j to the task ready queue que_ready_M(j)
5      t_j^avail ← 0
6  end for

7  repeat
8      min_end ← ∞              //the minimum ending time
9      task_sel ← null          //the task selected
10     for each resource r_i
           //max-min phase 1
11         find the task v_j with the maximum priority value
           from que_ready_i
           //max-min phase 2
12         compute the ending time t_j^e for v_j using equation 2
13         if t_j^e < min_end
14             min_end ← t_j^e      //the minimum ending time
15             task_sel ← j         //record the selected task
16         end if
17     end for

18     res_sel ← M(task_sel)
19     remove v_task_sel from que_ready_res_sel
20     add v_task_sel to que_res_sel
21     t_S^res_sel = idle(res_sel) = t_task_sel^e
22     for each child task v_i of task v_task_sel
23         update t_i^avail using equation 1
24         if v_i is ready to run, add it to que_ready_M(i)
25     end for
26 until every que_ready_i is empty
```

5.3 Evaluation

In evaluation step, most GAs only evaluate the quality of a solution, they do not improve the scheduling for a solution. In our evaluation operation, LAGA schedules the task execution order for a new solution first. Then it

calculates the estimated ending time t_i^e for each task v_i, so that we can evaluate the makespan and the failure factor of the new scheduling *S* using equation 5.

To give an optimized task execution order for a task-resource mapping string, we use a new two phase max-min strategy as shown in *Algorithm 1*, which is specifically proposed for GAs. For each resource, the algorithm first selects its next to be scheduled task which has the maximum priority based on *TaskPH1* or *TaskPH2*. Then, from all the next to be scheduled tasks of the resources, it selects the task with the minimum ending time to be scheduled. Given a task-resource mapping string *M* (the mapping function), all the tasks assigned to resource r_i are put into que_i in their scheduling order, and the algorithm outputs the estimated completion time t_S^i for each resource r_i in the new scheduling *S*. que_ready_i is the queue containing the unscheduled tasks which are ready to run on resource r_i.

The algorithm works as follows: (i) add each entry task v_j to the task ready queue of its assigned resource $M(j)$, and set the task's available starting time to 0 (line 3~6); (ii) select the task with the maximum priority for each resource (line 11); (iii) Among all the selected tasks, the task v_{task_sel} with the minimum ending time is selected to be scheduled (line 12~16); (iv) schedule the selected task v_{task_sel} (line 18~20), update the task completion time and the idle time for resource $M(task\_sel)$ (line 21); (v) update the state for all the child tasks of the scheduled task (line 22~25); (vi) repeat step ii-v until all the tasks have been scheduled.

Theorem 1. The time complexity of the evaluation algorithm is $O(n\log n + nm + d)$, where *m* is the number of resources, *n* is the number of nodes (tasks) in a DAG and *d* is the number of directed edges (dependence constraints).

Proof. The time complexity of initializing the task ready queue is O(*n*) (line 3~6). An entire iteration (line 7~26) schedules one task at a time. So it will run *n* times. To effectively sort and select a task for each resource (line 11), it takes time $O(\log n)$. The time complexity of computing the task ending time and select the task with the minimum ending time is O(*m*) (line 12~16). The time complexity of line 18~21 is O(1). So the time complexity of repeating line 8~21 is $O(n(\log n + m + 1))$. To update the available time for the child tasks (line 22~25), it consumes time O(*d*). Thus, the whole time complexity for the evaluation algorithm is $O(n + n(\log n + m + 1) + d) = O(n\log n + nm + d)$.

5.4 Selection

In GA, the fitness function is used to measure and select the solutions. As our goal is to optimize the makespan and reliability for an application under the time constraint, the sum of weighted global ratios (SWGR) model [17] can be used to compute the fitness. So the fitness value of a scheduling S can be defined as:

$$f(S) = \omega_1 \cdot \frac{fal(S)-\text{minFal}}{\text{maxFal}-\text{minFal}} + \omega_2 \cdot \frac{time(S)-\text{minTime}}{\text{maxTime}-\text{minTime}} + penalty(S) \quad \text{where } \omega_1 + \omega_2 = 1 \qquad (10)$$

$$penalty(S) = \begin{cases} 0 & if\ time(S)<D \\ 1 & if\ time(S)>D \end{cases}$$

Here maxFal and minFal are the maximum and minimum failure factors for the solutions in the current generation respectively, while maxTime and minTime are the maximum and minimum makespan respectively. The first two elements of $f(S)$ encourage the algorithm to choose the solutions with minimum failure factor and minimum makespan. The third element $penalty(S)$ is to handle the time constraint. If the makespan of a scheduling exceeds the deadline *D*, the function gives a penalty to its fitness value. To select the solutions for the next generation, the chromosomes are first ordered in the descending order of their fitness value $f(S)$. Then the algorithm uses the commonly used roulette wheel selection scheme [4] to choose solutions for the next generation. The details of this scheme can be found in [4] and thus will not be repeated here.

6. Experiment and Evaluation

Like many previous works [4,5,15,17], we use a random DAG graph generator to simulate the application as three parameters: the number of tasks, the mean outdegree of a task node and the mean task size. In our simulation, the number of tasks in a workflow application is chosen between 40 and 200. The mean outdegree for a task node is set to be 2. The task's size is chosen uniformly between 1×10^4 Million instructions (MI) and 15×10^6 MI. For the computing environment, we also simulate it as three parameters: the number of resources, the resource's mean speed and the resource mean failure rate. There are 40 resources, their speeds are uniformly distributed in $[5\times10^{-4}, 10^{-3}]$ milliseconds per instruction and their failure rates are assumed to be uniformly distributed from $10^{-3}/h$ to $10^{-4}/h$ [17].

For the other parameters in the system, the fitness evaluation weight ω_1 and ω_2 are set to be 0.5, so the algorithm assigns the same priority to both reliability and makespan. The probability p_c for crossover operation is 0.5, and p_m for mutation operation is 0.25. The population size of LAGA is 20. For each kind of workflow application with the same parameters, we

create 5 instances so that they can have a wide representation. In addition, for each workflow application, we run the genetic algorithms 3 times to get their average results.

6.1 LAGA compared with list heuristics

DLS [10] and RDLS [18] are two of the best existing list heuristics to optimize makespan or reliability for workflow applications [6,16]. To compare our LAGA with these two list heuristics, we run DLS and RDLS 100 times respectively to get the average result. The number of tasks varies from 40 to 200. Fig. 3 shows that LAGA can provide the best solutions for both makespan and reliability. In particular, LAGA achieves a considerably larger improvement ratio (of about 15%) for makespan and reliability when the number of tasks is small (40 tasks), as compared to when the number of tasks is large (200 tasks). This is because when there are fewer tasks, there will be more idle resources for LAGA to choose for each task. Hence, LAGA is able to examine each of them to find the most befitting resource. But list heuristics only examine one resource according to the heuristic value, which may not be the best one.

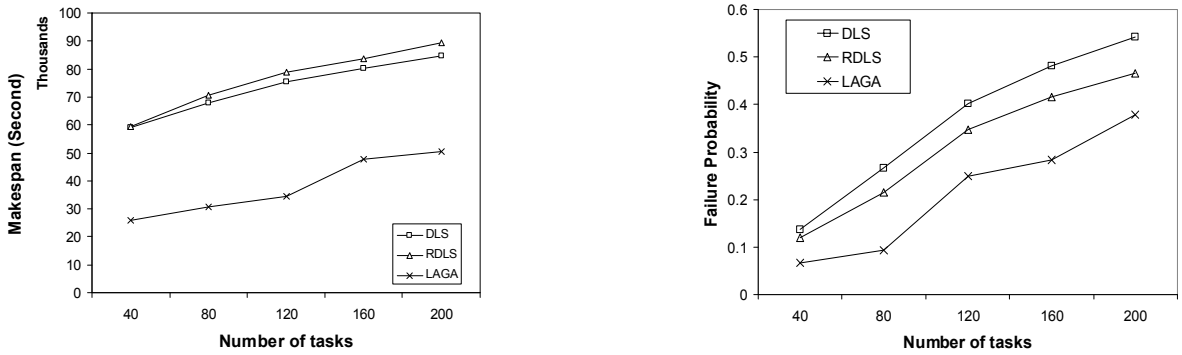


Fig. 3 Makespan and failure probability of a scheduling solution given by DLS, RDLS and LAGA.

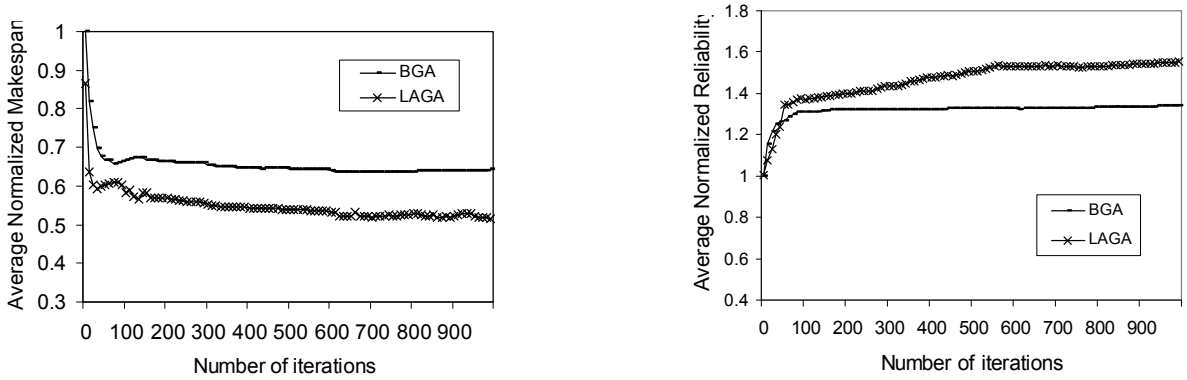
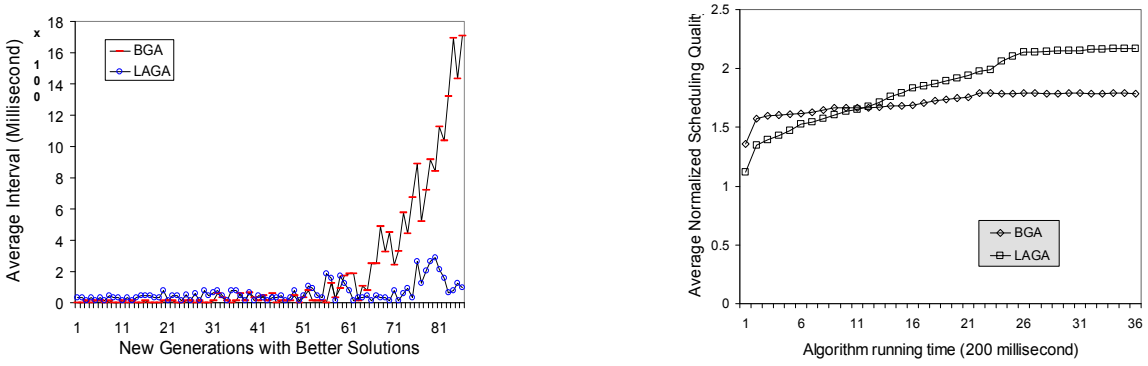


Fig. 4 Average Normalized makespan and reliability of a scheduling in terms of iterations.



a. Time needed for a new generation

b. Scheduling quality improvement over time

Fig. 5 Performance evaluation of BGA and LAGA in terms of time.

6.2 LAGA compared with another GA

We compare our LAGA with BGA [17] which evolves a solution randomly. Their performances are compared in terms of iteration and time. For the comparison in terms of iteration, we compute the average normalized makespan and the normalized reliability of the solutions, which are the mean makespan and reliability of the current generation normalized by the mean makespan and failure factor of the initial generation respectively. The application has 200 tasks, while the number of iterations of the two GAs are 1000. Fig. 4 shows LAGA improves the makespan and reliability for an application more quickly than BGA. And within the same iteration, LAGA can always give better quality scheduling solutions than BGA.

To compare the performance in terms of time, we do two experiments. In the first experiment, the average time needed for a new generation (a generation with better quality solutions) is calculated. Fig. 5a shows BGA needs less time (16 millisecond less on average) than LAGA for a new generation at the start of the evolution (generations 1~31), but needs much more time for a new generation at the end (generations 62~). Since BGA randomly searches for better solutions, it is easy to find a better solution at the start of the evolution, but gradually becomes harder to find a better solution at the end. On the other hand, LAGA needs to run the evaluation algorithm which is more time-consuming and thus evolves more slowly at the start of the evolution. But, our priority heuristics are able to ensure that LAGA still finds better solutions using the similar amount of time at the end of the evolution.

In the second experiment, the average scheduling quality of the two GAs over the algorithm running time is compared. We sample the normalized scheduling quality of the two algorithms after every 200 milliseconds. The normalized scheduling quality of a GA is the sum of the normalized reliability and the inverse of the normalized makespan. In Fig. 5b, at the start of the evolution, BGA improves the quality of the solutions more quickly than LAGA. Then it becomes very difficult and slower for BGA to improve the quality, while LAGA outperforms BGA to supply better quality solutions at the end of the evolution. Hence, Fig. 5b proves our analysis of the previous experiment (Fig. 5a) is correct.

6.3 Efficiency of Priority Heuristics

a) Efficiency of *ResPH*

To evaluate the efficiency of *ResPH*, we compare LAGA using only *ResPH* with BGA. The tested application has 200 tasks. Fig. 6 shows the average normalized scheduling quality (the same definition as above) of the two GAs over the number of iterations. It can be seen that *ResPH* gives better scheduling quality at the start of the evolution. But at the end, *ResPH* is no longer able to further improve the scheduling quality, thus resulting in LAGA achieving the same scheduling quality as BGA. This shows that after a period of evolution, it will be difficult for LAGA to improve the quality of a solution by just assigning a task to a faster or a more reliable resource.

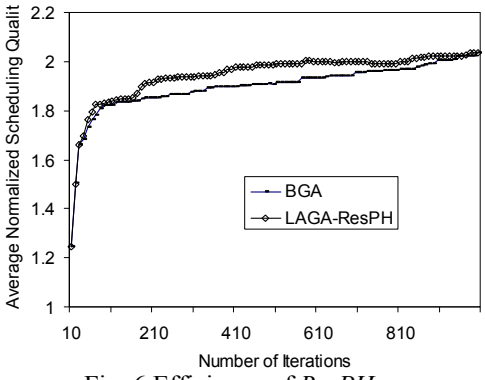


Fig. 6 Efficiency of *ResPH*.

b) Efficiency of *TaskPH1* and *TaskPH2*

We compare the efficiency of our task priority heuristics (*TaskPH1* and *TaskPH2*) on LAGA. For the experiments, the number of tasks in the application varies from 40 to 200. Fig. 7 shows the average makespan and reliability of the solutions given by two types of LAGAs using *TaskPH1* or *TaskPH2*, normalized by the makespan and reliability of the solutions given by BGA respectively. We can observe that both *TaskPH1* and *TaskPH2* enable LAGA to have a lower makespan (normalized makespan < 1) and a higher reliability (normalized reliability > 1) than BGA. But, *TaskPH2* achieves a significantly lower makespan and higher reliability than *TaskPH1* when the workflow application is of medium size (120 tasks). Otherwise, *TaskPH1* and *TaskPH2* achieve similar performance for the case when workflow application of small or large size. This is because when the number of tasks is small, the GA can find a good solution even without heuristics. When the number of tasks is large, every resource will be assigned many tasks, which makes it very difficult to estimate the completion time for a long task path. So in this case, *TaskPH2* cannot outperform *TaskPH1* although it can predict a more precise estimation for the completion time.

7. Conclusion and Future Work

In this paper, we proposed a Look-Ahead Genetic Algorithm (LAGA) to optimize both makespan and reliability intelligently for workflow applications. We defined three heuristics to decide the priorities for a resource and a task, which can be used by LAGA. LAGA evolves only the task-resource mapping for a solution in genetic operators, and the solution's task execution order is evolved in the evaluation step using our proposed max-min strategy based on our task priority heuristics. Experiment results show that LAGA can give much better

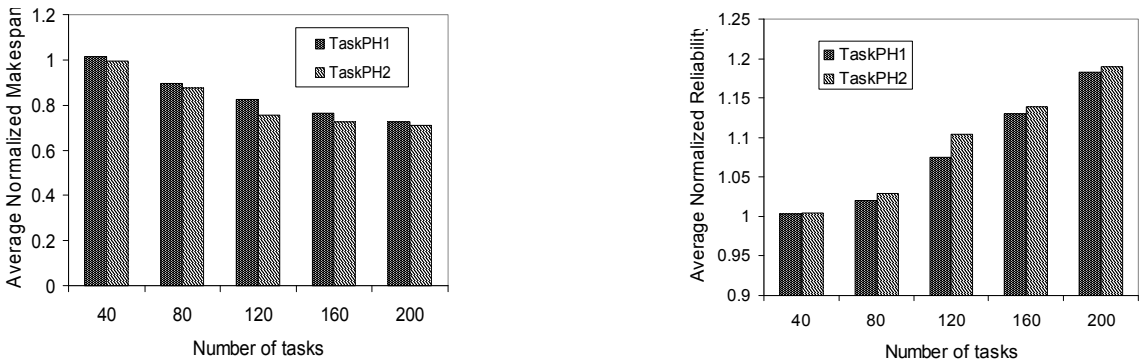


Fig. 7 Average normalized makespan and reliability of a scheduling given by LAGA using *TaskPH1* or *TaskPH2*.

quality solutions than list heuristics DLS and RDLS; moreover, it outperforms another genetic algorithm BGA in evolving scheduling solutions in terms of both iterations and time.

In future, we are going to extend our scheduling problem model to include the communication time between tasks. Our resource and task priority heuristics will also take into consideration the communication time, so that our genetic algorithm does not need to be changed to enable reliable scheduling for communication intensive applications.

References

[1] J. Dongarra, E. Jeannot, E. Saule and Z. Shi, Bi-objective Scheduling Algorithms for Optimizing Makespan and Reliability on Heterogeneous Systems. ACM Symp. on Parallelism in Algorithms and Architectures, 2007.

[2] D. Lima, Y. Onga, Y. Jinb, B. Sendhoffb, and B. Lee, Efficient Hierarchical Parallel Genetic Algorithms using Grid computing, Future Generation Computer Systems, 23(4):658-670, 2007.

[3] R. Hall, A.L. Rosenberg, and A. Venkataramani, A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. IEEE International Symposium on Parallel and Distributed Processing , 2007.

[4] L. Wang, H. J. Siegel, V. P. Roychowdhury, et al., Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, J. of Parallel and Distributed Computing, 47(1):8-22, 1997.

[5] M. Hakem and F. Butelle, Critical path scheduling parallel programs on unbounded number of processors. Int'l Journal of Foundations of Computer Science, 17(2):287-301, 2006.

[6] M.I. Daoud and N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, J. of Parallel and Distributed Computing, 68(4):399-409, 2008.

[7] T.D. Braun, H. J. Siegel, N. Beck, et al., A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing, 61(6):810-837, 2001.

[8] J. Yu, M. Kirley, and R. Buyya, Multi-objective Planning for Workflow Execution on Grids, IEEE/ACM International Conference on Grid Computing, 2007.

[9] H. Topcuoglu, S. Hariri, and M.Y. Wu, Performance-effective and low complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distributed Systems, 13(3):260-274, 2002.

[10] G.C. Sih and E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distributed Systems, 4(2):175-187, 1993.

[11] X. Wang, C. Yeo, R. Buyya and J. Su, Reliability-Driven Reputation Based Scheduling for Public-Resource Computing Using GA, IEEE 23rd International Conference on Advanced Information Networking and Applications, 2009.

[12] S. Song, K. Hwang, and Y.K. Kwok, Risk-Resilient Heuristics and Genetic Algorithms for Security-Assured Grid Job Scheduling, IEEE Trans. on Computers, 55(6):703-719, 2006.

[13] M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer. Bi-criteria Scheduling of Scientific Workflows for the Grid. IEEE International Symposium on Cluster Computing and the Grid, 2008.

[14] R.C. Corrêa, A. Ferreira, and P. Rebreyend, Scheduling Multiprocessor Tasks with Genetic Algorithms. IEEE Trans. on Parallel and Distributed Systems. 10(8): 825-837, 1999.

[15] S.C. Kim, S. Lee, and J. Hahm, Push-Pull: Deterministic Search-Based DAG Scheduling for Heterogeneous Cluster Systems, IEEE Trans. on Parallel and Distributed Systems, 18(11):1489-1052, 2007.

[16] M. Hakem and F. Butelle, Reliability and Scheduling on Systems Subject to Failures. International Conference on Parallel Processing (ICPP), 2007.

[17] A. Dogan and F. Ozguner. Bi-objective Scheduling Algorithms for Execution Time-Reliability Trade-off in Heterogeneous Computing Systems. The Computer Journal. 48(3):300-314, 2005.

[18] A. Dogan and F. Ozguner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. IEEE Trans. on Parallel and Distributed Systems, 13(03):308-323, 2002.

[19] A. Benoit, M. Hakem, and Y. Robert. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. IEEE International Symposium on Parallel and Distributed Processing , 2008.