Resource management for moldable parallel tasks supporting slot time in the Cloud

Jianmin Li[1]*
[1] School of Computer and Information Engineering, Xiamen University of Technology, Xiamen, 361024, China
[e-mail: lijianmin2006@sina.cn]
*Corresponding author: Jianmin Li

Received Novmeber 21, 2018; revised January 31, 2019; revised February 23, 2019; accepted February 14, 2019; published September 30, 2019

Abstract

Moldable parallel tasks are widely used in different areas, such as weather forecast, biocomputing, mechanical calculation, and so on. Considering the deadline and the speedup, scheduling moldable parallel tasks becomes a difficulty. Past work majorly focuses on the LA (List Algorithms) or OMA (Optimizing the Middle Algorithms). Different from prior work, our work normalizes execution time and makes all tasks have the same scope in normalized execution time: [0,1], and then according to the normalized execution time, a method is used to search for the reference execution time without considering the deadline of tasks. According to the reference execution time, we get an initial scheduling result based on AFCFS (Adaptive First Comes First Served) policy. Finally, a heuristic approach is used to improve the performance of the initial scheduling result. We call our method HSRET (a Heuristic Scheduling method based on Reference Execution Time). Comparisons to other methods show that HSRET has good performance in AWT (Average Waiting Time), AET (Average Execution Time), and PUT (Percentages of Unfinished Tasks).

Keywords: moldable parallel tasks; resource management; slot time; normalized execution time;

1. Introduction

With the development of Cloud computing, parallel tasks have widely used in different areas [1]. The moldable parallel task is one of the most important parallel tasks which makes the user not give attention to the DAG (Directed Acyclic Graph) of the program. The user just gives some parameters to the program, and gets the result in a short amount of time [2]. The moldable parallel task model [2] has widely been used in diverse areas, such as cognitive computing technology [4], weather forecast [5,6], mobile computing, biocomputing, mechanical calculation, and so on [7].

The problem of scheduling of moldable parallel tasks needs to take multiple aspects into consideration, such as the deadline of parallel tasks [8], system load, the speedup of tasks, network [9], dependency of different parallel tasks [7, 9], and so on [7, 8, 9]. Sometimes, those aspects are conflicting with each other. For example, if a parallel task has a higher value in the parallelism (giving more resources to the parallel task), it shortens the execution time; but at the same time (according to Amdahl's law [10]), it consumes more resources and makes other tasks have to reduce the resources. Sometimes, that would make other tasks not be finished before their deadlines. Furthermore, the non-linear relation between the parallelism and the execution time makes the scheduling problem more difficult than other environments [5].

Researchers have done much work for the scheduling of parallel tasks in different environments, such as Grid, cluster [11], multi-core system [12], and cloud environment [8]. The scheduling targets include minimizing the makespan (or execution time) [13], reducing the cost [14], saving the energy consumption [15] and so on. Those scheduling methods include approximation algorithm [3, 7], agent-based algorithm [16], iterative approach and other methods. Recently, Map-Reduce [17] is also used to schedule moldable parallel tasks.

That work either supposes that the task has a forecast speedup under different parallelisms [19], or supposes that we know the details of the DAG of the parallel task. However, it is very difficult to get an accurate forecast of the speedup under different parallelisms [20]. Even we know the detail of the DAG, how to schedule it is also a very challenging problem. Those two aspects bring a negative effect on those scheduling methods and make the scheduling method more difficult.

Different from past work, we do not take account of the DAG of the program, which is very difficult to control for the scheduler. We just take account of the speedup of the parallel tasks and the system load of the system. Based on the system load and the speedup with different numbers of resources, we propose our scheduling method.

Main contributions of our paper focus on:

(1) we give a system model for moldable parallel tasks, which support slot time, speedup and so on;

(2) we try to normalize the execution time of different parallel tasks to make them have a same scope of [0,1];

(3) we propose a scheduling for the moldable parallel tasks;

(4) we compare our method with other methods in different environments, especially when the system has an inaccurate value in the speedup.

The framework of our paper is introduced as follows. Section 2 gives a literature review of the scheduling methods for parallel tasks, especially for moldable parallel tasks. Section 3 provides an example for moldable parallel tasks-a program for weather forecast. Section 4 addresses our proposed system architecture and the related model, and at the same time, it also gives a deep analysis to the system. Section 5 gives a scheduling method for the scheduling of moldable parallel tasks. Section 6 describes the simulation performance of our proposed methods and existing methods. We concluded our study and further work in Section 7.

2. Related Work

D. G. Feitelson et al., "[2] referred to tasks with a fixed number of processors (parallelism) as rigid tasks, tasks that can be resized only at launch time as moldable parallel tasks, and tasks that can grow or shrink at runtime as the malleable task. In other words, jobs with a fixed parallelism, if the parallelism is set at the beginning and never can be changed during the execution, the task belongs to moldable tasks; if the parallelism can be changed at any time, the task belongs to malleable parallel task.

For the scheduling of parallel tasks, most methods are proposed by smartly selecting the route and time of the sub-tasks in the DAG of those parallel tasks. Those methods always try to shorten the execution time. D. Sánchez et al., "[16] proposed an agent-based architecture to manage and execute independent parallel tasks on a dynamic network. They introduced an application on their proposed architecture to support the execution of a complex knowledge acquisition task by an adequate load balancing policy. W. YiRong et al., "[11] divided the entire scheduling process of scheduling MOWS (Mixed-parallel Online Workflow Scheduling) into four phases: task prioritizing, waiting queue scheduling, task rearrangement, and task allocation. They developed four new methods: shortest-workflow-first, priority-based backfilling, preemptive task execution and All-EFT (ALL Early Finished Task First) task allocation, for scheduling NOWS tasks in speed-heterogeneous multi-cluster environments. L. Keqin [21] investigated the problem of non-clairvoyant scheduling of independent parallel tasks on single and multiple multicore processors. For a single multicore processor, they used LTF (Largest Task First) to get the asymptotic worst-case performance bound for a non-clairvoyant offline scheduling algorithm. For multiple multicore processors, he used RTF (Random Task First) for a non-clairvoyant online scheduling problem. R. M. Pathan et al., "[22] proposed a two-level GFP (preemptive Global Fixed-priority scheduling Policy) for the scheduling of a real-time parallel application that is modeled as a collection of parallel and recurrent tasks on a multicore platform: a task-level scheduler first determines the highest-priority ready task and a subtask-level scheduler selects its highest-priority subtask for execution. Q. Wang et al., "[23] proposed a new parallel job scheduling method based on a classification method of resources from different attributes (including the memory, bandwidth, CPU)-CPLMT (Cloud Parallel scheduling based on the Lists of Multiple Attributes). The classification method categorized resources into different kinds according to the number of resources that satisfied the job from different attributes of the resource, such as the speed of the resource, memory and so on. Most interference-based analysis techniques are not directly applicable to parallel programming model, so, H. S. Chwa et al., "[24] extended the notion of interference to capture thread-level parallelism more accurately. They leveraged parallelism-aware interference to derive efficient EDF (Earliest Deadline First) schedulability tests that are directly applicable to parallel task models, including DAG models on multi-core platforms. Those methods always target to shorten the execution time by smartly selecting the execution route of DAG of the parallel tasks.

Some scheduling methods not only try to shorten the execution time, but also try to improve performance of other aspects, such as reducing peak memory [27], the energy consumption [28, 29, 30] and so on. K. Enver et al., "tried to reduce the peak memory in a parallel execution environment. They modeled the tasks as a DAG and targeted to find a topological ordering which has the maximum number of cut edges at any point. The vertices and edges have weights, and the aim is to minimize the maximum weight of cut edges in addition to the weight of the last vertex before the cut. B. Mahmood et al., "[28] addressed the real-time scheduling problem of parallel tasks on a performance asymmetric multicore processor with multiple cores targeting to reduce the power consumption. Based on DVFS (Dynamic Voltage and Frequency Scaling) technology, they used parallel EDF - first divides the tasks into *m* segments and then executes these distributed tasks in EDF fashion. H. F. Sheikh et al., "[29] proposed MOEA (a Multi-Objective Evolutionary Algorithm), which tries to determine Pareto optimal solutions with simultaneous optimization of performance (P), energy (E), and temperature (T). Their work included problem-specific solution encoding, determining the initial population of the solution space, and the genetic operators that get efficient solutions in a short amount of time. They presented a methodology to select one solution from the Pareto front according to the user's preference. M. Shojafar et al., "[30] proposed an energy-efficient adaptive resource scheduler for NetFCs (Networked Fog Centers). It is operated at the edge of the vehicular network and are connected to the served VCs (Vehicular Clients) through I2V (Infrastructure-to-Vehicular) TCP/IP-based single-hop mobile links. Taking account of the locally measured states of the TCP/IP connections, they try to maximize the overall communication-plus-computing energy efficiency, and meet the application-induced hard QoS (Quality of Service) requirements on the minimum transmission rates, maximum delays and delay-jitters. Those methods always try to consider multiple targets of scheduling, and based on the multiple targets, they propose different methods from different aspects. Other methods also try to consider the speedup of parallel tasks and ignore the DAG (Directed Acyclic Graph) of the parallel tasks. They always improve the performance by smartly selecting the speedup of parallel tasks. Hao et al., "[25] considered the scheduling of parallel tasks in multi-Cloud environment. They categorized jobs into different lists according to the waiting time of the jobs and every job has different parallelisms. At the same time, a new method-ZOMT (the scheduling parallel tasks based on ZERO-ONE scheduling with Multiple Targets) is proposed to solve the problem of scheduling parallel jobs. M. Beji et al., "[26] tried to schedule parallel application by resizing the application and finding the appropriate sub-platform with the optimal number of resources (clusters, processors) from the original platform. There are three steps in the scheduling: firstly, determining of the computing clusters; secondly, determining the optimal number of processors in each cluster; finally placing the tasks on the appropriate processors. Though those methods take account of the speedup, all of them always neglect the speedup which is not accurate in the scheduling. Most important of all, our moldable parallel tasks support slot time, which brings users much convenience, such as checking the error points, getting the phrase result, and so on.

3. An example for moldable parallel task model

In this section, we will give an example for the moldable parallel task, and then, we give the model of the moldable parallel task.

3.1 Moldable parallel task model

We model the moldable parallel task t_i $(1 \leq i \leq I)$ as:

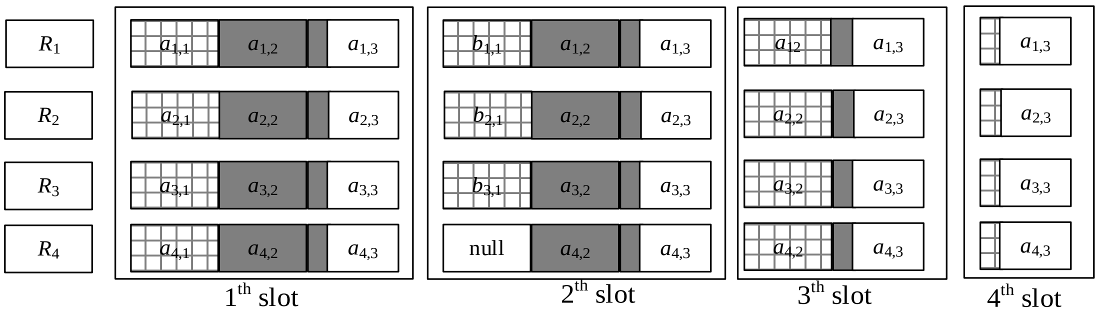
$$t_i = \{ar_i, dl_i, < pl_i^j, et_i^j, sp_i^j >, minp_i, maxp_i\} \; (minp_i \leq j \leq maxp_i) \qquad (1)$$

ar_i is the arrival time, dl_i is the deadline. pl_i^j and et_i^j is the jth parallelism and the related execution time $(pl_i^j < pl_i^{j+1})$. sp_i^j is the speedup when the parallelism of the task is pl_i^j. $minp_i$ and $maxp_i$ is the minimum parallelism and the maximum parallelism of the task. I is the total number of tasks. In formula (1), the execution time et_i^j is the time executed on a SR (standard resource).

3.2 Moldable parallel task supporting slot time model

The moldable parallel task always needs much time to execute them, and sometimes, there are some errors from the task or the computing resources, the task may be broken in the execution. So, slot time is used to support moldable parallel tasks. Slot time not only helps us to check the error that happens in the execution, but also offers us an easy way to schedule those tasks. During the slot time, the task holds the allocated resources exclusively, until the end of every slot time. At the end of the slot time, the task can give up the execution right even if it is not finished.

Fig. 1 gives an example of the scheduling moldable tasks. The rectangle with small squares is the sub-task that has been finished. The black rectangle is the sub-task that has not been scheduled. There are four resources and the task comes first, which is paralleled by 4*3 small tasks. The execution time of every task is a *SL* (Slot Time). In the first slot, the task *a* comes and gets 4*1 slots time. Then the task *b* comes, which has a short time to the deadline, so, it has to be executed with the parallelism of 3. After that, in the time for the two slots time to come, the rest of tasks of *a* $(a_{i,j}, i = 2,3)$ are executed.



R_1
R_2
R_3
R_4
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{4,1}$ $a_{4,2}$ $a_{4,3}$
1th slot
$b_{1,1}$ $a_{1,2}$ $a_{1,3}$
$b_{2,1}$ $a_{2,2}$ $a_{2,3}$
$b_{3,1}$ $a_{3,2}$ $a_{3,3}$
null $a_{4,2}$ $a_{4,3}$
2th slot
$a_{1,2}$ $a_{1,3}$
$a_{2,2}$ $a_{2,3}$
$a_{3,2}$ $a_{3,3}$
$a_{4,2}$ $a_{4,3}$
3th slot
$a_{1,3}$
$a_{2,3}$
$a_{3,3}$
$a_{4,3}$
4th slot

Fig. 1. An example for moldable parallel tasks

3.3 A motive example for scheduling of moldable parallel task supporting slot time model

In this section, we will give an example to schedule 4 tasks (*a*, *b*, *c*, *d*) on five resources (R_1~R_5). Suppose that all those tasks arrive at the beginning. **Table 1** gives attributes of those four tasks (the deadline and the arrival time denoted by the number of slot times),

Table 1. An example of scheduling of moldable parallel tasks

Task	deadline	Arrival time	<parallelism, execution time>	Available selection
a	5	0	<1,4> <2,3> <3,3>	<1,4> <2,3> <3,3>
b	7	0	<1,8><2,5><3,4>	<2,5><3,4>
c	6	0	<1,7><2,4><3,3>	<2,4><3,3>
d	2	3	<1,5><2,3><4,2>	<4,2>

The tasks *a*, *b* and *c* arrive at the beginning time, and the task *d* arrives at the 3th slot time (**Fig. 2**). Just considering the deadline, the job can select the parallelism in column 5 of **Table 1**. To reduce the consumed resources, we select the parallelism with the smallest parallelism that meets the requirement. So, the parallelisms of *a*, *b* and *c* are 1 ($a_{1,1}$~$a_{1,4}$), 2 ($b_{1,1}$~$b_{2,5}$) and 2 ($c_{1,1}$~$c_{2,4}$) , respectively. When the task *d* comes, it can only be paralleled as the parallelism is 4 to meet the deadline. So, the sub-tasks of *b* ($b_{1,4}$, $b_{2,4}$, $b_{1,5}$, $b_{2,5}$) and *c* ($c_{1,4}$, $c_{2,4}$) are suspended. After the execution of *d*, the sub-tasks of *b* and *c* are beginning to be executed again.

In fact, we can also schedule tasks with other kinds of scheduling policies: task with different parallelisms and different orders. To the best of our knowledge, only our method can ensure the four tasks are finished before their deadlines. But, we also find that we always make the task consume the lowest resources, and if the system is under a low load, it may increase the execution time.

From the above-mentioned example, we find in the scheduling of moldable parallel tasks:

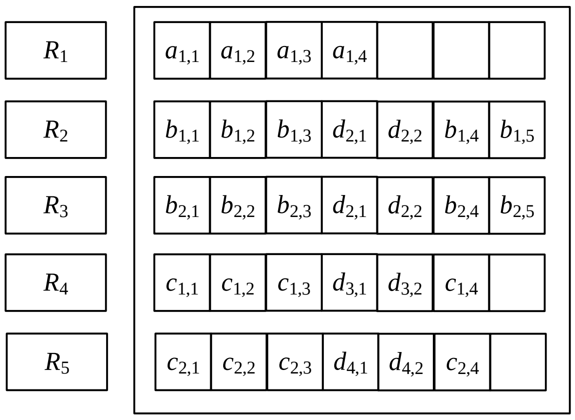
(1) the parallelism of the moldable parallel task cannot be changed in the execution;

(2) the sub-tasks of a task can be suspended during the execution (at the end of every slot time);

(3) the selection of parallelisms is decided by the system load;

(4) the scheduling order of the sub-tasks of a task can be changed during execution.

So, there are two main steps in the scheduling of malleable parallel tasks: (1) assigning the parallelism, and (2) deciding the scheduling order of sub-tasks.



R_1
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$
R_2
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $d_{2,1}$ $d_{2,2}$ $b_{1,4}$ $b_{1,5}$
R_3
$b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $d_{2,1}$ $d_{2,2}$ $b_{2,4}$ $b_{2,5}$
R_4
$c_{1,1}$ $c_{1,2}$ $c_{1,3}$ $d_{3,1}$ $d_{3,2}$ $c_{1,4}$
R_5
$c_{2,1}$ $c_{2,2}$ $c_{2,3}$ $d_{4,1}$ $d_{4,2}$ $c_{2,4}$

Fig. 2. Scheduling moldable parallel tasks supporting slot time

4. An analysis to the system

In this section, first, we will present the system framework, and then, we will give the method to decide the parallelism and the related scheduling order of every sub-task of scheduled moldable parallel tasks.

4.1 System framework

Our study takes a simulation model to address performance issues associated with the task that can be parallelized by different numbers of sub-tasks with different execution time. **Fig. 3** gives the module that is more related to our scheduling of moldable parallel tasks. There are four phases for the scheduling of parallel tasks [26]: submission (Phase 1), mapping or task-to-node allocation (Phase 2), parallelism decision (Phase 3), and sub-task scheduling (Phase 4). Submission is the interface for users to submit their tasks. The users submit the task and the related information of the task, such as the deadline, the speedup under different parallelisms and so on. According to the system load of every cluster, mapping (or task-to-node allocation) decides to allocate the coming tasks to which clusters. Parallelism decision decides the number of assigned resources according to the system load, the deadline of jobs, and other requirements. The third phase decides the parallelism which ensures that the system meets the global load: (1) to maximize the number of finished tasks (2) to minimize the average execution time. Phase 4 decides the scheduling of sub-tasks. Because we can not forecast the system load with no errors, sometimes, we do not schedule those tasks as FCFS (First Come First Service) policy. We may give some sub-tasks higher priorities of being executed. The main models include: *Accounting*, *SLM* (System Load Monitoring), *PDS* (Parallelism Decision Scheduler), *STS* (Sub-tasks Scheduler), *LS* (Local Scheduling of sub-tasks).

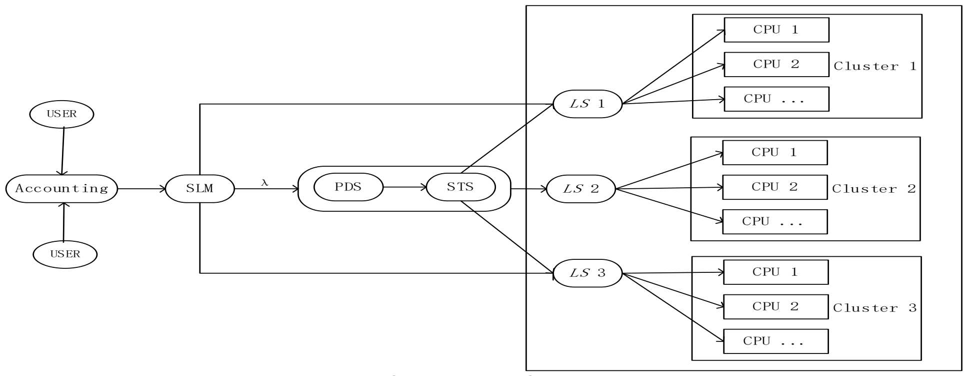


Fig. 3. Scheduling framework of moldable parallel tasks

Accounting: a user submits a moldable parallel task to the system through the *Accounting* tool. The user not only submits the code and the related data of the job, but also submits the relation between the parallelism and the execution time of the task. *Accounting* also holds the responsibilities such as ensuring every task is a secure task, recording the behavior of users, and so on. In summary, *Accounting* gives the interface for users to submit their tasks and the related information to tasks to the system.

SLM: *SLM* forecasts the system load information according to load information of all clusters. *LS* (Local Scheduler 1~3) reports the system load to *SLM* and *SLM* forecasts the system load according to the report from *LS*. *SLM* decides whether permits the new coming task to enter the system according to the forecast of the system load.

PDS: according to the forecast system load, *PDS* decides the number of assigned resources. *PDS* decides the parallelism of the task according to: (1) the system load, with the system load increasing, the number of resources drops; (2) the relation between the execution time and the parallelism, which is decided by the task. *PDS* gives the parallelism to those tasks and the parallelism can not be chanage during the execution.

STS: according to the load information from different clusters, such as *LS* 1 and *LS* 2, *STS* decides to assign the parallel sub-tasks to which clusters. *STS* also considers the system load and the deadline, and decides the scheduling order of all parallel sub-tasks. Sometimes, the scheduling order of the parallel sub-tasks may not be FCFS. Some sub-tasks can be brought forward because other sub-tasks have deadlines close to the deadline.

Under the model, the four phrases (submission, deciding parallelism, task-to-cluster allocation, sub-task scheduling) work together to schedule resources. At first, we estimate the system load according to the system load of different clusters (submission), and then according to the speedup of parallel tasks and other requirements (deadline) to the tasks, decide the number of assigned resources (deciding parallelism). The speedup of the parallel task gives the relation between the number of assigned resources and the execution time. After we get the speedup of different tasks, we decide to assign how many resources to tasks (task-to-cluster allocation) and schedule sub-tasks accordingly (sub-task scheduling). *LS* is in charge of the local resources and ensures the task can be finished before its deadline even we can not get an accurate speedup. *LS* is in charge of the resource virtualization, including the role for VM consolidation, resource allocations and monitoring of the request scattered in various data centers. If the system load is very low, and it can shut down some VMs to enhance the resource utility. On the contrary, if it finds the system has a higher system load, it may use the DVFS [29] technology to make the CPU work with a higher speedup.

From the system model, we know that there are two main problems in the scheduling: how to decide the parallelism and how to schedule those sub-tasks of all parallel tasks.

4.2 An analysis of the system

As we know that, the parallelism is decided by the system load. If the system has a low system load, we can give the task more resources to shorten the execution time; on the contrary, if the system has a high load, we can just assign the resources to ensure the task can be finished before its deadline. We call the parallelism reference parallelism which makes every task have the same normalized execution time, and every task has the minimum execution time without considering the deadline and other requirements.

The problem is the system load is dynamic especially for moldable parallel tasks. There are two kinds of system loads: maximum system load (*maxsl*) and minimum system load (*minsl*). *maxsl* and *minsl* are the system loads when all tasks have the maximum parallelism ($minp_i$ in formula (1)) or minimum parallelism ($maxp_i$ in formula (2)).We calculate the system load under the two assumptions: (1) the task has no deadline; (2) the system load is average allocated to the time from the task arrival to the deadline. We suppose that there are *TN* tasks which arrive before the slot time *now* and have a deadline more than *now*.

$$Jset = \{J_{aid} | TN \geq aid \geq 1\} \tag{2}$$

The processing ability of the system is *TP* (which is denoted by the number of standard computing resources). Consumed resources that when the system has the maximum parallelism ($crmax$) and minimum parallelism (*crmin*) are:

$$crmax = \sum_1^{TN} \frac{pl_i^{jmax} * et_i^{jmax}}{(dl_i - ar_i) * TP} \tag{3}$$

$$crmin = \sum_1^{TN} \frac{pl_i^{jmin} * et_i^{jmin}}{(dl_i - ar_i) * TP} \tag{4}$$

Where, $jmax = maxp_i$ and $jmin = minp_i$.
So:

$$minsl = min\ (crmin/TP\ , 1) \tag{5}$$

$$maxsl = max\ (crmax/TP\ , 1) \tag{6}$$

In formulas (5) and (6), the functions *max* and *min* return the maximum and the minimum.

There are three cases in the scheduling:

Case 1: $maxsl = 1$. In this case, we can allocate as many resources as possible to the task. In other words, the task can be executed with the maximum parallelism.

Case 2: $minsl = 1$. In this case, even every task is executed under minimum parallelism, there are some tasks that cannot be finished as request.

Case 3: $minsl \leq maxsl \leq 1$. In this case, we need to decide the parallelism of different tasks.

In case 1, we have no problem in the scheduling; and in Case 3, we have no methods to meet the scheduling requirement. So, our research focuses on Case 2. First, we need to normalize the execution time of parallel tasks to make them have a scope of [0, 1]. According to formula (1), we set the normalized execution time et_i^j of the task t_i as net_i^j:

$$net_i^j = (et_i^j - min\ (et_i^*))/(max\ (et_i^*) - min\ (et_i^*)) \tag{7}$$

et_i^* is the set of execution time under different parallelisms for the task t_i. $max\ (et_i^*)$ and $min\ (et_i^*)$ are the maximum and minimum execution time.

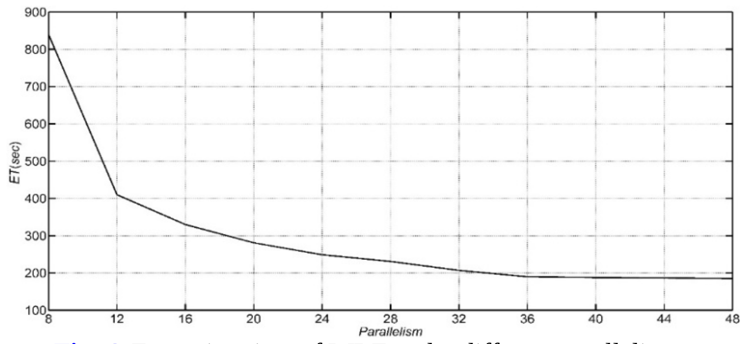


Fig. 4. Execution time of WRF under different parallelisms

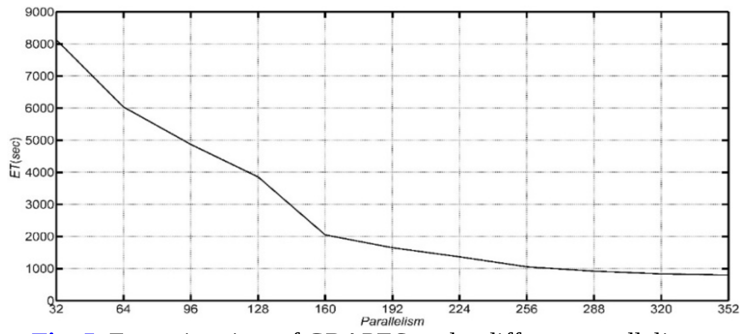


Fig. 5. Execution time of GRAPES under different parallelisms

Fig. 4 and **Fig. 5** give the execution time of WRF (Weather Research and Forecasting model) and GRAPES (Global/Regional Assimilation and Prediction Enhanced System) under different parallelisms (the two models are executed on the same configuration). WRF and GRAPES are widely used in the weather forecast. We can find that: (1) the execution time has changed with different scopes; (2) the execution time has different trends; (2) GRAPES and WRF have different scope of parallelism. Those differences make how to decide parallelism a difficulty. So, to ensure the execution time has the same scope, formula (7) is used to normalize the execution time. **Fig. 6** and **Fig. 7** are the normalized execution time of GRAPES and WRF. We find that the normalized data has the same trend with the original data.

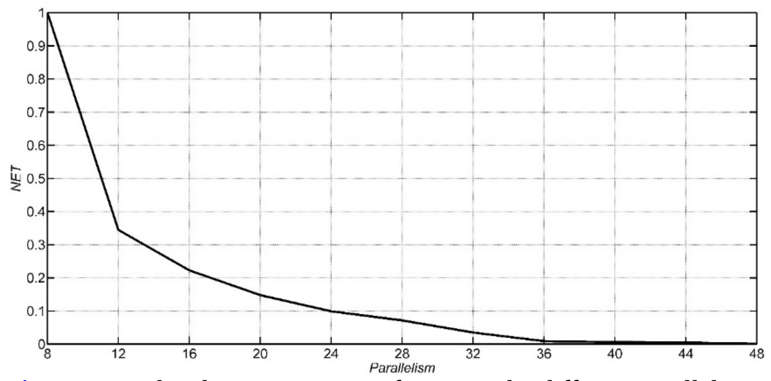


Fig. 6. Normalized execution time of WRF under different parallelisms

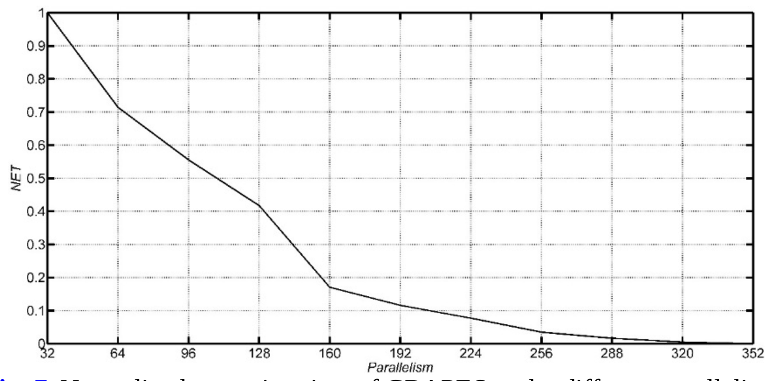


Fig. 7. Normalized execution time of GRAPES under different parallelisms

5. Scheduling method for mold parallel tasks

5.1 Resource scheduling for moldable parallel tasks to decide normalized execution time

We know that, the system load and the deadline of tasks play an important role to decide the parallelism. First, we need to get the available parallelism that ensures the task can be finished before its deadline. For the task t_i, suppose that the scope of available parallelisms is [$avip_i$, $maxp_i$] ($avip_i \geq minp_i$). Using formula (7), the available parallelism can be denoted as [$navip_i$, 1]. When the task t_i is executed under the state $< pl_i^{jsel}, et_i^{jsel}, sp_i^{jsel} >$(the parallelism pl_i^{jsel}, the execution time et_i^{jsel}, and the speedup sp_i^{jsel}), the task average adds the load to every slot of the system and it is denoted by ld_i^{jsel}:

$$ld_i^{jsel} = \frac{pl_i^{jsel} * et_i^{jsel}}{(dl_i - ar_i) * T} \quad (8)$$

T is the computing ability of all resources of every slot. We suppose that every task has the same normalized execution time, denoted by $avgpl$. Algorithm 1 gives the method to get the normalized execution time:

Algorithm 1: Getnet(T, *I*, *Nslot*) // gets the normalized execution time, T is the total processing ability of all resources, *I* is the number of tasks, *Nslot* is the total number of slots in the scheduling;

```
1:  $snet = 1$; // suppose the selected normalized execution time is 1
2:  $beget = 0$, $endet = 1$; // change the normalized parallelism from 0 to 1 without taking account of the deadline;
3:  $minstep = \frac{1}{\text{maxet}(minp_i) - \text{minet}(maxp_i)}$; // get the minstep;
4:  $eget = 0$;
5:  While ($endet - beget$)> $minstep$ do
6:      For $i = 1{:}I$ do
7:          According to the normalized execution time to get the execution time;
8:          According to the execution time to get the execution state of different tasks;
9:          Suppose $t_i$ average adds load to every slot according to formula (8);
            //suppose the normalized execution time is $snet$;
10:     Get the minimum load minl of every slot;
11:     If minl>1 then
12:         $endet = (endet + beget)/2$;
13:     Else
14:         $beget = (endet + beget)/2$;
15:     $snet = (endpl + begpl)/2$;
16: Get the execution state of different tasks when the normalized execution time is $snet$.
```

The main idea of Algorithm 1 is using binary search to find the best normalized execution time, which is the maximum parallelism which ensures that every slot is not overloaded (less than 1). Line 1 supposes that the normalized execution is 1. Line 2 supposes that the available normalized execution has the range of [0,1]. $beget$ and $endet$ are lower bound and upper bound of execution time. Line 3 gets the minimum step ($minstep$). $\text{maxet}(minp_i)$ returns to the maximum execution time when every task has the lowest parallelism. $\text{minet}(maxp_i)$ returns to the minimum execution time when every task has the highest parallelism. Lines 5-14 get the parallelism until the range is less than the minimum step $minstep$. Lines 6-9 calculate the load of every slot when the normalized execution is $snet$. First, we get the execution time according to the normalized execution time (line 7), then we get the execution state of different tasks (line 8), and lastly, the task average adds load to every slot. Line 10 gets the minimum load of every slot ($minl$). Lines 11~14 check whether the load $minl$ is more than 1. If it is, the upper bound $endet$ becomes $(endet + beget)/2$ (line 13); otherwise, the lower bound $endet$ becomes $(endet + beget)/2$ (line 15). Line 16 returns to the execution state of different tasks.

5.2 Resource scheduling for moldable parallel tasks to decide execution time considering deadlines

Algorithm 2: Selet(I, $snet$) // I is the number of tasks, $snet$ is the selected normalized execution time.

```
1:  Get unscheduled tasks ($unt$) when all tasks are executed under normalized execution time $snet$
    under AFCFS policy;
2:  Sort tasks in $unt$ as the ascending order of the arrival time;
3:  For every task $t_i$ in $unt$ do
4:      Get tasks which arrive before $t_i$ and have the deadline that is less than the deadline of $t_i$,
        denoted by $lft_i$;
5:      Sch1($lft_i$, $t_i$);
6:      If Checksch($t_i$) then
7:          Schedule $t_i$;
8:      Else
9:          Get tasks which arrive after $t_i$ and have the deadline that is less than the deadline of $t_i$,
            denoted by $midt_i$;
10:         Sch2($midt_i$, $t_i$);
11:         If Checksch($t_i$) then
12:             Schedule $t_i$;
13:         Else
14:             Get tasks which arrive after $t_i$ and have the deadline that is more than the deadline of $t_i$,
                denoted by $rgt_i$;
15:             Sch3($rgt_i$, $t_i$);
16:             If Checksch($t_i$) then
17:                 Schedule $t_i$;
18:             Else
19:                 drop $t_i$;
```

In fact, some tasks may not be finished as our request under the selected normalized execution time because of the deadline. So, we should give details to find the right execution time to meet the deadline. Algorithm 2 is used to get the selected execution time of every task. We get the unscheduled tasks unt (line 1), and then re-schedule the tasks in unt. In Algorithm 2, I is the number of tasks, $snet$ is the selected normalized execution time.

As shown in Fig. 8 and Algorithm 2, jobs arrive before Ar_i and have a deadline which is less than Dl_i, denoted by lft_i, which would be scheduled as Sch1(rgt_i) (line 5); tasks arrive after Ar_i and have a deadline which is less than Dl_i, denoted by $midt_i$, which would be scheduled as Sch2($midt_i$) (line 10); tasks arrive after Ar_i and have a deadline which is more than Dl_i, denoted by rgt_i, which would be scheduled as Sch3(rgt_i) (line 15). Checksch(t_i) checks whether the task t_i can be scheduled before the deadline.

Algorithm 3: Sch1(lft_i, t_i); // trying to scheduling the task t_i based on reschedule lft_i

```
1:  $stet = snet$; // $snet$ records selected normalized time
2:  $savr = 0, minr = 0, nstet = snet$; // $savr$ records saved computing resources, $minr$ records
    the minimum $savr$ under different normalized execution time;
3:  While $t_i$ cannot be allocated do
4:    For every task $temp \in lft_i$ do
5:      If $temp$ can be allocated before $Ar_i$ when the normalized execution time is $nstet$, then
6:        Calculate the saving computing resources $nsavr$ for the task $t_i$;
7:        $savr = savr + nsavr$;
8:      Else // for tasks that cannot be finished before $Ar_i$
9:        $temp1 = snet, temp2 = snet$;
10:       While $temp1 > minstep$ do
11:         $temp1 = temp1 - minstep$;
12:         Get the minimum saving resource for reducing normalized execution time ($sa1$);
13:       While $temp2 < 1$ do
14:         $temp2 = temp2 - minstep$;
15:         Get the minimum saving resource for reducing normalized execution time ($sa2$);
16:       Comparing $sa1$ and $sa2$, and selecting the bigger (as $sa$);
17:   If $(savr + sa) < minr$ then
18:     $minr = savr, stet = nstet$;
19:   $stet = stet - minstep$;
```

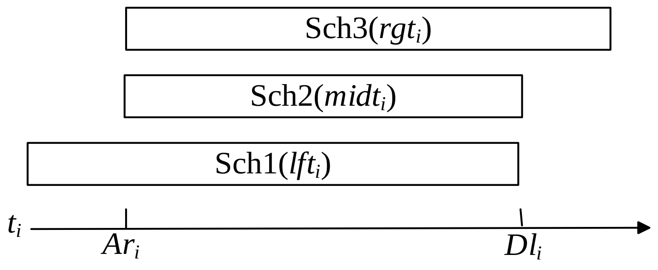


Fig. 8. The scheduling methods for different kinds of tasks

For different kinds of jobs (lft_i, $midt_i$ and rgt_i), we give different policies.

For jobs in rgt_i, if the job can be finished before Ar_i, we always select the maximum execution time but ensure they can be finished before Ar_i (lines 6~9 Algorithm 3, the same for the following); otherwise, two methods – enhancing (lines 15~18) or reducing (lines 11~14) the normalized execution time are used to check which has the minimum saving computing resources. Algorithm 3 gives the details. $snet$ records selected normalized time. $savr$ records saved computing resources, $minr$ records the minimum $savr$ under different normalized execution time. First of all, we suppose that the jobs are executed within the normalized execution time, to check whether they would meet the deadline (lines 3~5). Then, for jobs that can be finished before the arrival time of t_i (Ar_i), we reduce the normalized execution time with a step of $minstep$ (line 6, Algorithm 3), to calculate the saving computing resources ($nsavr$) (line 7), until it can meet the deadline. For the task that cannot be finished before Ar_i, enhancing (lines 10~12) and reducing normalized execution time (lines 13~15) are used to get the minimum saving execution time, and we select the policy which saves more computing resources (line 19). At last, we choose the solution which saves more computing resources.

For jobs in $midt_i$, our policy is simple, we just want to enhance the normalized execution time, and to reduce the system load. The reason is, non-linear speedup moldable parallel tasks enhance the computing requirement with the enhancement of speedup (reducing the normalized execution time). The algorithm for $midt_i$ is simple, so, we do not give it (Sch2($midt_i$)) in details.

Algorithm 4: Sch3(rgt_i)

```
1:  $stet = snet$; // $snet$ records selected normalized time
2:  $minr = 0$; // $minr$ records the minimum $savr$ under different normalized execution time;
3:  While $t_i$ cannot be allocated do
4:    $savr = 0$, $asr = 0$;// $savr$ and $asavr$ record the saving computing resources before the
      deadline $Dl_i$ and after $Dl_i$
5:    While $temp1 < 1$ do
6:        For every task $temp \in rgt_i$ do
7:          If $temp$ can be finished before $Dl_i$ when the normalized execution time is $nstet$ then
8:            Calculate the saving computing resources $nsavr$ for the task $t_i$ before $Dl_i$;
9:            Calculate the saving computing resources $asavr$ for the task $t_i$ after $Dl_i$;
10:           $savr = savr + nsavr$;
11:           $asr = asr + asavr$;
12:       $temp1 = temp1 + minstep$;
13:   If ($avr < minr$) and ($asr > 0$) then
14:       $minr = savr, stet = nstet$;
15:       If Checksch($t_i$) then
16:         Schedule $t_i$;
```

For jobs in rgt_i, the scheduling method must ensure: (1) reducing the workload before Dl_i (reduce the workload between Ar_i and Dl_i for the unfinished task t_i) and (2) after Dl_i (reducing the workload after Dl_i for the future unfinished jobs). For (1), we must ensure reducing the workload until the task t_i can be executed as requested. Different from lft_i, which is to minimize saving computing resources. Because we do not want to enhance the execution time if there are enough resources supplement. For (2), we just ensure that the task t_i consumes less resources when it is executed under normalized execution time. Algorithm 4 gives the details of scheduling those tasks. In line 1, $snet$ records the selected normalized time. $minr$ records the minimum $savr$ under different normalized execution time in line 2. $savr$ and $asavr$ record the saving computing resources before the deadline Dl_i and after Dl_i (line 4, Algorithm 4). We check unscheduled task in rgt_i (lines 3~16), enhancing the normalized execution with a step of $minstep$ (line 12), to get the saving computing resources before the deadline $Dl_i(nsavr$, line 8) and after Dl_i ($asavr$, line 9). Line 8 calculates the saving computing resources $nsavr$ for the task t_i before Dl_i. Line 9 calculates the saving computing resources $asavr$ for the task t_i after Dl_i. We check whether the scheduling solution has minimum saving computing resources (line 15), and at the same time, consumed less resources after Dl_i (line 13). If it is, we check whether the task t_i can be scheduled (lines 15~16). If it is, we schedule it (line 16).

6. Simulations and comparisons

In the simulation, we will compare our method HSRET with CBSP [26], All-EFT [11] and AFCFS (Adapted First Come First Served) [5, 23, 25]. We have introduced CBSP and All-EFT in Section 2. AFCFS is widely used in many systems, and we suppose that AFCFS schedules parallel tasks as “First Come First Served” policy, and randomly select one cluster which ensures the parallel task can be finished before its deadline. We suppose AFCFS always prefer to select the minimum parallelism that ensures the tasks can be executed before their deadlines.

6.1 Simulation environment

In our simulation, there are 10 clusters, each of which has 1000 computing nodes (CPU). Because the sub-tasks of a parallel task always need to exchange information with each other, a task can not be parallelly executed to two clusters. In other words, a task can only be parallelly executed in one cluster. We model our simulation as **Fig. 3**. There are 10 *LC*s (*LC*1~*LC*10) in our system. Most of clusters only have a few CPUs (1~3)(http://www.cs.huji.ac.il/labs/parallel/workload), so we suppose there are only one kind of CPUs in one cluster. All nodes in one Cluster have the same computing capacity and the computing speed of every node is [0.8, 1.2] times to a standard computing resource. In our simulation, there are two kinds of parallel tasks in the scheduling: WRF and GRAPES. The two kinds of tasks have the same ratio in the number of tasks belonging to WRF and GRAPES. We suppose that the speedup has the same value to WRF or GRAPES (in **Fig. 4** or **Fig. 5)**. Suppose that the length of every parallel task obeys uniform distribution in [500, 9500] minutes (under the minimum parallelism for WRF or for GRAPES) when they are parallelly executed on some standard resources (with minimum parallelism in Figs. 4 and 5). According to the speed of selected CPUs and the parallelism, we get the execution time. The deadline of every parallel task is a random in [1.5, 5] times of the execution time when the parallel task has the minimum parallelism. To get the accurate execution time under different parallelisms of the parallel tasks is very difficult, so we will investigate our scheduling method in different accuracies of the forecast of execution time (ρ), 100%, 95%, 90%. “100%” means that we can get an absolutely accurate value in forecasting execution time under different parallelisms. “95%” means that if we forecast the execution time is 1, then the actual execution time would be changed in [1, 1.05]. “90%” means that if we forecast the execution time is 1, then the actual execution time would be changed in [1, 1.1].

6.2 Comparison and discussion

In the simulation, we will compare four methods in AET (Average execution time), AWT (Average waiting time), and PUT (percentage of unfinished tasks before deadline). We will evaluate two kinds of slot time: 360 minutes and 720 minutes. We consider the system when it has a relatively high system load, so when the slot time is 180 (*mins*), the arrival rate λ is changed from 100 to 150, with a step of 10; when the slot time is 360 (*mins*), the arrival rate is changed from 200 to 300, with a step of 20. We suppose that the number of arriving tasks in every slot time is a random in [1, 2*λ-1]. We will test those methods in 1000000 slot time.

6.2.1 Comparison of AET

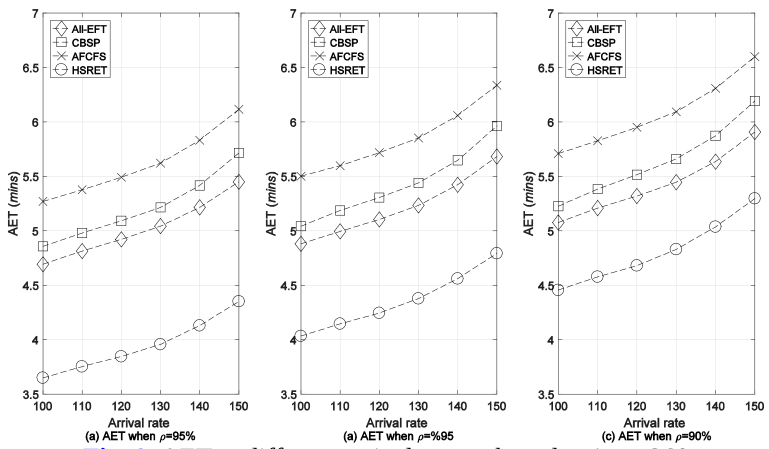


Fig. 9. AET at different arrival rates when slot time=360

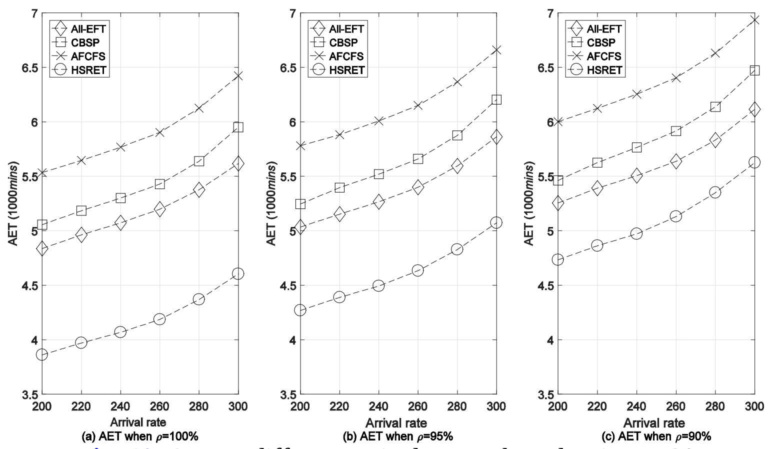


Fig. 10. AET at different arrival rates when slot time=720

Figs. 9 and **10** are the AET of different methods at different arrival rates and with different accuracies about the forecast of execution time under different parallelisms. **Fig. 9** is the AET when the slot time is 360 (minutes) and **Fig. 10** is the AET when the slot time is 720 (minutes).

Generally, no matter which slot time is, the AET of all four methods are increasing with the enhancement of arrival rates. The order of AET from big to small is: AFCFS, CBSP, ALL-EFT and HSRET. AFCFS has the largest value in AET because it always makes every task have the minimum resources to ensure it can be finished before its deadline. ALL-EFT always ensures the task to be executed in the shortest time, so it has a relatively lower value in AET. CBSP has three steps and it always assigns resources according to the system load. So, the AET of CBSP has a tradeoff between AFCFS and ALL-EFT. HSRET considers the system load, and then according to the system load and other requirements (deadline, ρ) to schedule resources. Thus it holds the lowest value in AET.

All methods have an increasing trend with the increase of arrival rate and the decrease of ρ. When the slot time is 360 *mins*, the average AETs of ALL-EFT, CBSP, AFCFS and HSRET are 5.2153E+03, 5.4145E+03, 5.8336E+03 and 4.3674E+03, respectively. To ALL-EFT, CBSP, AFCFS, HSRET average reduces by 16.26%, 19.34% and 25.13% in AET. When the slot time is 720 *mins*, the average AETs of ALL-EFT, CBSP, AFCFS and HSRET are 5.3767 E+03, 5.6382 E+03, 6.1357 E+03 and 4.6249 E+03, respectively. To ALL-EFT, CBSP, AFCFS, HSRET average reduces by 13.98%, 17.97% and 24.62%.

With the decrease of ρ, all methods have a little increase in AET. This is because it becomes difficult and not accurate to predict the system load for all methods. For example, HSRET under $\rho = 90\%$ increases by 9.45% and 18.02% to the AET when $\rho = 95\%$ and $\rho = 100\%$ when the slot time is 360.

The value of slot time also has an effect on AET. The AETs (of all methods) have a smaller value when the slot time is 720 (secs) compare to the AETs when the slot time is 360 (secs). This is because with the increasing of slot time, more resource fragments are consumed for the parallel tasks.

6.2.2 Comparison of AWT

Fig. 11 is the AWT of different methods at the arrival rate changed from 100 to 150 with a step of 10. **Fig. 12** is the AWT of different methods at the arrival rate changed from 200 to 300 with a step of 20. **Table 3** is the average AWT under different conditions.

The order of AWT from big to small is: ALL-EFT, CBSP, AFCFS, HSRET, no matter the slot time is 360 (**Fig. 11**) or 720 (**Fig. 12**). To AWT of ALL-EFT, CBSP and AFCFS, and AWT of HSRET 196.4, 124.0603 and 54.2381, about 34.91%, 25.3% and 19.02% when the slot time is 360 (**Fig. 11**); HSRET average reduces by 204.0325, 122.8922 and 46.6326, about 34.48%, 24.07% and 10.74% in AWT when the slot time is 720 (**Fig. 12)**.

ALL-EFT has the highest value in AWT, because ALL-EFT always waits for adequate resources to shorten the execution time or to ensure the deadline requirement. CBSF sometimes is required to wait for resources for the selected parallelism. We can find that AFCFS has a lower value in AWT expecting HSRET, because AFCFS always allocate resources once they come and with the smallest resources requirement. HSRET gets the initial scheduling list as AFCFS, and then according to the initial list, gets an optimistic scheduling result. So, HSRET always has the lowest value in AWT.

Compared to Section 6.2.1 about the AET, we find that there are some conflicts in AWT and AET, especially for AFCFS and ALL-EFT. AFCFS always allocates tasks when the task comes, so they have a lower value in AWT, but it always gives only the lowest resources to ensure meeting the deadline of the task, so, it has a higher value in AET. Contrary to AFCFS, ALL-EFT always makes the task execute in the shortest execution time under the system load requirement, so it has a higher value in AET and a lower value in AWT.

The accuracy of ρ plays an important role for AWTs of all methods. AWTs of all methods have a little increasing trend with the increasing of ρ. The reason is when we do not get a inaccurate of ρ, some tasks may not be executed as we wish, and that makes other tasks wait more time.

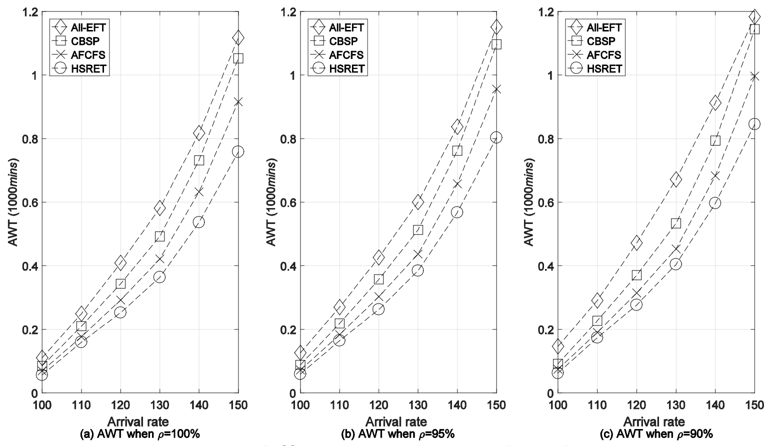


Fig. 11. AWT at different arrival rates when slot time=360

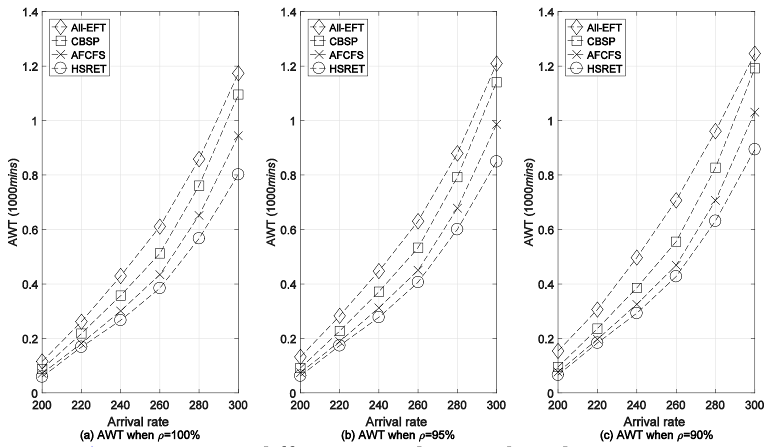


Fig. 12. AWT at different arrival rates when slot time=720

6.2.3 Comparison of PUT

Fig. 13 is the PUT of different methods when the arrival rate is changed from 100 to 150 with a step of 10. **Fig. 14** is the PUT of different methods when the arrival rate is changed from 200 to 300 with a step of 20.

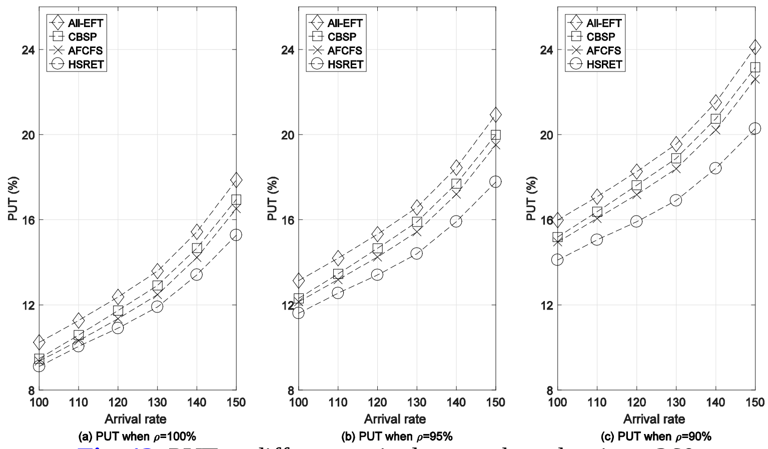


Fig. 13. PUT at different arrival rates when slot time=360

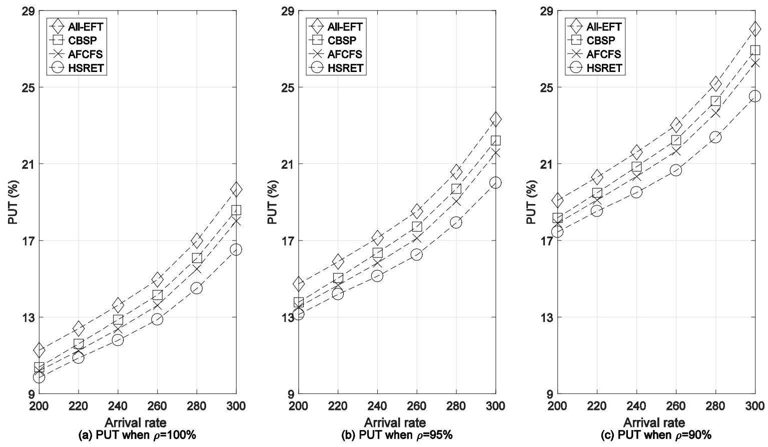


Fig. 14. PUT at different arrival rates when slot time=720

In general, all methods increase with the enhancement of arrival rates and the value of ρ. HSRET always has the lowest value in PUT, followed by AFCFS, CBSP and ALL-EFT, respectively. HSRET always not only considers the deadline, but also considers the system load, and based on the AFCFS policy, it optimizes the scheduling result to satisfy multiple targets, so it has the best performance in PUT. AFCFS always gives tasks the smallest resources to ensure they can be finished before deadlines, so it also has a relatively low value in PUT, but at the same time, it has the highest value in AET (Section 6.2.1). ALL-EFT always gives every task as many resources as possible under the system load, so it has relative low value in AET, and at the same time, because of the dynamic of the system load, it does not perform well in PUB. CBSP also has the same problem to HSRET, because there is a difficulty for them to schedule resources in the dynamic load environment.

For **Figs 13~14**, we can find the accuracy of ρ, also plays an important role in the scheduling result of PUT. PUTs of all methods with $\rho = 100\%$ increase by about 3% and 5% to the condition when $\rho = 95\%$ and $\rho = 90\%$. For example, PUT of HSRET under $\rho = 100\%$ enhances by 17.56% and 29.88% to the condition when $\rho = 95\%$ and $\rho = 90\%$ (when the slot time is 360). With the drop of ρ, all methods have difficulty to forecast whether the resources are enough to ensure those tasks can be finished before their deadlines. They always save more resources to satisfy some other tasks when they need more resources than our forecast.

6.3 Complexity analysis

Suppose the *I* is the number of tasks, the maximum parallelism of all tasks is *maxp*, the number of cluster is *cn*, the maximum number of resources in a cluster is *maxr*.

Algorithm 1 tries to get the normalized execution time. The complexity of lines 5~15 (Algorihm 1) is *maxp*, and the complexity of lines 6~9 is *I*. So the complexity of Algorithm 1 is:

$$\mathrm{O}(\text{Algorithm 1}) = \mathrm{O}(I^{maxp} + maxp + cn * maxr)$$

In Algorithm 2, for every task (*I* is the number of tasks), according to the different kinds of tasks, Sch1(), Sch2() and Sch3() are used to schedule those tasks to different clusters (the number of clusters is *cn*). So, the complexity of Algorithm 2 is:

$$\mathrm{O}(\text{Algorithm 2}) = \big(\mathrm{O}(\text{Sch1}) + \mathrm{O}(\text{Sch2}) + \mathrm{O}(\text{Sch3})\big) * \mathrm{O}(cn)$$

Algorithm 3 (Sch1) tries to search for scheduling method for the task t_i (the number of tasks is I) based on reschedule lft_i (We take the number of tasks in lft_i as a constant $con1$, the resource number is $maxr * cn$). So, the complexity of Algorithm 3 is:

$$\mathrm{O}(\text{Algorithm 3}) = \mathrm{O}(\text{Sch1}) = \mathrm{O}(I * con1) = \mathrm{O}(I)$$

Sch2 just schedules tasks (the number of tasks is *I*) to enhance the normalized execution time and reduces the system load with *maxp* steps, so the complexity of Sch2:

$$\mathrm{O}(\text{Sch2}) = \mathrm{O}(maxp * I)$$

Algorithm 4 (Sch3) tries to search for scheduling method for the task t_i (the number of tasks is I) based on reschedule rgt_i (We take the number of tasks in rgt_i as a constant $con2$). So, the complexity of Algorithm 4 is:

$$\mathrm{O}(\text{Algorithm 4}) = \mathrm{O}(\text{Sch3}) = \mathrm{O}(I * con2) = \mathrm{O}(I)$$

So, the complexity of our method is:

$$\begin{aligned} &\mathrm{O}(\text{Algorithm 1}) + \mathrm{O}(\text{Algorithm 2}) \\ &= \mathrm{O}(I^{maxp} + maxp + cn * maxr) + \big(\mathrm{O}(\text{Sch1}) + \mathrm{O}(\text{Sch2}) + \mathrm{O}(\text{Sch3})\big) * \mathrm{O}(cn) \\ &= \mathrm{O}(I^{maxp} + maxp + cn * maxr) + (\mathrm{O}(I) + \mathrm{O}(maxp * I) + \mathrm{O}(I)) * \mathrm{O}(cn) \\ &= \mathrm{O}(I^{maxp} + maxp + cn * maxr) + \mathrm{O}(I * cn * maxp) \end{aligned}$$

And most of time, we can get the normalized execution time for the system load (or the load forecast), so, we can take the complexity of Algorithm 1 as $\mathrm{O}(1)$, and then the complexity of our method becomes:

$$\mathrm{O}(I * cn * maxp)$$

For every task ($\mathrm{O}(I)$), AFCFS tries to search in every cluster ($\mathrm{O}(cn)$), to find a parallelism ($\mathrm{O}(maxp)$) to ensure finishing the task before its deadline, so the complexity of AFCFS is the same as our method: $\mathrm{O}(I * cn * maxp)$. ALL-EFT tris to select a parallelism ($\mathrm{O}(maxp)$) for every task ($\mathrm{O}(I)$) in every cluster ($\mathrm{O}(cn)$) to ensure that the task has the minimum execution time, and repeat the step until all tasks have been scheduled ($\mathrm{O}(I)$). So, the complexity of ALL-EFT is: $\mathrm{O}(I^2 * cn * maxp)$. The complexity of CBSP is decided by the three steps: firstly, determining of the computing clusters ($\mathrm{O}(cn)$); secondly, determining the optimal number of processors in each cluster($\mathrm{O}(cn)$); finally, placing the tasks on the appropriate processors ($\mathrm{O}(I * maxp)$). So, the complexity of CBSP is $\mathrm{O}(I * cn^2 * maxp)$. In summary, our method has the same complexity to AFCFS, and it is less than the complexity of ALL-EFT and CBSP.

7. Conclusion and future work

In this paper, we pay attention to the scheduling of moldable parallel tasks with non-linear speedup. We try to normalize those parallel tasks and make them have the same scope of execution time. Based on our analysis, we propose a scheduling method HSRET in the paper. First, HSRET tries to schedule tasks to make all tasks executed at the same reference execution time, and then according to the details of the deadline and other multiple scheduling targets, HSRET gets an initial scheduling result, and based on the initial result, some optimized methods are used to improve it. Simulation results show that our method has performed better in AET, AWT and PUT. Recently, for most of research work on the energy consumption of the parallel tasks, those researchers try to find some scheduling methods to save the energy consumption, and at the same time, to keep other metrics. As future work, we try to use the normalized execution time to work on the energy-aware [30] scheduling for moldable parallel tasks.

Acknowledgment

This work was supported in part by a grant from National Natural Science Foundation of China (No. 61672442),Fujian Province Science and Technology Plan Project (No. 2016Y0079),Xiamen Science and Technology Plan Project (No. 3502Z20183055), high-level talents program of Xiamen University of Technology (No. YKJ16020R)

Reference

[1] Thoman P, Dichev K, Heller T, et al., “A taxonomy of task-based parallel programming technologies for high-performance computing,” *Journal of Supercomputing*, 74(4), 1422-1434, 2018. Article (CrossRef Link).

[2] Feitelson D G, Rudolph L, “Toward convergence in job schedulers for parallel supercomputers,” *Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg*, 1-26, 1996. Article (CrossRef Link).

[3] Fan L, Zhang F, Wang G, et al., “An effective approximation algorithm for the Malleable Parallel Task Scheduling problem,” *Journal of Parallel & Distributed Computing*, 72(5), 693-704, 2012. Article (CrossRef Link).

[4] Memeti S, Pllana S, “PAPA: A Parallel Programming Assistant Powered by IBM Watson Cognitive Computing Technology,” *Journal of Computational Science*, 26, 275-284, 2018. Article (CrossRef Link).

[5] Hao Y, Wang L, Zheng M, “An adaptive algorithm for scheduling parallel jobs in meteorological Cloud,” *Knowledge-Based Systems*, 98(C), 226-240, 2016. Article (CrossRef Link).

[6] Wen Na, Liu Z, Li L, “Direct ENSO impact on East Asian summer precipitation in the developing summer,” *Climate Dynamics*, 52(11), 6799-6815, 2019. Article (CrossRef Link).

[7] Chen C Y, “An Improved Approximation for Scheduling Malleable Tasks with Precedence Constraints via Iterative Method,” *IEEE Transactions on Parallel & Distributed Systems*, 28(9), 1937-1946, 2018. Article (CrossRef Link).

[8] Wu X, Loiseau P, “Algorithms for Scheduling Deadline-Sensitive Malleable Tasks,” in *Proc. of Allerton Conference on Communication, Control, and Computing*, 530-537, 2015. Article (CrossRef Link).

[9] Shaoqi Wang,Wei Chen, Xiaobo Zhou, Liqiang Zhang,Yin Wang, “Dependency-aware Network Adaptive Scheduling of Data-Intensive Parallel Jobs,” *IEEE Transactions on Parallel & Distributed Systems*, 30(3), 515-529, 2019. Article (CrossRef Link).

[10] Verner, Uri, A. Mendelson, and A. Schuster, "Extending Amdahl's Law for Multicores with Turbo Boost," *IEEE Computer Architecture Letters*, 16(1), 30-33, 2017. Article (CrossRef Link).
[11] Wang Y R, Huang K C, Wang F J, "Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments," *Future Generation Computer Systems*, 60(C), 35-47, 2016. Article (CrossRef Link).
[12] Saifullah A, Agrawal K, Lu C, et al., "Multi-core Real-Time Scheduling for Generalized Parallel Task Models," in *Proc. of Real-Time Systems Symposium. IEEE*, 217-226, 2012. Article (CrossRef Link).
[13] Casanova H, Desprez F, Suter F, "Minimizing Stretch and Makespan of Multiple Parallel Task Graphs via Malleable Allocations," in *Proc. of International Conference on Parallel Processing. IEEE*, 71-80, 2010. Article (CrossRef Link).
[14] Xin Y, Xie Z Q, Yang J., "A load balance oriented cost efficient scheduling method for parallel tasks," *Academic Press Ltd.*, 81, 37-46, 2017. Article (CrossRef Link).
[15] Sanders P, Speck J, "Energy efficient frequency scaling and scheduling for malleable tasks," in *Proc. of International Conference on Parallel Processing. Springer-Verlag*, 167-178, 2012. Article (CrossRef Link).
[16] Sánchez D, Isern D, Ángel Rodríguez-Rozas, et al., "Agent-based platform to support the execution of parallel tasks," *Expert Systems with Applications*, 38(6), 6644-6656, 2011. Article (CrossRef Link).
[17] Evermann J, "Scalable Process Discovery Using Map-Reduce," *IEEE Transactions on Services Computing*, 9(3), 469-481, 2016. Article (CrossRef Link).
[18] Nagarajan V, Wolf J, Balmin A, et al., "Malleable scheduling for flows of jobs and applications to MapReduce," *Journal of Scheduling*, 22(4), 393-411, 2019. Article (CrossRef Link).
[19] Marchal L, Simon B, Sinnen O, et al., "Malleable Task-Graph Scheduling with a Practical Speed-Up Model," *IEEE Transactions on Parallel & Distributed Systems*, 29(6), 1357-1370, 2018. Article (CrossRef Link)
[20] Saifullah A, Ferry D, Li J, et al., "Parallel Real-Time Scheduling of DAGs," *IEEE Transactions on Parallel & Distributed Systems*, 25(12), 3242-3252, 2014. Article (CrossRef Link).
[21] Li K, "Non-clairvoyant scheduling of independent parallel tasks on single and multiple multicore processors," *Journal of Parallel & Distributed Computing*, 2018. Article (CrossRef Link).
[22] Pathan R M, Voudouris P, Stenstrom P, "Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms," *IEEE Transactions on Parallel & Distributed Systems*, 29(4), 915-928, 2018. Article (CrossRef Link).
[23] Wang Q, Hou R, Hao Y, et al., "A parallel tasks Scheduling heuristic in the Cloud with multiple attributes," *Ksii Transactions on Internet & Information Systems*, 12(1), 287-307, 2018. Article (CrossRef Link).
[24] Chwa H S, Lee J, Lee J, et al., "Global EDF Schedulability Analysis for Parallel Tasks on Multi-Core Platforms," *IEEE Transactions on Parallel & Distributed Systems*, 28(5), 1331-1345, 2017. Article (CrossRef Link).
[25] Hao Y, Xia M, Wen N, et al., "Parallel task scheduling under multi-Clouds," *Ksii Transactions on Internet & Information Systems*, 11(1), 39-60, 2017. Article (CrossRef Link).
[26] M. Beji, S. Achour, "Resizing of Heterogeneous Platforms and the Optimization of Parallel Applications," in *Proc. of 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, United Kingdom*, 154-161, 2018. Article (CrossRef Link).
[27] Kayaaslan E, Lambert T, Marchal L, et al., "Scheduling series-parallel task graphs to minimize peak memory," *Theoretical Computer Science*, 707, 1-23, 2018. Article (CrossRef Link).
[28] Mahmood B, Ahmad N, Malik S U R, et al., "Power-efficient Scheduling of Parallel Real-time Tasks on Performance Asymmetric Multicore Processors," *Sustainable Computing Informatics & Systems*, 17, 81-95, 2018. Article (CrossRef Link).
[29] Sheikh H F, Ahmad I, Fan D, "An Evolutionary Technique for Performance-Energy-Temperature Optimized Scheduling of Parallel Tasks on Multi-Core Processors," *IEEE Transactions on Parallel & Distributed Systems*, 27(3), 668-681, 2016. Article (CrossRef Link).

[30] Shojafar M, Cordeschi N, Baccarelli E, “Energy-efficient Adaptive Resource Management for Real-time Vehicular Cloud Services,” *IEEE Transactions on Cloud Computing*, 7(1), 196-209, 2019. Article (CrossRef Link).

Jianmin Li received the M.S. degree in Computer Science Department from Xiamen University in 2009 and the Ph.D degree in Department of Automation of Xiamen University in 2015. He is currently a faculty of School of Computer and Information Engineering from Xiamen University of Technology. His research interests include computer vision, machine learning and pattern recognition.