Journal of Computers Vol. 28, No. 1, 2017, pp. 43-56
doi:10.3966/199115592017022801004

Task Clustering Heuristics for Efficient Execution Time Reduction in Workflow Scheduling

Kuo-Chan Huang[1], Di-Syuan Gu[1], Hsiao-Ching Liu[1], and Hsi-Ya Chang[2]

[1] Department of Computer Science, National Taichung University of Education
Taichung 403, Taiwan, ROC
kchuang@mail.ntcu.edu.tw, ejeu67195@gmail.com, unlimited_19800830@yahoo.com.tw

[2] National Center for High-Performance Computing
Hsinchu 300, Taiwan, ROC
9203117@nchc.narl.org.tw

Received 10 June 2015; Revised 24 August 2015; Accepted 14 September 2015

Abstract. Nowadays, many large-scale scientific and engineering applications are usually constructed as dependent task graphs, or called workflows, for describing complex interrelated computation and communication among constituent software modules or programs. Therefore, scheduling workflows efficiently becomes an important issue in modern parallel computing environments, such as cluster, grid, and cloud. Task clustering is one of the major categories of task graph scheduling approaches, aiming at reducing inter-task communication costs. In this paper, we propose three new task clustering approaches, Critical Path Clustering Heuristic (CPCH), Larger Edge First Heuristic (LEFH), and Critical Child First Heuristic (CCFH), which are expected to achieve better task graph execution performance by trying to minimize the communication costs along execution paths. The proposed schemes were evaluated with a series of simulation experiments and compared to a typical clustering based task graph scheduling approach in the literature. The experimental results indicate that the proposed CPCH, LEFH, and CCFH heuristics outperform the typical scheme significantly, up to 21% performance improvement in terms of average makespan for workflows of large Communication-to-Computation Ratio (CCR).

Keywords: execution time reduction, task clustering, workflow scheduling

1 Introduction

As modern high-performance computing applications are becoming even more complex and computation-demanding, many large-scale scientific and engineering applications are now usually constructed as dependent task graphs, or called workflows, due to large amounts of interrelated computation and communication, where different tasks within a task graph might execute distinct programs and have data or execution dependency among them [1-2]. Most workflow applications can be represented by Directed Acyclic Graphs (DAG) for describing inter-task dependencies [3]. Fig. 1 is a workflow example of DAG structure. Each node represents a task which executes a specific program. The number next to each node means the required execution time of the task. Each edge represents the inter-task dependency. The number next to an edge means the inter-task data transmission cost. A workflow scheduler has to schedule and allocate each task according to the dependencies specified in the workflow definition.

However, in practice, a workflow application is usually not developed by drawing an arbitrary DAG, but instead developed with some kind of tools based on a specific programing model, e.g. YAWL [6] and BPEL [7]. The underlying programming model usually has some characteristics that would restrict the set of DAGs that can be developed with it. Therefore, most workflow applications have more regular structures than the general DAG in Fig. 1. As discussed in [4-5], fork-join is a common and major construct widely appearing in the restricted DAG structures of many real workflow applications developed based on popu

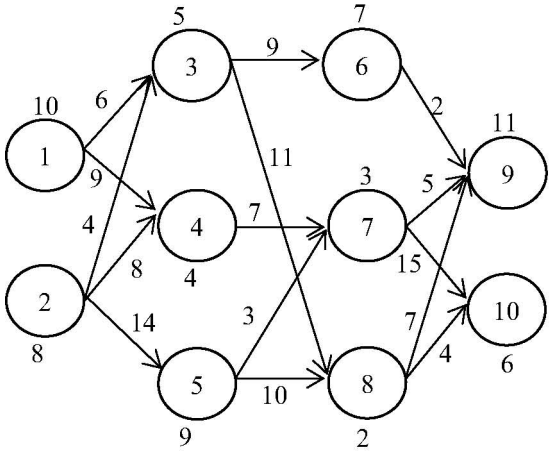

Fig. 1. A workflow example of DAG structure

lar workflow programming models, e.g. YAWL [6] and BPEL [7]. Fig. 2 is an example of such fork-join DAG-based workflows consisting of a series of *sequence* patterns and *fork-join* structures. A sequence pattern is the kind of paths in a DAG where each node has exactly one child except the last node. The set of tasks in a sequence pattern have to be executed sequentially. For example, tasks {1, 2}, tasks {5, 6}, and tasks {10, 11} are typical sequence patterns in Fig. 2. A fork-join structure in a DAG is usually used to represent potential execution parallelism. It starts with a *fork* node and ends with a *join* node, containing several sequence patterns between these two nodes. A fork node has more than one child and a join node has more than one parent. In a fork-join structure, the number of children of the fork node is equivalent to the amount of parents of the join node. The set of sequence patterns in a fork-join structure can be executed in parallel if there are enough computational resources. In Fig. 2, there are two fork-join structures: tasks {2, 3, 4, 5} and tasks {6, 7, 8, 9, 10}. Nodes 2 and 6 are fork nodes and nodes 5 and 10 are join nodes. Tasks 3 and 4 represent potential execution parallelism in the first fork-join structure. Fork and join nodes usually are the intersections of sequence patterns and fork-join structures, and participate in both a sequence pattern and a fork-join structure. For example, node 2, a fork node, is the intersection of the sequence pattern {1, 2} and the fork-join structure {2, 3, 4, 5}, while node 10, a join node, is the intersection of the fork-join structure {6, 7, 8, 9, 10} and the sequence pattern {10, 11}. Since fork-join DAG captures the underlying structure of most real workflow applications [4][5], in this paper we focus on the scheduling issues of fork-join DAG-based workflows.

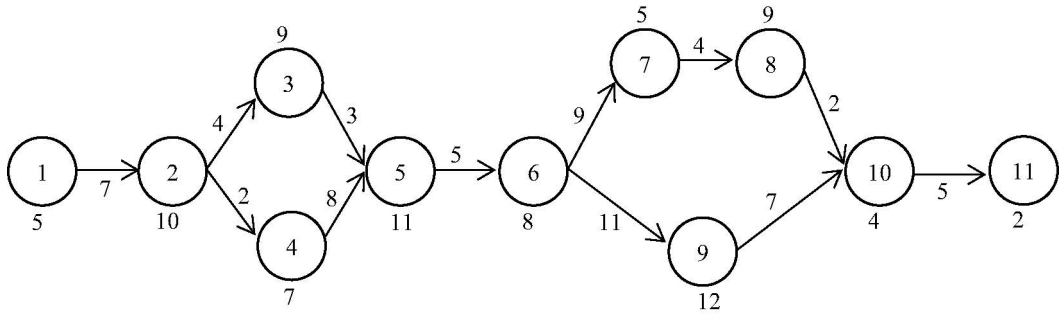

Fig.2. An example of fork-join DAG-based workflows

The time period from the submission of a workflow to its completion is usually called *makespan*, which is a common criterion used to compare the performance of different workflow scheduling algorithms. Apparently, a shorter makespan indicates a better execution performance. In this paper, we use the average makespan of a set of different workflows to measure and compare the performance of various scheduling algorithms under study.

Task graph scheduling in parallel and distributed environments, in general, is a NP-complete problem [8-9]. Therefore, many heuristic methods have been proposed [3, 10-14]. Heuristic-based task graph scheduling algorithms usually can be classified into three types: (1) list-based, (2) clustering-based, and

(3) duplication-based. Minimizing inter-task communication costs plays an important role in producing better workflow execution schedules. Clustering-based heuristics were shown to be more effective than list-based heuristics for minimizing inter-task communication costs and thus reducing workflow execution makespan in many cases, especially for workflows of fork-join DAG structure [4, 19]. Although duplication-based heuristics also have the advantage of reducing inter-task communication costs, they are less effective than others when considering modern shared parallel computing platforms, e.g. cluster, grid, and cloud. This is because task duplication consumes extra computational resources. Reduction of inter-task communication is achieved at the cost of increased computation workload induced by task duplication. For single workflow scheduling, the duplicated tasks usually utilize the resources which would be idle otherwise, and therefore would not hurt the start time of other tasks. However, for modern shared parallel computing platforms, e.g. cluster, grid, and cloud, where it's common that several workflows are running simultaneously, the duplicated tasks of one workflow would compete with tasks of other workflows for resources, and thus delay the finish time of workflows. Since clustering-based approaches are superior to list-based and duplication-based methods in most cases, in this paper we focus on the study of clustering-based scheduling approaches for fork-join DAG-based workflows.

Among various clustering-based task graph scheduling methods [4, 15-19], Path Clustering Heuristic (PCH) is a typical one and was developed for dealing with fork-join based task graphs specifically, which has been shown effective in [4, 19]. PCH partitions a task graph of fork-join structure, e.g. Fig. 2, into several task groups first, and then allocates these task groups onto processors. PCH builds a task group by performing a depth-first search on the task graph until reaching a task t_s which has an unscheduled predecessor. The task t_s is not included in the current task group. Therefore, a join node with several predecessors will always be clustered with the last predecessor reaching it. However, such kind of clustering doesn't always minimize critical communication costs for workflow execution. In this paper, we propose three new task clustering approaches, Critical Path Clustering Heuristic (CPCH), Larger Edge First Heuristic (LEFH), and Critical Child First Heuristic (CCFH), which are expected to achieve better workflow execution performance than existing methods by trying to minimize the communication costs along execution paths. The proposed task clustering approaches were evaluated through a series of simulation experiments. Experimental results indicate that the proposed approaches outperform the previous PCH scheme significantly, up to 21% performance improvement in terms of average makespan for workflows of large Communication-to-Computation Ratio (CCR).

The remainder of this paper is organized as follows. Section 2 discusses related works on task graph scheduling, especially the clustering heuristics. Section 3 describes and discusses our CPCH, LEFH, and CCFH approaches. Section 4 evaluates and compares our three approaches with PCH through a series of simulation experiments. Section 5 concludes this paper.

2 Related Work

Most heuristic-based workflow scheduling algorithms usually can be classified into three types: (1) list-based, (2) clustering-based, and (3) duplication-based. List-based heuristic approaches [20-23] first assign a priority to each task within a workflow and then maintain a list of all unscheduled tasks according to the decreasing order of their priority values. The scheduler repeatedly schedules the tasks onto processors based on the list. How to minimize communication cost is an important issue for many kinds of parallel job scheduling systems [24]. For workflows, the main idea of clustering-based scheduling algorithms [15-18, 25] is to reduce communication delay by clustering the tasks of heavy communication into the same group and then scheduling an entire task group onto the same processor. Duplication-based heuristic methods [26-29] duplicate a task on the set of processors where its successors run in order to minimize inter-task communication costs and thus improve the overall makespan of an entire workflow.

Clustering-based heuristics have been shown to be more effective than other heuristics in many cases, especially for workflows of fork-join DAG structure [4, 19]. Therefore, in this paper we focus on the study of clustering-based approaches. Several task clustering approaches for workflow scheduling have been proposed in the literature [25], including linear clustering [30-31], single edge clustering [31-32], list scheduling as clustering [17-18], and Path Clustering Heuristic (PCH) [4, 19]. The linear clustering algorithm [30-31] repeats to find the critical path among the tasks not yet been clustered into any group. In linear clustering only dependent tasks would be clustered into the same group [16, 25]. The main part of the single edge clustering algorithm [31-32] sorts edges in decreasing order of their weights. The

weights then become the priority of grouping. For each edge, the clustering algorithm checks if zeroing the edge would lead to a new schedule shorter or equal to the original schedule. If yes, the two clusters connected by this edge are merged into a single cluster. This clustering approach has high computational complexity. The list scheduling as clustering algorithm [17-18] is actually an adapted list scheduling approach. In each step of the algorithm, it deals with a node and considers all the edges pointing to that node for zeroing.

Path Clustering Heuristic (PCH) was proposed for scheduling fork-join DAG-based workflows and has been shown to be effective in [4, 19]. PCH adopts a two-step framework for workflow scheduling. At the first step a clustering scheme is used to partition a workflow into several task groups. Then, in the second step a heuristic is applied to allocate these task groups onto processors. In this paper, we adopt the same scheduling framework as PCH and develop several new task clustering schemes to further improve workflow execution performance.

3 Task Clustering Approaches

In this paper, we focus on the workflows containing fork-join structures, which can be represented by a Directed Acyclic Graph (DAG), G (V, E, w, c), where:

- V is the set of tasks, $t_n \in V$, |V| = number of tasks;
- E is the set of directed edges, $e_n \in E$, |E| = number of edges;
- w is computation cost of a task, e.g. w_i is the computation cost of task i;
- c is communication cost of an edge, e.g. $c_{i,j}$ is the communication cost between tasks i and j.

Each workflow starts at one node, named the entry node, and finishes at one node, named the exit node. Each node in the workflow is a task representing a specific job or program to execute and each edge represents the data dependence between two tasks. Each task starts its execution only after receiving all data from its parent tasks.

The following defines several task attributes which will be used in describing the workflow scheduling algorithms:

- Priority:

$$P_i = \begin{cases} w_i & \text{,if task i is the exit node of the workflow} \\ w_i + \max\limits_{t_j \in succ(n_i)} (c_{i,j} + P_j) & \text{, otherwise} \end{cases}$$

succ(n_i) : the set of immediate successors of task i.

- Earliest start time:

$$EST_i = \begin{cases} 0 & \text{, if } t_i \text{ is the entry node of the workflow} \\ \max_{th \in pred(ti)} (ETS_h + w_h + c_{h,i}) & \text{, otherwise} \end{cases}$$

representing the earliest time that task i is ready for execution

- Earliest finish time:

$$EFT_i = EST_i + w_i$$

representing the earliest time that task i can finish its execution.

Workflow scheduling can be viewed as the spatial and temporal assignment of the constituent tasks to processors. The spatial assignment is the allocation of tasks to the processors. The temporal assignment is the attribution of a start time to each task. It is usually assumed that data transfer between two tasks assigned to the same processor incurs no communication costs when discussing workflow scheduling [25]. A schedule is feasible if and only if all nodes n and edges e in a workflow comply with the following conditions 1 and 2 [25].

- Condition 1: exclusive processor allocation
  For any two nodes, n_i and n_j, allocated to the same processor, either $t_s(n_i) < t_f(n_i) <= t_s(n_j) < t_f(n_j)$ or $t_s(n_j) < t_f(n_j) <= t_s(n_i) < t_f(n_i)$, where t_s is the start time of a task and t_f is its finish time.
- Condition 2: precedence constraint

For any two nodes, n_i and n_j where n_i is the parent of n_i, $t_s(n_j) >= t_f(n_i)$ if n_i and n_j are allocated to the same processor, and $t_s(n_j) >= t_f(n_i) + c_{i,j}$ if n_i and n_j are allocated to different processors.

The required execution time of a workflow on a parallel computing platform is defined to be its *makespan*. In this paper, we deal with the workflow scheduling problem which is to determine a feasible schedule for a specific workflow on a particular parallel computing platform with the aim of achieving an as-short-as-possible makespan.

Typical clustering-based workflow scheduling approaches, e.g. PCH [4, 5], usually adopt an iterative two-step framework as shown in Algorithm 1. The first step within the iterative procedure, i.e. line 3, adopts a particular task clustering scheme to find a group of unscheduled tasks, which varies in different clustering-based methods. Then, in the second step, i.e. line 4, a heuristic is applied to allocate the task group onto a processor based on some performance criterion, which is usually the Earliest Finish Time (EFT) of the task group in most methods. In this paper, we develop three new task clustering heuristics for clustering-based workflow scheduling approaches to improve workflow execution performance, i.e. makespan.

Algorithm 1: Clustering-based workflow scheduling
Input: a DAG-based workflow to be scheduled on a parallel platform for execution
Output: an schedule arranging the workflow's execution on the parallel platform

```
1. Compute all tasks' attributes
2. while there are unscheduled nodes do
3.     group ⇐ get_next_task_group()
4.     allocate(group)
5. end while
6. return the workflow's execution schedule
```

Algorithm 2 in the following presents the scheme used in PCH to cluster a set of tasks into a task group. PCH repeatedly applies the scheme to build task groups of a workflow until no unclustered task is left. The clustering process starts with n, the unclustered node with highest priority, as described at line 2. Line 3 includes the node n as the first node of the new cluster. It then performs a depth-first search starting from node n until reaching a task which has an unclustered predecessor, as described through line 4 to line 12. Line 13 returns the task group built.

Algorithm 2 Get_next_task_group for PCH
Input: the set of tasks in a workflow, which are not included in any task group yet
Output: a new task group produced by the task clustering heuristic

```
1.  group ←ϕ
2.  n ← unscheduled node with highest Priority
3.  group ← group ∪ n
4.  while (n has unscheduled successors) do
5.      n_succ ← successor_i of n with highest P_i + EST_i
6.    if (n_succ has an unscheduled predecessor)
7.      break;
8.    else
9.      group ← group ∪ n_succ
10.     n ← n_succ
11.   end if
12. end while
13. return group
```

Fig. 3 is an example where PCH builds 4 task groups: {1, 2, 3, 6}, {4, 7}, {5, 9}, and {8, 10, 11} first. Then, the computation cost of each task group can be computed based on the computation cost of each task within the group, and the communication costs between different task groups can be calculated according to the communication cost on corresponding edges. After that, PCH allocates the task groups onto processors. The lower part of Fig. 3 shows an example where PCH allocates the task groups onto 3

processors. The makespan for this workflow execution is 180.

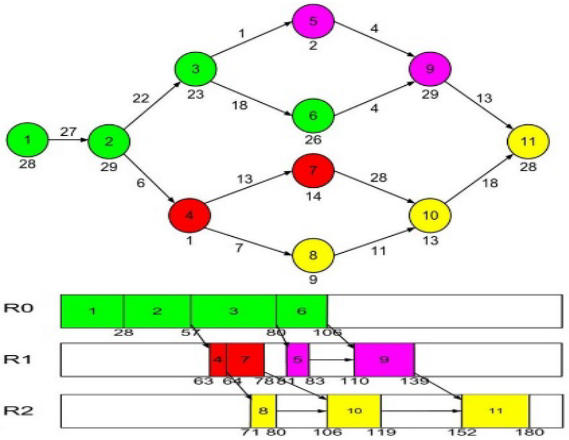

Fig. 3. Example of task clustering and scheduling in PCH

Looking at the above example carefully, we can find that PCH does not always produce the best task clustering results. For example, in Fig. 3, it might be better if node 6 and node 9 are put into the same group for communication cost minimization since node 6 has a larger computation cost than node 5 and is therefore on the critical path. Zeroing the communication cost between node 6 and node 9 can effectively shorten the length of critical path and hence reduce the makespan of the entire workflow. The above observation motivated our work in this paper to develop new task clustering schemes to further improve workflow execution performance. The following subsections presents three new task clustering heuristics which differ in how they handle fork nodes or join nodes.

3.1 A Critical Path Clustering Heuristic

In PCH [4][5], a join node with several predecessors will always be clustered with the predecessor which is the last one to reach it. However, such a clustering approach doesn't always minimize the communication costs during workflow execution and might lead to a degraded performance. In the following, we propose a Critical Path Clustering Heuristic (CPCH) for task clustering, which tries to minimize inter-task communication costs by clustering a join node with the predecessor on the longest path going through it.

Fig. 4 is an example illustrating how CPCH works. Taking node 9 for example, it would be clustered with node 5, as shown in Fig. 3, when using the PCH algorithm [4]. On the other hand, since the longest path going through node 9 is {1, 2, 3, 6, 9}, our CPCH chooses to cluster node 9 with node 6. Finally, CPCH partitions the workflow into 4 task groups: {1, 2, 3, 6, 9, 11}, {4, 7}, {5}, {8, 9} and leads to a makespan of 165, shorter than the schedule produced by PCH in Fig. 3.

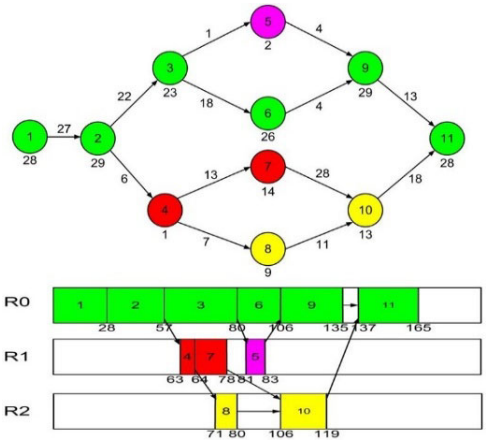

Fig. 4. An example of CPCH

Algorithm 3 describes the CPCH scheme in details. The clustering process starts with n, the unscheduled node with highest priority, as described at line 1. For each successor of n, lines 4 to 10 check whether n is on the longest path from the entry node to it. For those successors whose longest paths do not go through n, their priority values, L_i, are set to -1 for excluding them from current task group. If no successors' longest path goes through n, the task clustering process stops, as described at lines 11 to 12, and the task group built so far is returned at line 19. Otherwise, the successor on the longest path, with the largest L_i, will be included in current task group and the task clustering process repeats to explore the successors of the new node, as described at lines 13 to 17.

Algorithm 3 Get_next_task_group_for CPCH
Input: the set of tasks in a workflow, which are not included in any task group yet
Output: a new task group produced by the task clustering heuristic

```
1.  n ← the unscheduled node with highest priority
2.  group ← n
3.  while (n has unscheduled successors) do
4.      for each successor i of n do
5.          if  n is on the longest path from the entry node to successor i then
6.              L_i = EST_i + P_i
7.          else
8.              L_i = -1
9.          end if
10.     end for
11.     if there exists no L_i > 0 then
12.         break
13.     else
14.         n_succ ← successor i of n with the largest L_i
15.     end if
16.     group ← group ∪ n_succ
17.     n ← n_succ
18. end while
19. return group
```

3.2 Larger Edge First Heuristic

The above CPCH approach focuses on how to cluster join nodes and adopts the same clustering scheme as PCH to deal with fork nodes, where the child with the largest $EST_i + P_i$ will be included into the task group. Sometimes the child with the largest communication cost to the fork node will not be included into the task group because it has a smaller P_i. In this section we present a new task clustering approach, called Larger Edge First Heuristic (LEFH), which adopts a greedy method to deal with the fork node, always clustering a fork node and the child with the largest communication cost to it into the same task group.

Fig. 5 is an example for illustrating the potential advantage of LEFH. Fig. 5 (a) shows the task groups and the resultant schedule produced by CPCH and Fig. 5 (b) is for LEFH. Taking node 3 for example, it would be clustered with node 5 when using CPCH. On the other hand, since node 3 has a larger communication cost to node 6 than to node 5, LEFH chooses to cluster node 3 with node 6. Finally, LEFH generates 4 task groups: {1, 2, 4, 8}, {3, 6}, {5, 9}, {7, 10, 11} and leads to a makespan of 125, shorter than the schedule produced by CPCH in Fig. 5 (a).

Algorithm 4 in the following describes the LEFH scheme in details. The difference between LEFH and CPCH lies in line 6 where L_i is set to the communication cost between node n and its successor i. This arrangement allows a fork node to be clustered with the successor with the largest communication cost to it, while retaining the advantage of CPCH that clusters a join node with the parent on the longest path going through it.

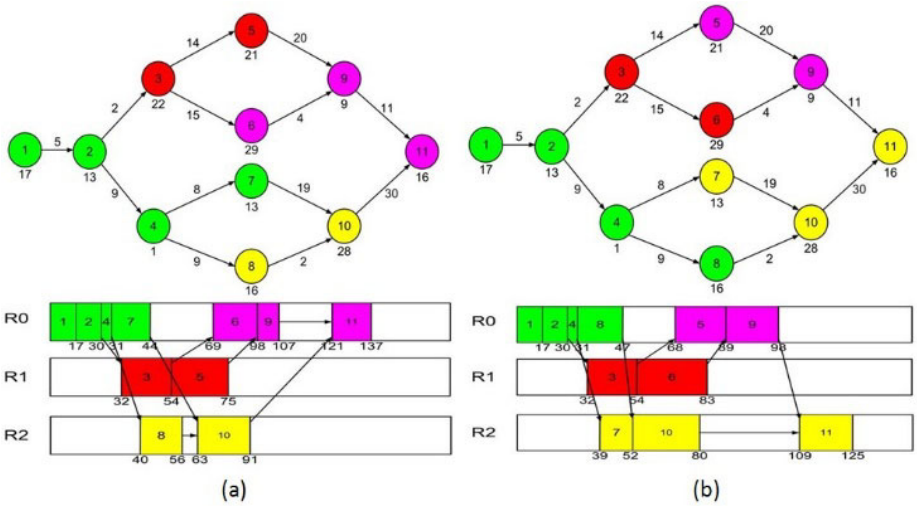

Fig. 5. An example illustrating the advantage of LEFH; (a) CPCH; (b) LEFH

Algorithm 4 Get_next_task_group for LEFH
Input: the set of tasks in a workflow, which are not included in any task group yet
Output: a new task group produced by the task clustering heuristic

```
1.  n ← the unscheduled node with highest priority
2.  group ← n
3.  while (n has unscheduled successors) do
4.      for each successor i of n do
5.          if  n is on the longest path from the entry node to successor i then
6.              L_i = c_{n,i} //the communication cost between node n and its successor i
7.          else
8.              L_i = -1
9.          end if
10.     end for
11.     if there exists no L_i > 0 then
12.         break
13.     else
14.         n_succ ← successor i of n with the largest L_i
15.     end if
16.     group ← group ∪ n_succ
17.     n ← n_succ
18. end while
19. return group
```

3.3 Critical Child First Heuristic

This section presents another new task clustering approach, called Critical Child First Heuristic (CCFH), which tries to make a compromise between PCH and LEFH when dealing with the fork node. CCFH takes not only the communication cost but also the computation cost into consideration when evaluating each child of a fork node. It clusters a fork node with the child of the largest total cost of computation and communication to it. The algorithm of CCFH is similar to Algorithm 3 while line 6 is changed to $L_i = EST_i + w_i$.

Fig. 6 is an example comparing LEFH and CCFH for illustrating the potential benefits of CCFH. Taking node 3 for example, it is clustered with node 6 in LEFH because node 3 has a larger communication cost to node 6 than to node 5. On the other hand, node 3 is clustered with node 5 when using CCFH since node 5 has a larger total cost of computation and communication than node 6. Finally, CCFH generates 4 task groups, {1, 2, 4, 7}, {3, 5, 9, 11}, {6}, {8, 10}, and leads to a makespan of 147, shorter than the schedule produced by LEFH.

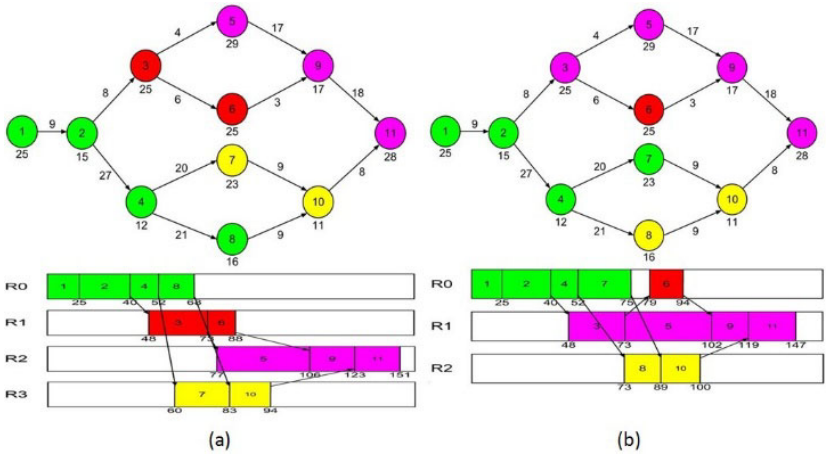

Fig. 6. An example illustrating the advantage of CCFH; (a) LEFH; (b) CCFH

4 Performance Evaluation

This section presents a series of simulation experiments which evaluate the proposed task clustering approaches in terms of average makespan. The makespan measures the time period between the start and completion of a workflow. We compare the proposed CPCH, LEFH, and CCFH with the PCH scheme [4].

We implemented a DAG generator to produce synthetic workflows of fork-join structures for the following simulation experiments. The generated workflows can be broadly classified into two types: single-level fork-join, e.g. Fig. 6, and double-level fork-join, e.g. Fig. 2. Each workflow contains one entry node and one exit node, and at each level there might be three to six fork-join structures randomly. The DAG generator can generate workflows with different CCR values [25]: 0.1, 1, and 10. It assigns a random weight to each node and edge according to the specified CCR value. The computing resources for executing the workflows are assumed to be a speed-homogeneous parallel system. Each experiment was conducted with 100 different synthetic workflows and the average makespan was calculated.

Fig. 7 and Fig. 8 show the experimental results of the single-level and double-level fork-join workflows, respectively, under different CCR values. The performance shown in the figures is the improvement ratio compared to PCH. It is clear that all the three proposed approaches achieve better performance, in term of average makespan, than PCH since they adopt an improved task clustering approach to dealing with the join nodes. The performance improvement ratio rises as CCR increases, up to 21% performance improvement. The three proposed approaches, CPCH, LEFH, and CCFH, differ mainly in how they handle the fork nodes. CPCH outperforms LEFH and CCFH when CCR is small, e.g. 0.1 in the figures. This is because the advantage of LEFH and CCFH based on zeroing the larger communication cost between a fork node and its successors becomes less significant when CCR is low, only small communication costs being saved. This is also revealed in that LEFH performs even worse than CCFH for low CCR values since LEFH only considers the communication costs while CCFH also takes successors' computation costs into account. On the other hand, LEFH and CCFH achieve better performance than CPCH for medium and large CCR values, e.g. 1 and 10 in the figures. Moreover, LEFH performs the best for large CCR value, e.g. 10 in the figures, since it always zeroes the largest communication cost between a fork node and its successors and this advantage is amplified with large CCR values.

In the following, we evaluate the proposed approaches with workflows of different branch lengths in the fork-join structure, where the CCR value is 1. Fig. 9 and Fig. 10 show the performance results of single-level and double-level fork-join workflows, respectively. The experimental results show that the proposed CPCH, LEFH and CCFH approaches outperform PCH consistently through different branch lengths.

In the following, we evaluate the proposed approaches with the workflow structure of a real-world application, LIGO [33]. The CCR value in this experiment is low. The workflow structure of LIGO is shown in Fig. 11. The experimental results in Fig. 12 show that the proposed CPCH, LEFH, and CCFH approaches outperform PCH, achieving more than 2% performance improvement in terms of average makespan, similar to the cases of CCR=0.1 in Fig. 8 and Fig. 9.

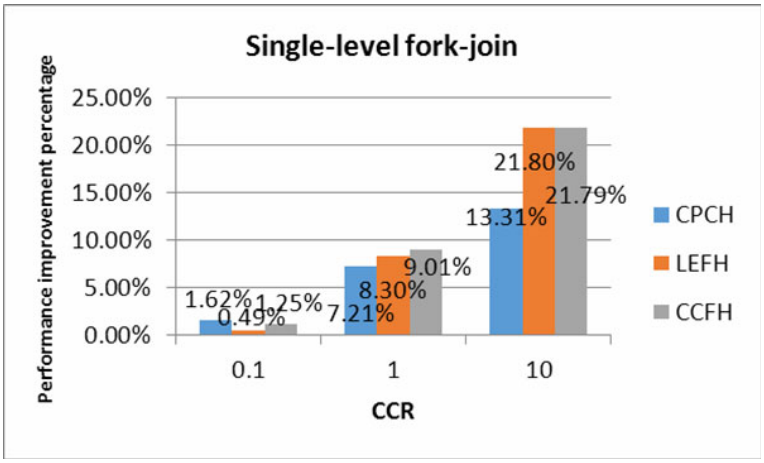

Fig. 7. Performance results of single-level fork-join workflows

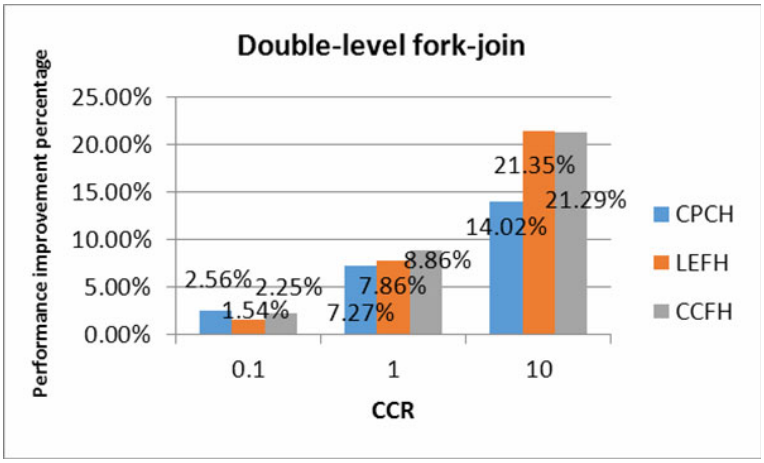

Fig. 8. Performance results of double-level fork-join workflows

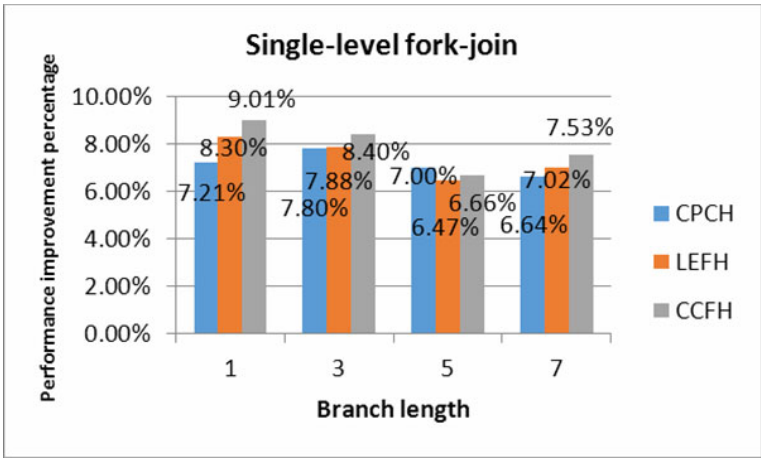

Fig. 9. Performance with different branch lengths for single-level fork-join structure

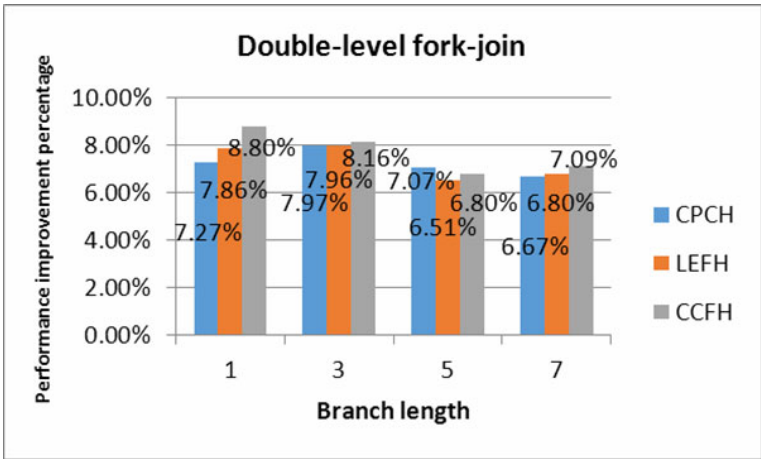

Fig. 10. Performance with different branch lengths for double-level fork-join structure

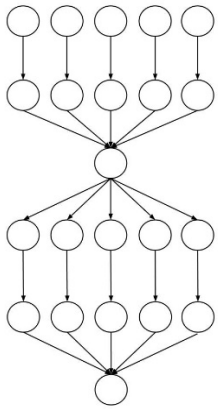

Fig. 11. Workflow structure of LIGO

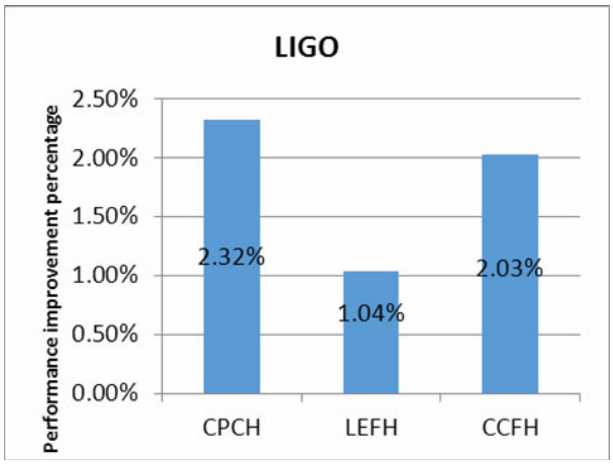

Fig. 12. Performance results for LIGO

In summary, the above experimental results, based on both synthetic workflows and a real workflow application, show that all our three proposed approaches outperform PCH [4] in a variety of different scenarios, including different levels of fork-join structure, CCR, and branch lengths. The relative strength of the three proposed approaches varies in different scenarios. In general, the performance improvement ratio grows as CCR increases. For smaller CCR, CPCH performs the best, while LEFH outperforms others for large CCR value, e.g. 10 in the figures, since it zeroes the largest communication cost between a fork node and its successors.

5 Conclusions

Nowadays, more and more large-scale scientific and engineering applications are usually constructed as workflows due to complex and large amounts of interrelated computation and communication. Therefore, scheduling workflow efficiently becomes an important issue in modern parallel computing environments. Clustering-based approaches are one of the major types of workflow scheduling methods, aiming at reducing inter-task communication costs. In this paper, we propose three new task clustering approaches, Critical Path Clustering Heuristic (CPCH), Larger Edge First Heuristic (LEFH), and Critical Child First Heuristic (CCFH), to scheduling workflows of fork-join structure, which have been shown to be the common structure of many workflow applications [4-5], in parallel systems. The three task clustering heuristics differ in how they handle fork nodes or join nodes. CPCH minimizes the inter-task communication cost by clustering a join node with the ancestor on the longest path going through it. LEFH adopts a greedy heuristic to cluster a fork node with the child having the largest communication cost from it. CCFH makes a compromise between PCH [4] and LEFH by clustering a fork node with the child of the largest cost of both communication and computation.
The proposed task clustering approaches have been evaluated with a series of simulation experiments and compared to PCH [4]. The experimental results indicate that all the three proposed approaches outperform PCH consistently through different CCR values. The performance improvement becomes even larger as CCR increases, up to 21% performance improvement when CCR is 10. More specifically, CPCH performs the best for small CCR, e.g. 0.1, CCFH achieves the best performance with medium CCR, e.g. 1, and LEFH outperforms the others when CCR is high, e.g. 10. This indicates that task clustering in workflow scheduling is a complex issue and users should carefully select an appropriate method according to the properties of workflow applications.

Acknowledgements

This paper is partially based upon work supported by National Science Council (NSC), Taiwan, under grants no. NSC101-2221-E-142-002-MY2.

References

[1] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. Silva, M. Livny, K. Wenger, Pegasus: a workflow management system for science automation, Future Generation Computer Systems 46(2015) 17-35.

[2] J. J. Durillo, R. Prodan, Multi-Objective Workflow Scheduling in Amazon EC2, Cluster Computing 17(2)(2014) 169-189.

[3] M. Wieczorek, R. Prodan, A. Hoheisel, M. Wieczorek, R. Prodan, A. Hoheisel, Taxonomies of the multi-criteria grid workflow scheduling problem, Grid Middleware and Services, 2008.

[4] L.F. Bittencourt, E.R.M. Madeira, A performance-oriented adaptive scheduler for dependent tasks on grids, Concurrency and Computation: Practice and Experience 20(2008) 1029-1049.

[5] L.F. Bittencourt, E.R.M. Madeira, Fulfilling task dependence gaps for workflow scheduling on grids, in: Proc. of the 3rd IEEE International Conference on Signal-Image Technology and Internet Based Systems (SITIS), 2007.

[6] YAWL (Yet Another Workflow Language). <http://en.wikipedia.org/wiki/YAWL>, 2015 (accessed 15.06).

[7] BPEL (Business Process Execution Language). <http://en.wikipedia.org/wiki/Business_Process_Execution_Language>, 2015 (accessed 15.06).

[8] R.M. Gary, S.D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, New York, 1979.

[9] D.J. Ullman, NP-complete scheduling problems, Journal of Computer and Systems Sciences 10(1975) 384-393.

[10] K.C. Huang, Y.L. Tsai, H.C. Liu, Task ranking and allocation in list-based workflow scheduling on parallel computing platform, The Journal of Supercomputing 71(1)(2015) 217-240.

[11] Y. Wang, W. Shi, E. Berrocal, On performance resilient scheduling for scientific workflows in HPC systems with constrained storage resources, in: Proc. of the 6th ACM Workshop on Scientific Cloud Computing, 2015.

[12] L.F. Bittencourt, R. Sakellariou, E.R.M. Madeira, DAG scheduling using a look ahead variant of the heterogeneous earliest finish time algorithm, in: Proc. of 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010.

[13] M. Rahman, R. Ranjan, R. Buyya, Cooperative and decentralized workflow scheduling in global grids, Future Generation Computer Systems 26(2010) 753-768.

[14] F. Ding, R. Zhang, K. Ruan, J. Lin, Z. Zhao, A QoS-based scheduling approach for complex workflow applications, in: Proc. of the Fifth Annual ChinaGrid Conference, 2010.

[15] K. Bochenina, N. Butakov, A. Dukhanov, D. Nasonov, A clustering-based approach to static scheduling of multiple workflows with soft deadlines in heterogeneous distributed systems, Procedia Computer Science 51(2015) 2827-2831.

[16] J.S. Kim, C.J. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proc. of Int'l Conf. Parallel Processing, 1988.

[17] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Transactions on Parallel and Distributed System 5(9)(1994) 951-967.

[18] M. Wu, D. Gajski, Hypertool: a programming aid for message passing system, IEEE Transactions on Parallel and Distributed Systems 1(1990) 330-343.

[19] L.F. Bittencourt, E.R.M. Madeira, F.R.L. Cicerre, L.E. Buzato, A path clustering heuristic for scheduling task graphs onto a grid, in: Proc. of the 3rd ACM International Workshop on Middleware for Grid Computing, 2005.

[20] C.C. Hsu, K.C. Huang, F.J. Wang, Online scheduling of workflow applications in grid environment, Future Generation Computer Systems 27(6)(2011) 860-870.

[21] Z. Cai, X. Li, J.N.D. Gupta, Heuristics for provisioning services to workflows in XaaS clouds, IEEE Transactions on Services Computing PP(99)(2014) 1.

[22] H.R. Boveiri, An efficient task priority measurement for list-scheduling in multiprocessor environments, International Journal of Software Engineering & Its Applications 9(5)(2015) 233-246.

[23] H.R. Boveiri, List-scheduling techniques in homogeneous multiprocessor environments: a survey, International Journal of Software Engineering & Its Applications 9(4)(2015) 123-132.

[24] U. Fiore, F. Palmieri, A. Castiglione, A.D. Santis, A cluster-based data-centric model for network-aware task scheduling in distributed systems, International Journal of Parallel Programming 42(5)(2014) 755-775.

[25] O. Sinnen, Task Scheduling for Parallel Systems, Wiley-Interscience, New York, 2007.

[26] S.G. Ahmad, C.S. Liew, M.M. Rafique, E.U. Munir, S.U. Khan, Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems, in: Proc. of 2014 IEEE Fourth International Conference on Big Data and Cloud Computing (BdCloud), 2014.

[27] J. Zhang, J. Luo, F. Dong, Scientific workflow scheduling in non-dedicated heterogeneous multicluster with advance reservations, Integrated Computer-Aided Engineering 22(3)(2015) 261-280.

[28] B. Kruatrachue, G.T. Lewis, Grain size determination for parallel processing, in: Proc. of IEEE Software, 2010.

[29] X. Tang, K. Li, G. Liao, An effective reliability-driven technique of allocating tasks on heterogeneous cluster systems, Cluster Computing 17(4)(2014) 1413-1425.

[30] A. Gerasoulis, T. Yang, On the granularity and clustering of directed acyclic task graphs, IEEE Transactions on Parallel and Distributed Systems 4(6)(1993) 686-701.

[31] V. Sarkar, Partitioning and scheduling parallel programs for execution on multiprocessors, [dissertation] Stanford, CA: Stanford University, 1987.

[32] A. Gerasoulis, T. Yang, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, Journal of Parallel and Distributed Computing 16(4)(1992) 276-291.

[33] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, M. Samidi, Scheduling data-intensive workflows onto storage-constrained distributed resources, in: Proc. of the Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007.