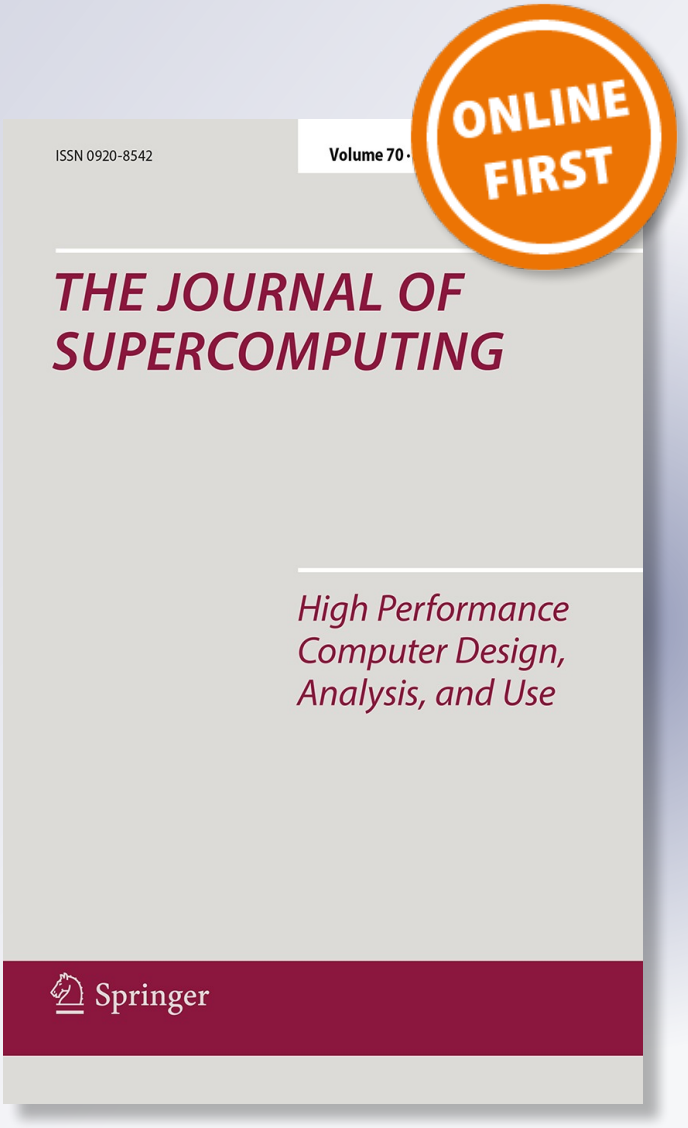
An optimized MapReduce workflow scheduling algorithm for heterogeneous computing

Zhuo Tang, Min Liu, Almoalmi Ammar, Kenli Li & Keqin Li

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-014-1335-2



Springer

J Supercomput
DOI 10.1007/s11227-014-1335-2

An optimized MapReduce workflow scheduling algorithm for heterogeneous computing

Zhuo Tang · Min Liu · Almoalmi Ammar · Kenli Li · Keqin Li

Abstract The MapReduce framework is considered to be an effective resolution for huge and parallel data processing. This paper treats a massive data processing workflow as a DAG graph consisting of MapReduce jobs. In a heterogeneous computing environment, the computation speed can be different even on the same slot depending on various jobs. For this problem, this paper proposes an optimized MapReduce workflow scheduling algorithm. This algorithm comprises a job prioritizing phase and a task assignment phase. First, the jobs can be classified as I/O-intensive and computing-intensive, and the priorities of all jobs are computed according to their corresponding types. Then, the suitable slots are allocated for each block, and the MapReduce tasks in the workflow are scheduled with respect to data locality. The experimental results show that the optimized MapReduce workflow scheduling algorithm can improve the performance of task scheduling and the rationality of resources allocation in heterogeneous computing.

Keywords Hadoop · Heterogeneous cluster · MapReduce · Scheduling · Workflow

1 Introduction

As data collection volumes grow rapidly, some complex computations are beyond the ability of our classical processing methods. This challenge requires that scientific computings have the abilities to handle massive amounts of data. One of the most suc-

Z. Tang (✉) · M. Liu · A. Ammar · Kenli Li · Keqin Li
College of Information Science and Engineering, Hunan University, Changsha 410082, China
e-mail: ztang@hnu.edu.cn

Keqin Li
Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

Published online: 19 November 2014

cessful frameworks for this propose is MapReduce [9], which has been implemented on some platforms such as GFS and Hadoop that process massive datasets in parallel ways. For data-intensive applications, MapReduce has increasingly attracted attention in many areas (e.g., energy mining [24], image processing [15], and meteorological analysis [5], etc). Unfortunately, some projects are composed of a set of MapReduce jobs with priority constraints, and there is lack of an efficient mechanism to schedule such projects, which is not a simple problem for MapReduce platforms such as Hadoop.

In order to solve these problems, researchers are trying to integrate MapReduce into workflow. Through this integration, not only can the whole parallelism be fully extended, but also the HDFS of Hadoop can lend the workflow a helpful hand to avoid the bottlenecks when processing mass datasets [25]. For example, the studies in [8,12,21,27] advanced the workflow processing capability of Map/Reduce, and provided support for the workflow level developers.

With the rapid development of information technology, the concept of workflow has been applied to automate large-scale science, and usually, a workflow is represented by a directed cyclic graph (DAG) [30]. It is worth mentioning that scheduling a DAG application is NP-hard in general [14], which is a challenge for researchers. In early stage, a few studies [7,19] proposed a MapReduce-enabled workflow system for special applications combined with other areas with exclusive requirements, but these researches proposed this system for a homogeneous cluster.

Recently, with the increasing demand on computing performance, CPU cannot satisfy it anymore. Therefore, the high-performance computing (HPC) industry's development turns to constituting between GPU and CPU to work together for addressing common tasks in heterogeneous clusters [1]. In this field, [6,22,23] introduced an adaptive scheduler providing dynamic resource allocation across jobs and hardware affinity in a heterogeneous cluster. In these works, the computing resources in the heterogeneous cluster are simply divided into the generic and accelerated pools, and the scheduling policy used for MapReduce tasks is based on the plain FIFO policy [29], which is a valuable enhancement.

Through the analysis of the MapReduce-enabled workflow system, we come up with an important observation, i.e., for one workflow model, the whole project can be considered as a DAG consisting of various jobs, and the DAG can be expanded into a bigger graph which refines each job into a set of tasks. Currently, there are few scheduling policies considering the composition of DAG scheduling and MapReduce scheduling.

In this paper, our contribution is to integrate MapReduce into a workflow, additionally to enrich a heterogeneous cluster by considering different combinations of performance and configuration of processors and constructing different computing resource pools.

We propose an optimized scheduling policy—MRWS (*MapReduce-enabled workflow scheduler*). MRWS adopts some ideas from heuristic algorithms [26] (e.g., HEFT (heterogeneous earliest-finish-time)), and realizes the schedule of the jobs in a MapReduce workflow through splitting a job into basic tasks which can be distributed to appropriate slots.

For a general DAG which consists of indivisible tasks, HEFT is better than other classical heuristic scheduling algorithms in most situations, and in particular, gives the best performance and speedup results for DAGs with high parallelism. Since this model is designed for processing massive data-intensive jobs, adopting the idea of HEFT is a good choice for MRWS. In this paper, jobs that can get benefit from accelerated machines are defined as CPU-intensive jobs, and the others are defined as I/O-intensive jobs. For the advantage of this model, each job of a DAG can be split into a number of tasks, and part of them can be scheduled on the idle time slots even if it is smaller than the demand time of a total job. Through this process, we can achieve higher efficiency and shorter makespan.

In coordinating a cluster, scheduling I/O-intensive jobs in the accelerated pool $Pool_{acc}$ may bring no benefit and result in competition with the CPU-intensive jobs which in fact could take advantage of them. A heterogeneous environment consists of computing, storage, and network resources with different capability and availability. But we only consider the computing power and the network bandwidth properties. In order to add awareness of the hardware heterogeneity into a scheduler, in a heterogeneous cluster, we need to group pools, with accelerated machines into $Pool_{acc}$ and regular machines into $Pool_{reg}$. Comparing to the references [22,23], these two pools are subdivided into more levels according to computing performance. Through this method, the resources can be used more effectively, and it is also more adapted to the characteristics of the HEFT algorithm. In an initial phase, by comparing the observed average task time (Treg) on $Pool_{reg}$ with that (Tacc) obtained on $Pool_{acc}$, we can get a conclusion that if Treg/Tacc reaches certain numerical value, then the job type is I/O-intensive; otherwise, it is CPU-intensive [22]. For a CPU-intensive job, its parameter T is marked as CPU; similarly, for an I/O-intensive job, its parameter T is marked as I/O. Hence, a job's parameter T is in favor of scheduling the job in the appropriate pool, thus improving the overall efficiency.

The remainder of the paper is organized as follows: section 2 reviews the related work. Then, Sect. 3 introduces the MapReduce-enabled workflow model and especially describes the MRWS algorithm. Experiments and analysis which support our contributions are presented in Sect. 4. Finally, Sect. 5 concludes this paper and describes the future work.

2 Related work

There are two main architectural approaches to implementing workflow, i.e., service orchestration and service choreography [3]. In orchestration, during part or the whole process, information or tasks are passed through a central engine from one participant (a machine, human, or resource) to another for an action, according to a set of procedural rules. Based on their functions, workflows can be classified into two types, i.e., business workflows and scientific workflows. Business workflows have been around for many years, which aim to automate and optimize an organization's processes accomplished by human or computer agents in a management system [17]. Especially in large organization, its processes often refer to various systems, roles, objects and the partial order or coordination among different activities. In addition,

the data structure, computing infrastructure are often heterogeneous. When workflows move from business sectors to scientific laboratories, the need to support large-scale, complex, fault-tolerant, and maintainable scientific processes and scientific workflows arise.

In this paper, we adopt orchestration, because it can span multiple applications and/or organizations. In addition, we choose a scientific workflow as composition of framework, because a scientific workflow tends to have a dataflow-oriented model and can abstract the details of a business process, while a business workflow places an emphasis on control-flow patterns and events [3]. A scientific workflow has recently become an enabling technology to automate and speed up the scientific discovery process, and there are much recent research interests in it, e.g., [11,18,28]. However, in orchestration, the engine can cause a bandwidth bottleneck when the workflow processes a huge dataset [4].

Currently, as HPC is widely and deeply used in various fields, the processing of some special application scenarios gets more complex [22,23]. This complexity is generally manifested by having more MapReduce jobs with priority restrictions among them, rather than having more complex map and reduce functions. In order to resolve this problem, researchers tried to translate it into a MapReduce workflow [7,19]. Thus, there are a few studies focusing on constructing MapReduce-enabled workflow systems in which a huge project, especially a data-intensive one, can be expressed as a DAG consisting of a set of MapReduce jobs. These systems can provide high parallelism for MapReduce-based workflows.

Now scientific workflow is becoming an efficient computational model. Some scientific workflow applications make contribution to the scientific researches, such as Kepler [27], Taverna [20], VIEW [16], Pegasus [10], and so on. Kepler is one of the globally fastest and most efficient HPC frameworks, which is designed to help scientists, analysts, and computer programmers to create, execute, and share models and analyses across a broad range of scientific and engineering disciplines. However, the workflow application is attached on top of Hadoop and scheduling of this packaged Hadoop is separated from the workflow management system, which unavoidably causes some performance overheads. Therefore, in some way, the performance of Kepler system can be more advanced in the future.

Oozie [2] is a server based workflow engine specialized in running workflow jobs with actions that execute Map/Reduce and Pig jobs on Hadoop. Oozie workflow definitions are written in hPDL (a XML Process Definition Language). In Oozie, actions of the workflow are also arranged in a DAG. However, it does not provide an optimization mechanism to schedule workflows considering scheduling issues, because "control dependency" from one action to another means that in the queue the second action cannot run until the first action has completed, and the XML Language cannot optimize the parallelism of actions [2].

There are some practical examples of carrying out some applications for many specific domains in the MapReduce-enabled workflow systems [7,19]. In homogeneous platforms, [7] proposed a high-performance system MRGIS (MapReduce Geographical Information System), which is a parallel and distributed computing model based on MapReduce clusters and can significantly improve the performance of GIS. Similarly, [19] introduced a model to perform intensive processing on climate satellite data with

high performance. However, their schedulers are simple and cannot support complex workflows.

Merging the GPU together with CPU into a heterogeneous cluster is the inevitable trend of future HPC. Additionally the workloads demand heterogeneous resources to get higher performance, because some workloads may be CPU-intensive whereas others are I/O-intensive. With hardware awareness of the superior performance and energy efficiency of the heterogeneous resources, these specialized workloads are assigned to their preferred slots, so that each slot can play its respective advantages and gain high performance especially when the set of jobs of a project share the resources.

Considering the defects in workflow scheduling in a heterogeneous cluster environment, this paper aims to construct a MapReduce-enabled scientific workflow model with static optimization scheduling policy in heterogeneous clusters to get good performance.

3 The MapReduce-enabled workflow model

3.1 Formation of the model

In this paper, we presume that a small-scale heterogeneous cluster may be private or rented from a public business cloud services for a single user only [29,30]. We further assume that the heterogeneous cluster is made up of several different pools. The pools, respectively, represent different combinations of homogeneous machines. Heterogeneous hardware (such as CPU, GPU, or SPUs in the Cell/BE processor, etc.) may include several different platforms. There are some popular programming framework for heterogeneous platform (i.e., OpenCL [13]) which can support different parallel platforms, while we can also emulate such behavior by implementing the specific jobs with multiple versions of their programming codes.

Table 1 shows that during the resource allocation and DAG analysis phases, we can just use JAVA as our programming language. However, in task scheduling and distribution phase, due to the heterogeneous computing evironments, after reading the input datas from a uniform distributed storage HDFS, the task processes will be finally compiled and converted as the processor execution languages. In this way, tasks can be executed on the appropriate node according to its own special features and character.

This paper proposes an improved algorithm which consists of two major phases shown in Fig. 1, i.e., a job prioritizing phase for generating the scheduling priority queue among all jobs on the workflow level, and a processor selection phase for

Table 1 Codes of different levels

Layered structure		Programming Framework
Workflow-level	DAG analysis	JAVA
	Resource location	JAVA
MapReduce-level	Task scheduling	OpenCL, Brook + CBE, CUDA, Cell-accelerated

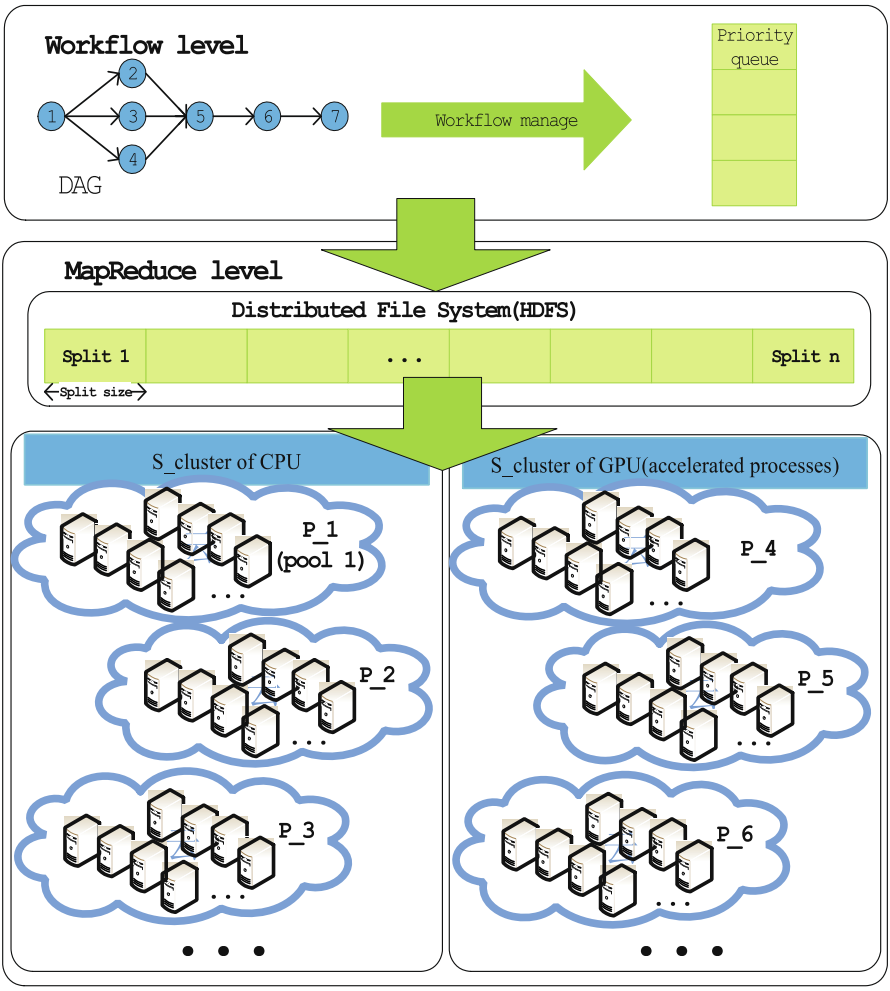


Fig. 1 The MapReduce-enabled scientific workflow model

selecting the jobs in the priority queue and allocating each block of the job on its best slot. The purpose is to be prepared to schedule tasks on the corresponding slots, and try to realize data locality on the MapReduce level. By effective combination of the features of the two phases, we aim to minimize the makespan and achieve high performance.

In the heterogeneous cluster as shown in Fig. 1, we define the small cluster composed of CPU devices as S_cluster of CPU and the small cluster composed of GPU devices as S_cluster of GPU. The small clusters are divided into resource pools and named as Pool_n (shortly P_n), and in each pool the slots have the same performance. While the devices in the same pool may not actually have exactly the same performance, in this paper, we do not emphasize on the detailed analysis of the mechanism to deal with this condition.

Without loss of generality, we consider MapReduce jobs to be the basic components of the workflow. Therefore, this special workflow model is denoted as

$$W = (J, E, D, T), \tag{1}$$

where the components are explained as follows:

- W denotes the workflow.
- J is a set of jobs, some of which are MapReduce jobs, and therefor, each of them can be split into a number of tasks. For other indivisible jobs, each one is just regarded as a task.
- E is a set of edges between job nodes, each has a value denoting the communicate cost.
- D is a set of data, in which each unit D_t is a two-tuple $(D_{t_{in}}, D_{t_{out}})$, which denotes the input data set and the output data set in the workflow process respectively.
- T is a set of properties of the tasks which are used to mark the jobs with the types (e.g., I/O- or CPU-intensive), which help in giving attributions to scheduling the tasks of special jobs on the applicable slots.

3.2 Scheduling algorithm in workflow

As mentioned above, a parallel MapReduce-enabled workflow can be represented by a DAG $W = (J, E, D, T)$, in which each node indicates a job, and each one can be completed with the MapReduce parallel computing model. A sample DAG is presented in Fig. 2, in which the job without parent is called an *entry task* and the job without child is called an *exit task*.

On the one hand, a job's execute time can be estimated as

$$W(j, D_{j_{in}}, D_{j_{out}}) = Size(max(D_{j_{in}}, D_{j_{out}}))/V_{p_n}(j), \tag{2}$$

where $D_{j_{in}}$ and $D_{j_{out}}$ are the input and output data sets of the job j, and $V_{p_n}(j)$ is defined as the processing speed of the node in the processor of type P_n. In this paper, the type of the processors in Pool(n) is defined as P_n. The communication time $C_{i,j}$ between jobs can be calculated as

$$C_{i,j} = Size(D_{j_{in}})/V_{tr}(j), \tag{3}$$

where $V_{tr}(j)$ is the transmission rate of the task j, which is different on different racks and different data centers. In classical heuristic algorithms, $C_{i,j}$ is the communication time from the node executing n_i to the node executing n_j. But in MRWS, it is the time from HDFS receives the output of n_i to HDFS finishes the allocation of the split data sets onto slots and ready for n_j. For simple understanding, in this paper, we describe $C_{i,j}$ as the communication time from job n_i to n_j .

For the computing time and communication time, in an initial phase, we can work out $V_{p_n}(j)$ and $V_{tr}(j)$ by testing on different pools and then calculating them with the

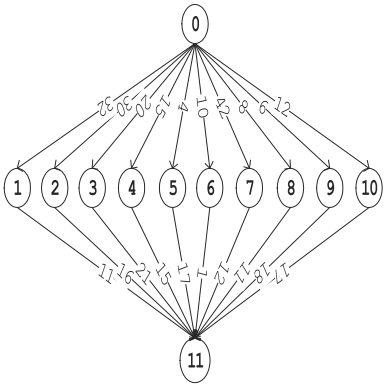


Fig. 2 A sample project graph with 12 MapReduce jobs

Table 2 Computation costs of jobs in each pool

Job_ID	Type of CPU intensity or I/O intensity	The cost of a job on each pool (in sec)					
		S_cluster of CPU			S_cluster of GPU		
		P_1	P_2	P_3	P_4	P_5	P_6
j_0	CPU	140	120	100	60	40	20
j_1	I/O	98	84	70	70	70	70
j_2	CPU	210	180	150	90	60	30
j_3	I/O	182	156	130	130	130	130
j_4	I/O	280	240	200	120	80	40
...	...	...	...	...	...	...	...

corresponding formulas. Furthermore, we can directly estimate them by processing small sets of jobs on the cluster in advance.

Accordingly to the DAG of workflow shown in Fig. 2, the jobs' computation cost in each pool and their types are shown in Table 2. The computation cost can be obtained through the estimate methods described above.

In this paper, jobs in the workflow are scheduled by MRWS, which extends the classic heuristic algorithm HEFT [26]. In the algorithm, jobs are explicitly sorted by their scheduling priorities which are based on the upward and downward rankings. The *upward rank* and the *downward rank* are defined as follows:

$$rank_u(j_i) = \overline{w_i} + \max_{j_j \in succ(j_i)} (\overline{c_{i,j}} + rank_u(j_j)), \tag{4}$$

$$rank_u(j_{exit}) = \overline{w_{exit}}, \tag{5}$$

$$rank_d(j_i) = \max_{j_j \in pred(j_i)} (rank_d(j_j) + \overline{w_i} + \overline{c_{i,j}}), \tag{6}$$

where $\overline{w_i}$ is the average computation cost of job j_i, $\overline{c_{i,j}}$ is the average communication cost from job i to j, $succ(j_i)$ is the set of immediate successors of job j_i, and $pred(j_i)$ is the set of immediate predecessors of job j_i. The *upward rank* starts from

Table 3 Values of attributes used in the scheduling algorithms for DAG in Fig. 2

Job_ID	$Rank_u$	$Rank_d$	$Rank_s$
j_0	740	0	570
j_1	328	112	240
j_2	376	110	240
j_3	404	100	240
j_4	415	95	240
j_5	457	84	240
j_6	338	90	240
j_7	582	122	240
j_8	405	88	240
j_9	522	86	240
j_{10}	393	92	240
j_{11}	240	464	0

the exit job and computed recursively by traversing the task graph upward. Similarly, the *downward rank* is computed recursively by traversing the task graph downward starting from the entry job, but the downward rank value of the entry job j_{entry} is zero.

As proposed in reference [7], let $c(j^i, D_{j^i_{in}}, D_{j^i_{out}})$ denote the execution cost of a task j^i, where $D_{j^i_{in}}$ and $D_{j^i_{out}}$ denote the input dataset and output dataset of j^i, respectively. In other words, the execution cost of a task is related to what operation it performs and the sizes of input and output datasets. For a task j^i, its finishing cost is defined by the following equation:

$$rank_s(j_i) = \sum_{j^i \in \{depd-descendant_w(j)\}} c(j^i, D_{j^i_{in}}, D_{j^i_{out}}), \tag{7}$$

where $c(j^i, D_{j^i_{in}}, D_{j^i_{out}})$ denotes the average computation cost, and $depd - descendant_w(j)$ is the descendant set of job j. The value of $rank_s$ is used to produce the scheduling sequence in [7].

The corresponding estimated rank values are illustrated in Table 3. With the upward rank policy, by non-increasing order of $rank_u$, the scheduling order of the jobs is (j_0, j_7, j_9, j_5, j_4, j_8, j_3, j_{10}, j_2, j_6, j_1, j_{11}). In the same way, with the downward rank policy, sorting the jobs by non-decreasing the order of $rank_d$, the scheduling sequence is (j_0, j_5, j_9, j_8, j_6, j_{10}, j_4, j_3, j_2, j_1, j_7, j_{11}). More easily we can simply sort the finish time of the jobs by non-increasing order of $rank_s$, and the scheduling sequence is (j_0, j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, j_{10}, j_{11}).

3.3 Resource allocation in MapReduce

In a heterogeneous cluster, slots differ from one to another. For a heterogeneous hardware structure, the processing time can also be different even on the same slot depending on various jobs. Hence, for all jobs in the DAG, the corresponding computing time

Table 4 Computation costs of tasks on each type slot (job 0's corresponding task is described as a_i.)

Tasks of jobs	Type of jobs	Number of map tasks	The processing time of a task on a slot					
			S_cluster of CPU			S_cluster of GPU		
			P_1	P_2	P_3	P_4	P_5	P_6
$0(a_i)$	CPU	20	7	6	5	3	2	1
$1(b_i)$	I/O	14	7	6	5	5	5	5
$2(c_i)$	CPU	30	7	6	5	3	2	1
$3(d_i)$	I/O	26	7	6	5	5	5	5
...	...	...	...	...	...	...	...	...

of tasks on a slot in each variable pool is known as shown in Table 4. The computing time can be obtained in the initial stage. After several map tasks of each job are executed in each pool, the computing time can be estimated in the initial stage.

Hadoop runs the input job by dividing it into tasks, which makes the processing to become better load-balanced. The computing speed and transfer speed can affect the task by determining the split size. For most jobs, a good split size tends to be the size of the HDFS block (64 MB by default), so the number of tasks can be defined as

$$N\_m = Size\ of\ job\ \ j / Size\ of\ block. \tag{8}$$

The resource allocation policy is as follows: TSD(TaskID, Proc, StartT, EndT) records the detailed information of the scheduling results, where TaskID is the mark of the task; Proc is the mark of the slot on which the task is assigned, and StartT and EndT are the start time and end time of the task. Additional variables are listed as follows:

- $t\_EST[t_i, s_i]$ and $t\_EFT[t_i, s_i]$: respectively, represent the slot s_i that can afford the earliest execution start time and finish time for task t_i.
- $t\_AST[t_i]$ and $t\_AFT[t_i]$: respectively, represent the actual execution start time and finish time of task t_i.
- $AST[j_i]$ and $AFT[j_i]$: respectively, represent the actual execution start time and finish time of job j_i.

For the first entry job j_{entry} in a DAG, $AST[j_{entry}] = t\_EST[t_{entry}, s_i] = 0$. The scheduling starts from the task t_{entry} which belongs to the entry job j_{entry}. As shown in (9) and (10), the values t_EST and t_EFT are worked out through a recursive method:

$$t\_EST[n_i, s_i] = \max\{readyTime(j_i), slotavailTime(s_i)\}. \tag{9}$$

$$t\_EFT[n_i, s_i] = t\_EST[n_i, s_i] + w(t_i, s_i). \tag{10}$$

$$readyTime(j_i) = \max_{n_m \in pred(n_i)} (AFT(n_m) + C_{m,i}). \tag{11}$$

In order to compute the t_EST of a task t_i split from job j_i, all immediate predecessor jobs of j_i must be scheduled, where $pred(n_i)$ is the set of immediate predecessor jobs of job j_i.

Then $readyTime(j_i)$ is the time point stamp at which all the data needed by job j_i are ready for it. For the scheduling policy is noninsertion-based, $slotavailTime(s_i)$ is the time stamp when the slot s_i completes the former allocated tasks and is ready to execute another task. $w(t_i, s_i)$ gives the estimated execution cost to complete task t_i on slot s_i.

When a circulate computing of the task t_i on each slot is done, by selecting the previous complete time $t\_EFT[n_i, s_m]$ as the actual finish time $t\_AFT[t_i]$, we can get the actual start time $t\_AST[t_i] = t\_AFT[t_i] - w(t_i, s_i)$. When all tasks of job j_i are scheduled, the last finish time of these tasks can be regarded as the job j_i's actual finish time: $AFT[j_i]$.

3.4 Combination algorithm description

The combination algorithm is described in Algorithm 1, which contains three subprocesses. These are all the subprocesses of the model MRWS:

Algorithm 1 MRWS (MapReduce-Enabled Workflow Scheduler)

Require:
Workflow DAG $W = (J, E, D, T)$ and a set of predict parameters.
Ensure:
The solutions.
1: Prior_order();
2: P_scheduling();
3: E_scheduling().

Algorithm 2 Priority Ranking (which ranks the priority of jobs in a DAG)

Require:
Workflow DAG $W = (J, E, D, T)$ and some information like Table 4 and so on.
Ensure:
The solutions.
1: **Prior_order()** {
2: Comp_$Rank_u$();
3: Sort_$Rank_u$();
4: }

The first subprocess is shown in Algorithm 2, in which the function Prior_order() aims to sort the priority queue of the jobs of the DAG to satisfy the precedence constraints. While the step $Compute\_Rank_u()$ works out $rank_u$ for all the jobs, which is shown in the first column of Table 3 by traversing the DAG upward and starting from the exit job, and function $Sort\_Rank_u()$ sorts the $job\_priority\_queue$ in non-increasing order according to the $rank_u$ values.

Algorithm 3 Preceding Analog Scheduling (which distributes tasks of each job onto the ideal slots)

Require:
Workflow DAG $W = (J, E, D, T)$, $job\_priority\_queue$, $j_i\_tasklist$, $slot\_set()$;
Ensure:
The solutions:
1: **P_scheduling**() {
2: **while** there are unscheduled jobs in the $job\_priority\_queue$ **do**
3: select the first job j_i, from the queue for scheduling;
4: **for** each task t_i (split from job j_i) in $j_i\_tasklist$ **do**
5: **for** each slot s_i in the $slot\_set()$ **do**
6: compute the value of $t\_EST[n_i, s_i]$ and $t\_EFT[n_i, s_i]$.
7: **end for**
8: Assign tasks t_i to the slot s_m which minimizes t_EFT of task t_i;
9: save the minimum time in t_EFT;
10: save the corresponding detail TSD(TaskID,Proc, StartT, EndT) in TSD_list;
11: delete the task t_i from $j_i\_tasklist$;
12: **end for**
13: delete the job j_i from the $job\_priority\_queue$;
14: Save $AFT[j_i]$ equals the last t_EFT of the tasks split from job j_i;
15: **end while**
16: }

Algorithm 4 Actual Scheduling (which is the implementation of Preceding Analog Scheduling with the control of HDFS)

Require:
$job\_priority\_queue$, $j_i\_tasklist$, TSD_list;
Ensure:
The solutions:
1: **E_scheduling**() {
2: **while** there are unexecuted jobs in $job\_priority\_queue$ **do**
3: select the first job j_i, from the queue for scheduling;
4: **for** each task t_i (split from job j_i) in $j_i\_tasklist$ **do**
5: Split_block();
6: scheduling();
7: delete the task t_i from $j_i\_tasklist$;
8: delete the information of task t_i in TSD_list;
9: **end for**
10: delete the job j_i from the $job\_priority\_queue$.
11: **end while**
12: }

Second, as shown in Algorithm 3, Preceding Analog Scheduling is based on the front of $job\_priority\_queue$. This algorithm is designed to submit the tasks which are split from each job while the whole jobs are not all distributed to the ideal slots. Specially, the function $P\_scheduling()$ is working in initial/pretreatment phrase, which combines with MapReduce polices.

In Step 4, the corresponding $j_i\_tasklist$ (values can be obtained from tasks of job j_i), and each value of the same job is on the same level. In Step 5, the $slot\_set$ is the set of all available slots. Between Step 4 and Step 12 , it acquires each $t\_EST[n_i, s_i]$ and $t\_EFT[n_i, s_i]$, which are the earliest execution start time and finish time of task t_i on slot s_i through the two layer nesting loop. The parameters in this algorithm can

be worked out though (9)(10)(11). Step 8 is to choose the minimal $t\_EST[n_i, s_i]$ of task t_i, and to save the corresponding detailed TSD in TSD_list. Finally, in Step 14, $AFT[j_i]$ is the time point when all the tasks of job j_i have been executed. If all the jobs in the DAG graph are scheduled, then the schedule length of the graph is the actual finish time of the last task of the exit job.

Due to the combination with the MapReduce policy, the smallest scheduling unit is a task not a job. In general, the task unit's execution time is obviously much shorter than the job unit. So, with the priority constraint, one of the special features of the MRWS is to enable these task units to insert into the idle slots in a flexible way, and the efficiency of the scheduler will obviously be advanced, even if the idle period of the slot is not long for the whole job.

Finally, as shown in Algorithm 4, the subprocess is the actual implementation of MRWS. MapReduce is the important data process framework of Hadoop, while Hadoop's another pivotal component is HDFS (Hadoop distributed file system). In this paper, the combination model also adopts HDFS to manage the bottom data.

In Step 5, HDFS splits the input data files of job j_i into blocks though equation (8). While in Step 6, according to TSD_list, HDFS schedules the corresponding tasks and executable codes onto the slots and then waits for execution. This policy has realized data locality optimization, without considering the error tolerance copies.

Figure 3 presents the details of the scheduling procedure for the DAG shown in Fig. 2 with an example, which is obtained by the MRWS policy. As shown in Fig. 3, in the experiment of this paper, the heterogeneous cluster is supposed to consist of 18 slots (c1–c18). Each pool contains 3 slots, for example, P_1 consists of c1–c3 which have the same computing capability. In the heterogeneous cluster, the computing cost of tasks on each type slot, which is denoted as $S\_cluster$, are defined in Table 4. In Fig. 3, as the description of MRWS, the basic lattice of the vertical coordinate represents one unit time, and the priority queue is on the right of the figure.

The numbers in each small rectangle means that this slot duration belongs to the corresponding task and job.According to the task priority queue, as Fig. 3 shows, the job 0 is the first scheduled task. After job 0 finished, based on its task executing time and the data transfer duration, we can acquire the start time of the job 7, and schedule it as job 0. Then, job 9 is scheduled to be inserted into the space between job 0 and job 7 according to the schedule method in Algorithm 3. In the same way, job 5 will be scheduled to the idle duration between job 0 and job 9. Because this gap is too small to place the whole job, only part of the job 5 can be scheduled into this duration, and the remainders can be inserted into the idle duration between job 9 and job 7. All the DAG jobs in the example are scheduled by this mean. In the scheduling algorithm facing to the workflow, the job will be blocked if it is not scheduled suitably, and it will also affect other jobs to be distributed to the suitable slots. Through the scheduling mechanism in this paper, the jobs can be assigned to the right nodes according to the types of processing tasks. And based on the method "insert into the idle duration", comparing to the other schedule algorithms, the shortened makespan and the improved resource utilization are illustrated in the experiments.

Following the priority queue, the scheduling starts from job 0. Figure 4 presents the details of scheduling, in MRWS algorithm, for splitting the tasks of job 0 on slots. From Table 4 we can know that the job 0 is a CPU (compute-intensive) job, and it can

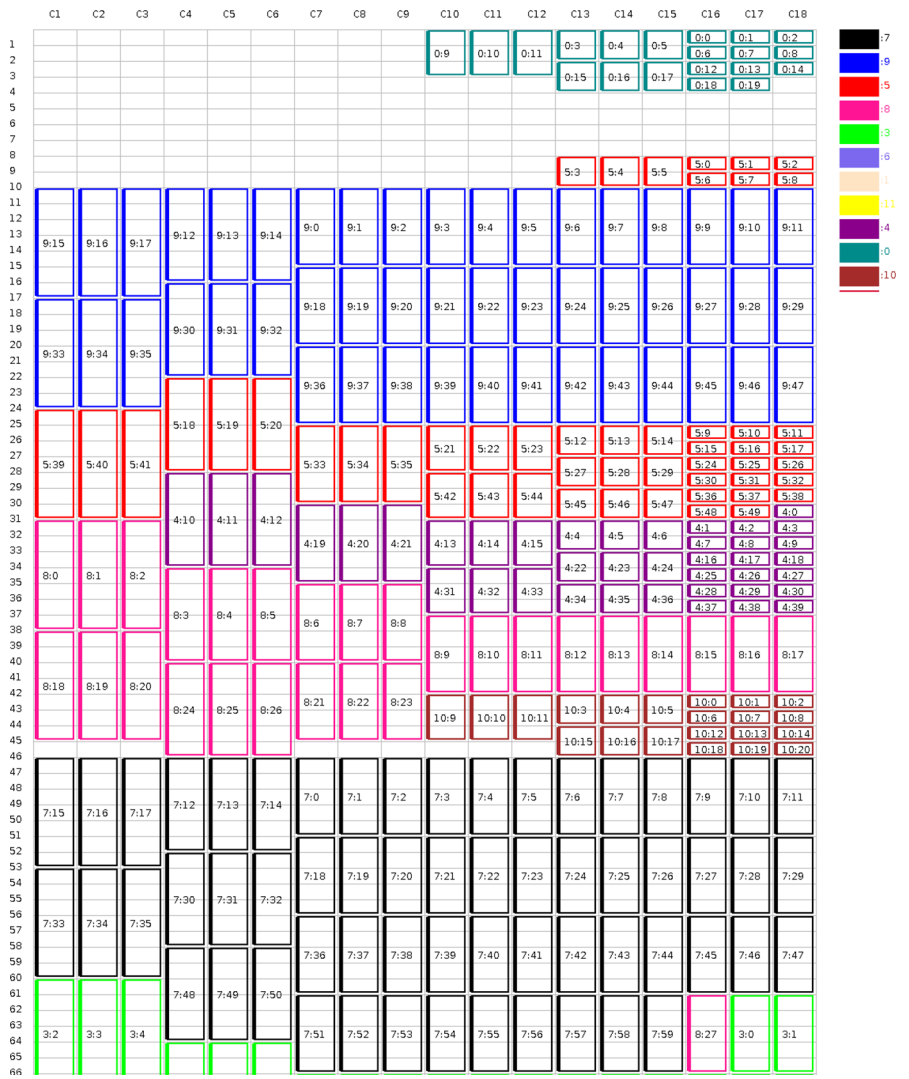


Fig. 3 Detailed resource allocation of the sample DAG

be split into 20 small map tasks to execute and these tasks' required unit execution time on the slot from pool P_1 to P_6, respectively, are (7, 6, 5, 3, 2, 1). The scheduling detailed process is described as follows: at first, we select task 0 from the task_list and according to Algorithm 3 of MRWS and find those slots (C16, C17, and C18) which can afford the earliest finishing time (t_EST). Because the three slots are beyond P_6, they have the same performance.

Thus, in this paper, due to the sequencing of the slot_set(), we select C16 to execute task 0 of job 0, which takes up to one unit of time on this slot. And then, the next tasks of job 0 also use the similar methods to choose their optimal slots.

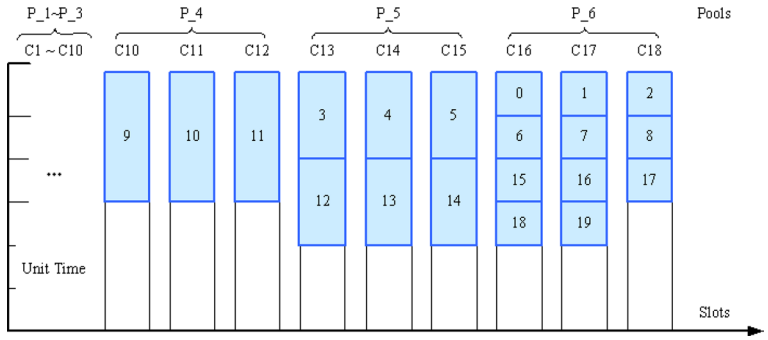


Fig. 4 The scheduling process in algorithm MRWS

Specifically, during the scheduling process, assigning a job to improper slots will not only make the work process slow, but may also prevent other jobs that prefer the slot from executing on it. Hence, when t_EST of a task on several slots have the same values, if the corresponding job is an I/O job, it will be scheduled to S_cluster of CPU, leaving the GPU cluster for later use by computation-intensive jobs.

After scheduling job 0, through Algorithm 3 we can determine job 7's start time which is the maximum value of the sum of the father node's finish time and the transmission time of job 7's required data. By scheduling job 9 after job 0 and 7, we can find an appropriate period of idle time between job 0 and job 7. Because the period is enough to insert all tasks of job 9, the job 9 is plugged in this time. For the job 5, we can find that part of job 5 can be scheduled in the period between job 0 and job 9, and the other parts have to be inserted in the period between job 9 and job 7. In the same way, we can process the remaining jobs in Fig. 3. As a result, the schedule length is shorter than other related policies.

In the next section, through the experiments, it can be illustrated that Algorithm MRWS can shorten the makespan, improve the jobs' efficiency, and reduce energy wastage.

4 Experiments and analysis

4.1 Randomly generated application DAGs

4.1.1 Performance results

In the following experiments, we compare the average metrics values of 3,000 different graphs which are generated randomly with the following initial parameters.

- *N_jobs*: It denotes the number of jobs.
- *N_tasks*: It denotes the number of tasks.
- *CCR*: It denotes the ratio of the average communication cost to the average computation cost. If a graph's CCR value is very low, it can be considered as a computation-intensive application.

– ∂: It is the shape parameter of the DAG graph. If $\partial >> 1.0$, it will generate a high parallelism graph; if $\partial << 1.0$, it will generate a long graph with low parallelism degree.

In the experiments, the compared scheduling policies are listed as follows:

– MRWS_NPI: A MRWS without "plugging in idle duration" policy. It is a compared algorithm in this experiment.
– SWS: A simple workflow scheduling algorithm proposed in [7]. In this method, the rank queue is generated according to latest finish time of the jobs. When it schedules a job based on the priority queue, it will wait until the former is finished and then schedule the next. Obviously, this algorithm does not have the character of parallelism on workflow level. This algorithm is similar to the simple algorithm of Oozie.
– WS_NWH: A general workflow scheduling algorithm for a homogeneous environment without awareness of hardware. In this policy, all the slots are considered the same, so during the execution, the quick slots should often wait until the slow slots get finished so as to finish a job.

The performance of the algorithms are compared with respect to various graph sizes in Fig. 5 and various CCR in Fig. 6.

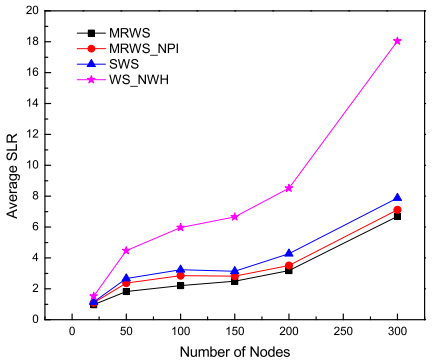
4.1.2 Performance analysis

To compare the performance of these scheduling policies, the comparison metrics, i.e., makespan, SLR, speedup, efficiency are defined similar to [26], while the only difference is that in our definition, the calculation factor is for the scenario of assigning tasks, instead of single jobs to a processor.

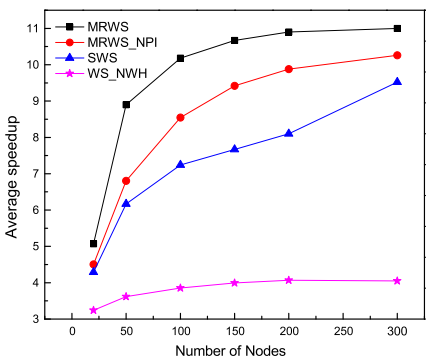
– SLR (Schedule Length Ratio): It normalizes the schedule length to a lower bound. In general, the scheduling algorithm which gives the lowest SLR of a graph is the best algorithm.
– Speedup: It denotes the effect of parallelization. It is computed by dividing the sequential execution time (i.e., cumulative computation costs of the tasks in the graph) by the parallel execution time (i.e., the makespan of the schedule).
– Efficiency: It is the ratio of the speedup value to the number of processors used.

The performance of these policies are compared with respect to various graph characteristics. Through numerous experiments, we find that when the CCR is fixed and only ∂ changes, the SLR will change rarely. Here we only consider the DAG with high parallelism, so ∂ is set with a big value, with $CCR \approx 1.24$ and $\partial > 15$. Fig. 5a, b, c shows the performance of the policies with respect to various graph sizes. Apparently, the average SLR and speedup values increase as the size of the DAG graph increases. These results demonstrate that the MRWS algorithm has obvious superiority in dealing with large workflows which can be denoted as complex DAGs.

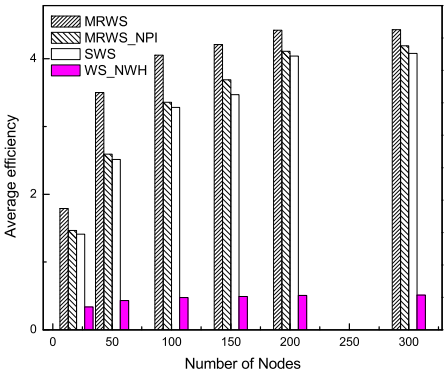
Next, the quality of the schedules for various CCR values are compared. When the node number of DAG is equal to 50, except of MRWS, the SLR of the policies with respect to various CCR values are compared in Fig. 6a. For the SLR ratio, for example, the ratio of MRWS_NPI is defined as (SLR_{MRWS_NPI} −



(a) Average SLR

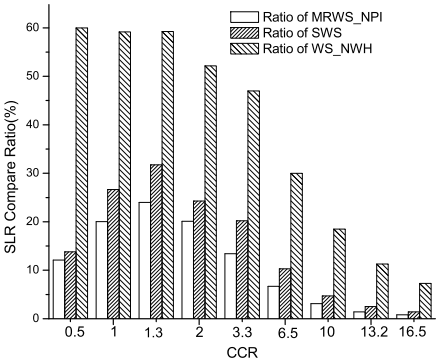


(b) Average speedup

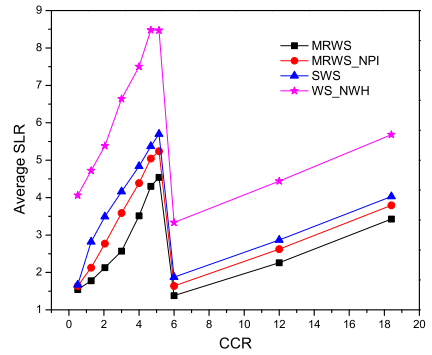


(c) Average efficiency

Fig. 5 The performance of the policies with respect to various graph sizes



(a) SLR compare ratio for various CCR



(b) Average SLR for various CCR

Fig. 6 The performance of the policies with respect to various CCR

$SLR_{MRWS})/SLR_{MRWS\_NPI}$. When the node number is equal to 100, from Fig. 6b, we can see the difference between the SLR of various scheduling policies with respect to CCR values.

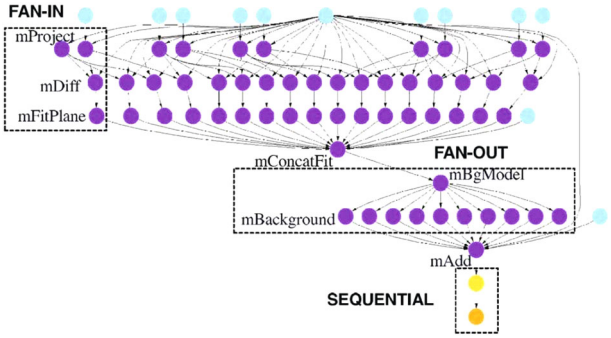
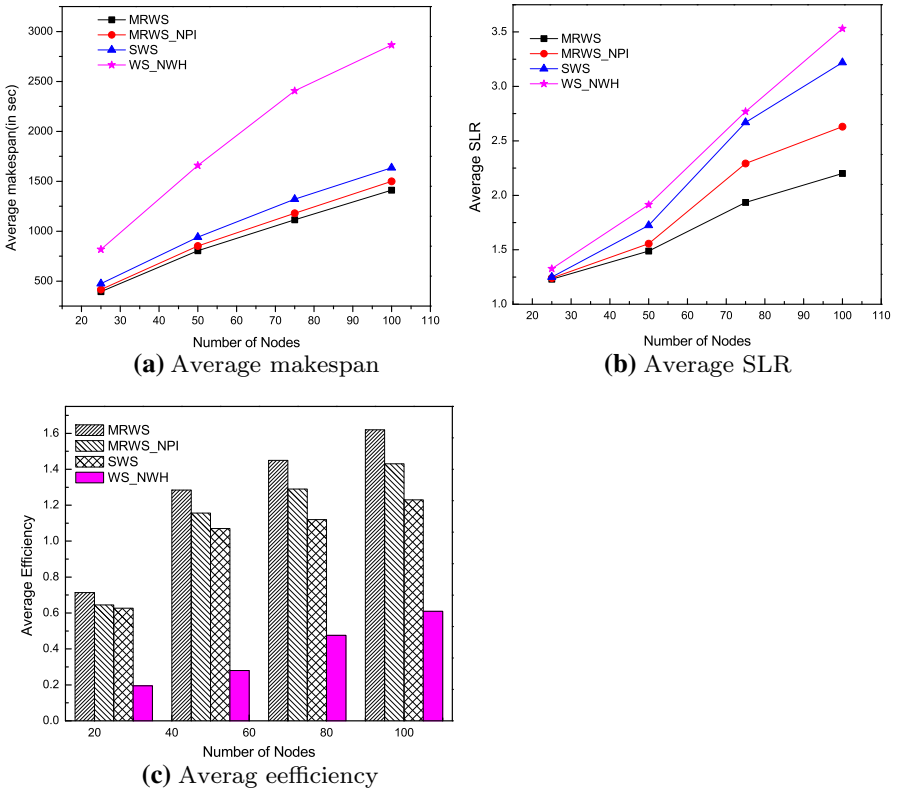


Fig. 7 Montage use-case scenario



(a) Average makespan

(b) Average SLR

(c) Averag eefficiency

Fig. 8 The performance of the policies with respect to various graph sizes

Based on these experiments, we observe that the MRWS policy outperforms the other policies for any graph size in terms of SLR, speedup, and efficiency. In addition to HEFT, the purpose of the MRWS algorithm is to find the appropriate processor and its periods for the jobs. So huge scale projects with high parallelism can generate more opportunity for plugging in, and it is more likely to realize the advantage of the

MRWS. In conclusion, the MRWS policy is a better choice for MapReduce-enabled workflows, especially for those with huge scale and high parallelism in heterogeneous systems.

4.2 Application DAGs of real word problems

Efficiently executing large-scale, data-intensive workflows common to scientific applications must take into account the volume and pattern of communication. For example, in Montage an all-sky mosaic computation can require at least 2 TB of data movement [31]. A typical montage workflow consists of 6 components: mProject, mDiff/mFitPlane, mConcatFit, mBgModel, mBackground, mAdd. Montage has some features of data-intensive scientific workflows [4]. First, it may result in huge data flow requirements. Second, its workflow pattern is common to many scientific applications. For the large-scale data-intensive scientific workflows, some researches manage to reduce the cost of communication on the orchestrations.

For the workflow shown in Fig. 7, we construct the corresponding DAG of the Montage scenario, and each node is a job which can be split into map and reduce tasks. For the experiments of Montage, the same CCR value was used.

Figure 8, respectively, gives the average SLR and efficiency of the policies when the number of nodes are varied from 25 to 100 with an increment of 25. It is also observed that MRWS is a practical and efficient policy for this application. Although the comparison objects are average performance values of policies, by this experiment, we can find that, for a special DAG graph, the larger difference in communication values and the higher performance will be obtained.

5 Conclusions

As the data collection volumes grow rapidly, some complex computation are beyond the ability of our classical processing methods. Considering the parallel processing for large-scale systems, a model combining MapReduce with workflows will bring a good solution to this problem. Currently, there are many researches on the scheduling policy for this combination model in homogeneous clusters or simple heterogeneous clusters. Through analyzing the defects in workflow schedule in the heterogeneous cluster environment, this paper proposes an optimization workflow scheduling algorithm. In this method, we treat a massive data processing workflow as a DAG graph consisting of MapReduce jobs and realize the schedule of the jobs in MapReduce-enabled workflows through splitting the job into basic tasks which can be distributed to the appropriate slots. In this paper, we propose an algorithm MRWS to optimize the scheduling to improve the performance and the experiments show the superiority of improving the schedule length and the parallel speedup for the workflow task in a heterogeneous environment.

Acknowledgments The authors are grateful to the three anonymous reviewers for their criticism and comments which have helped to improve the presentation and quality of the paper. This work is supported

by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005) National Natural Science Foundation of China (Grant Nos. 61103047,61370095).

References

1. http://www.hpc.ncep.noaa.gov/
2. Oozie. http://oozie.apache.org/
3. Barker A, Van Hemert J (2007) Scientific workflow: a survey and research directions. In: Proceedings of the 7th international conference on Parallel processing and applied mathematics, pp. 746–753. Springer
4. Barker A, Weissman JB, Hemert JI (2009) The circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration. Clust Comput 12(2):221–235
5. Barseghian D, Altintas I, Jones M, Crawl D, Potter N, Gallagher J, Cornillon P, Schildhauer M, Borer E, Seabloom E et al (2010) Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access and analysis. Ecol Inform 5(1):42–50
6. Calheiros R, Ranjan R, Beloglazov A, De Rose C, Buyya R (2011) Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw 41(1):23–50
7. Chen Q, Wang L, Shang Z (2008) Mrgis: a mapreduce-enabled high performance workflow system for gis. In: eScience, 2008. eScience'08. IEEE Fourth International Conference on, IEEE, pp. 646–651
8. Craddock Tracy Harwood (2008) e.a.: e-science: relieving bottlenecks in large-scale genome analyses. Nature Publishing Group, pp. 948–954
9. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation(OSDI), p. 137C150
10. Deelman E, Singh G, Su M, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman G, Good J et al (2005) Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci Progr 13(3):219–237
11. Fei X, Lu S (2012) A dataflow-based scientific workflow composition framework. Serv Comput IEEE Trans 5(1):45–58. doi:10.1109/TSC.2010.58
12. Fei X, Lu S, Lin C (2009) A mapreduce-enabled scientific workflow composition framework. In: IEEE International Conference on Web Services, 2009. ICWS 2009, IEEE, pp. 663–670
13. Group K. Opencl (open computing language) - the open standard for parallel programming of heterogeneous systems. In: URL http://www.khronos.org/opencl/
14. Johnson D, Garey M (1979) Computers and intractability: a guide to the theory of np-completeness. Freeman&Co, San Francisco
15. Lander G, Stagg S, Voss N, Cheng A, Fellmann D, Pulokas J, Yoshioka C, Irving C, Mulder A, Lau P et al (2009) Appion: an integrated, database-driven pipeline to facilitate em image processing. J Struct Biol 166(1):95–102
16. Lin C, Lu S, Lai Z, Chebotko A, Fei X, Hua J, Fotouhi F (2008) Service-oriented architecture for view: A visual scientific workflow management system. In: IEEE International Conference on Services Computing, 2008. SCC'08. IEEE, vol. 1, pp. 335–342
17. Ludäscher B, Weske M, Mcphillips T, Bowers S (2009) Scientific workflows: business as usual? Business process management pp. 31–47
18. McPhillips T, Bowers S, Zinn D, Ludäscher B (2009) Scientific workflow design for mere mortals. Futur Gener Comput Syst 25(5):541–551
19. Nguyen P, Halem M (2011) A mapreduce workflow system for architecting scientific data intensive applications. In: Proceeding of the 2nd international workshop on Software engineering for cloud computing, ACM, pp. 57–63
20. Oinn T Greenwood M, e.a. (2005) Taverna:lessons in creating a workflow environment for the life sciences. pp. 1067–1100
21. Pireddu L, Leo S, Zanetti G (2011). Mapreducing a genomic sequencing workflow. In: Proceedings of the second international workshop on MapReduce and its applications, ACM, pp. 67–74
22. Polo J, Carrera D, Becerra Y, Beltran V, Torres J, Ayguadé E (2010) Performance management of accelerated mapreduce workloads in heterogeneous clusters. In: 39th International Conference on Parallel Processing (ICPP2010)

23. Polo J, Carrera D, Becerra Y, Steinder M, Whalley I (2010) Performance-driven task co-scheduling for mapreduce environments. In: Network Operations and Management Symposium (NOMS), 2010 IEEE, pp. 373–380
24. Rooijers K, Kolmeder C, Juste C, Doré J, de Been M, Boeren S, Galan P, Beauvallet C, de Vos W, Schaap P (2011) An iterative workflow for mining the human intestinal metaproteome. BMC Genomics 12(1):6
25. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) 2010, pp. 1–10
26. Topcuouglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
27. Wang J, Crawl D, Altintas I (2009) Kepler+ hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, ACM, p. 12
28. Warr W (2012) Scientific workflow systems: Pipeline pilot and knime. Journal of computer-aided molecular design pp. 1–4
29. White T (2012) Hadoop: The definitive guide. O'Reilly Media
30. Wolf J, Rajan D, Hildrum K, Khandekar R, Kumar V, Parekh S, Wu K, Balmin A (2010) Flex: a slot allocation scheduling optimizer for mapreduce workloads. Middleware 2010:1–20
31. Jacob JC, Katz DS et. al (2004) The Montage architecture for gridenabled science processing of large, distributed datasets. In: Proceedings of the Earth Science Technology Conference, June 2004