International Journal of Innovative
Computing, Information and Control
Volume **14**, Number **6**, December **2018**

ICIC International ©2018 ISSN 1349-4198
pp. 2151–2161

BGSCHEDULER: BIPARTITE GRAPHIC MODEL BASED SCHEDULING FOR CLOUD COMPUTING TASKS

Yaqiong Li[1], Bo Zhou[1], Yongbo Liu[1], Shouchao Li[1] and Yunkui Song[2,*]

[1]Jiangsu Hoperun Software Company
No. 168, Ruanjian Ave., Yuhuatai Dist., Nanjing 210012, P. R. China

[2]Institute of Software, Chinese Academy of Sciences
4# South Fourth Street, Zhongguancun, Beijing 100190, P. R. China
*Corresponding author: songyk@otcaix.iscas.ac.cn

Received January 2018; revised May 2018

Abstract. *Scheduling a large scale of concurrent jobs is important for managing cloud computing systems. Existing works often use queue-based models to achieve a single goal, which cannot adapt to various applications and changing scenarios. To address this issue, we propose a novel scheduling framework-BGScheduler for concurrent tasks in cloud computing, which flexibly schedules tasks for various optimized goals to deal with changing requirements of applications. First, we use a bipartite graph model to present tasks' requirements and physical resources. Second, we choose fairness, constraints and priority as goals to match requirements and resources with a minimum cost maximum flow algorithm. Third, we propose an incremental optimization method to speed up the procedure of online solving a bipartite graph. Finally, we have implemented BGScheduler, and the experimental results demonstrate that BGScheduler has well flexibility to achieve various scheduling goals in a real scenario, and has decreased the scheduling delay in a simulated scenario.*

Keywords: Task scheduling, Bipartite graphic model, Performance optimization, Cloud computing

1. **Introduction.** In recent years, cloud computing has been widely used to provide Internet-based services with shared resources in various industrial fields [19,20]. A cloud computing system with thousands of physical machines processes a large scale of submitted concurrent jobs separated into many tasks. Allocating physical resources to concurrent tasks is an important issue for efficiently managing cloud computing systems [21]. Existing works scheduling tasks can be categorized as fairness based approaches constraint based approaches and priority based approaches as follows.

Fairness based scheduling approaches: Delay-schedule [10] based on a queue based model schedules tasks if resources can meet requirements, and waits for timeout if resources cannot meet requirements. Sparrow [11] based on sampling incrementally adjusts its scheduling strategy according to runtime status. Mesos [12] is a resource scheduling framework for heterogeneous tasks, which uses a two-stage queue-based model to provision tasks with their expected resources. Quincy [8] transforms the issue of provisioning resources for tasks to solve a problem of minimum cost maximum flow.

Constraint based scheduling approaches: Alsched [2] using a queue based model defines the utility function of allocating resources to tasks, and makes a scheduling decision based on the calculated utility. Tetrisched [13] which is the extension of Alsched mines the correlations between tasks according to runtime monitoring data to calculate the overall

DOI: 10.24507/ijicic.14.06.2151

utility. Bistro [14] based on a queue-based model designs an online and offline scheduling framework with a two-stage scheduling and reconfiguration mechanism. Paragon [15] uses a multi-queue model based method to categorize heterogeneous resources and applications, analyzes their different resource requirements, and then calculates their utilities automatically.

Priority based scheduling approaches: Borg [3] with a queue-based model sorts tasks and rearranges tasks according to their priorities. Omega [16] extends Borg in solving the runtime interference between tasks. Quasar [17] categorizes tasks and resources, constructs a model to evaluate the matching between tasks and resources, and then allocates resources to target tasks. [18] for data intensive scenarios uses DAG (Directed Acyclic Graph) models to characterize correlations between tasks, and schedules tasks on nodes with less network delay.

The above approaches always aim at achieving a fixed scheduling goal regardless of the requirements of various applications and changing scenarios. They always can achieve a single goal, which cannot adapt to various applications and changing scenarios in cloud computing. For example, e-commerce applications ought to process the tasks of recommending commodities for a priority goal, but process the tasks of calculating commodities for a fairness goal. To address the above issue, we propose a novel scheduling framework – BGScheduler for concurrent tasks in cloud computing, which considers the changing requirements of applications, and schedules tasks for an optimized goal. First, we use a bipartite graph to present tasks' requirements and physical resources. Second, we choose fairness, priority and constraints as goals to match requirements and resources with a minimum cost maximum flow algorithm. Third, we propose an incremental optimization method to speed up the procedure of online solving the bipartite graph. The contributions of this paper are as follows.

(1) We propose a bipartite graph to model tasks' requirements and physical resources, which can conveniently adapt to deployment environments by adjusting graphic structure.
(2) We match requirements and resources with a minimum cost maximum flow algorithm, which can achieve changing goals by adjusting the definition of edge weights (e.g., fairness, priority or constraints) in the graph.
(3) We propose an incremental optimization method to solve the bipartite graph, which can self-adapt to online changing application scenarios with limited computation complexity.
(4) We have implemented a prototype system – BGScheduler, and conducted extensive experiments in real scenarios to validate the proposed approach by comparing it with existing ones.

The remainder of this paper is organized as follows. Section 2 gives a problem statement and presents the overview of our approach. Section 3 presents the construction and solution of a bipartite graph. Section 4 validates the proposed approach with experiments in a real scenario and a simulated scenario. Section 5 concludes this paper and directs the future work.

2. **Scheduling Graph Construction.**

2.1. **Problem statement.**

We state the problem of scheduling tasks as matching the resource requirements of tasks with the available physical resources. Then, we define the problem as follows:

$$\boldsymbol{T} \times \boldsymbol{R} \rightarrow \tau^z,$$

where $\boldsymbol{T} = \{T_{1,1}, T_{1,2}, \ldots, T_{i,t}, T_{i,t+1}, \ldots, T_{M,N}\}$ is the task set, $T_{i,j}$ $(1 \leq i \leq M,\ 1 \leq t \leq N)$ is the jth task in the ith job, M is the job set size, N is the task set size; $\boldsymbol{R} = \{R_1, R_2, \ldots, R_S\}$ is the available physical resource set.

Existing approaches [10-12] scheduling concurrent tasks often use queue-based models with an FIFS (First Input First Service) strategy, so they cannot achieve multiple goals, which usually get a local optimal result. To address the above issue, we use a bipartite graph to characterize requirements and resources, and then calculate the global optimal results, which achieve the weight of minimum "cost" and maximum "flow" in the graph edges. As shown in Figure 1, we define the bipartite graph as:

$$G = (V, E, U, C),$$

where V is the node set providing physical resources; E is the edge set mapping tasks to resources; U is the capacities of edges; C is the cost of mapping operations; T in the left side is the task set; M in the right side is the resource set; X_i, R_i and M_i present a cluster, a rack and a resource to process tasks, respectively; $T_{1,2} \rightarrow M_1$ and $T_{1,2} \rightarrow U_1$ are edges, where M_1 and U_1 are the candidate resources for processing task $T_{1,2}$; the capacities and costs of edges present whether tasks can be allocated with resources.

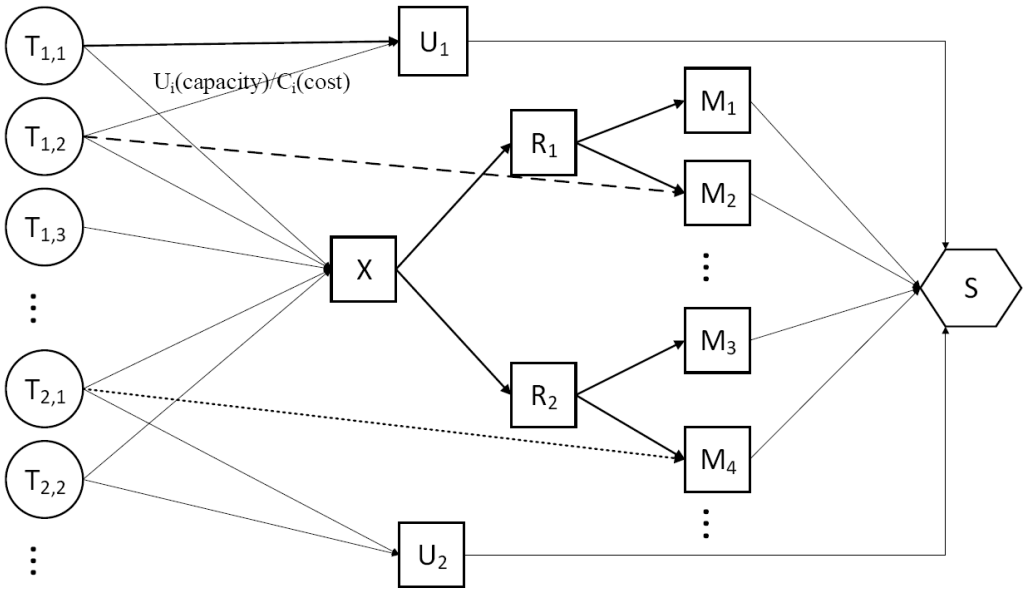

FIGURE 1. Bipartite graph model for scheduling tasks

We map scheduling strategies to the bipartite graph by adjusting the graphic struct, and adapt the bipartite graph to changing requirements of applications by adjusting the definition of edge weights.

2.2. **Scheduling strategies.** Existing scheduling frameworks [2,3,8] always can achieve only one goal, for example, Quincy supports fairness, Alsched supports constraint and Borg supports priority. However, our framework BGScheduler can support all the above three scheduling goals. This subsection introduces how to implement them to demonstrate the flexibility of BGScheduler, which can support various scheduling goals.

2.2.1. *Fairness.* Fairness is one of the important goals for scheduling jobs, which means that jobs share available resources equally using a fairness based algorithm (e.g., first come first serve, round robin). For example, Job_j with N_j tasks has A_j resource units calculated by a fairness based algorithm. We present fairness with the capacity of an edge

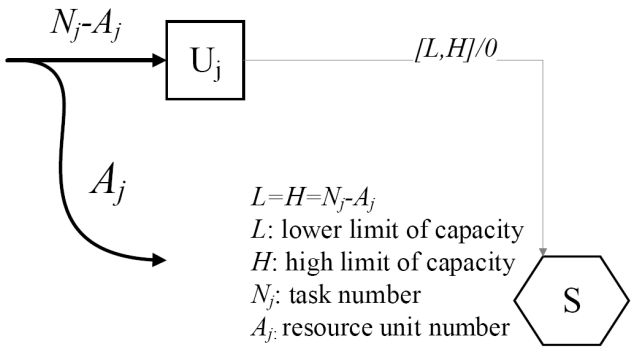

FIGURE 2. Graph construction for fairness

as shown in Figure 2. We set the capacity of edge $U_j \to S$ as $N_j - A_j$, and then the flow of the edge is $f_u = N_j - A_j$, which means that the number of waiting tasks in Job_j is $N_j - A_j$. According to the minimum cost maximum flow algorithm [4], Job_j can get A_j resource units.

We take an example to describe the fairness strategy. We want to assign Job_1 with three tasks $T_{1,1}$-$T_{1,3}$ and Job_2 with four tasks $T_{2,1}$-$T_{2,4}$ to four hosts. Each host processes one job, and each job requires two resource units. Thus, we set the capacity (i.e., the maximum number of waiting jobs) of edge $U_1 \to S$ as 1, and the capacity of edge $U_2 \to S$ as 2. Then, Job_1 has a waiting task and two running tasks on two hosts, and Job_2 has two waiting tasks and two running tasks on two hosts. Using the above setting, we can use BGScheduler to fairly assign each job with two hosts.

2.2.2. *Constraint.* The placement constraint rule can be described as: ⟨task, resources, utility⟩, where task represents the task to be placed in a node; resources represent the required resources for the task; utility represents the obtained utility brought from placing the task on the resources. The problem of placement constraint can be mapped to the issue of constructing an edge from task to resources (i.e., *task* → *resources*). We set the cost of the edge as (-utility), and then the maximum utility corresponds to the minimum cost.

As shown in Figure 3, we take an example to describe the constraint strategy. A task T_1 for image analysis requires GPU. A host M_1 has GPU, but a host M_2 does not have GPU. Then, we get utility as one, if T_1 is placed on M_1, but we get utility as zero, if T_1 is placed on M_0. We can construct the graph by presenting the above placement constraint, and then solve the graph to get maximum flow (i.e., minimum cost).

2.2.3. *Priority.* The tasks with a high priority can first obtain resources. The cost of allocating resource unit v composed of various physical resources (e.g., memory utilization,

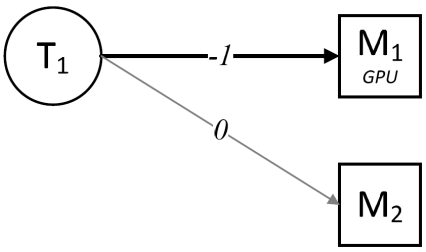

FIGURE 3. Graph construction for placement constraint

and CPU utilization) to task w is defined as:

$$cost(w,v) = [w_1, w_2, \ldots, w_n] \times \begin{bmatrix} memory \\ CPU \\ \vdots \\ d_n \end{bmatrix},$$

where w_i $(1 \leq i \leq n)$ presents the weight of the ith physical resource.

As shown in Figure 4, we take an example to describe the priority strategy. We set the priorities of three tasks T_1, T_2 and T_3 as one, two and five. Then, we calculate the cost as α_1, α_2, α_3 $(\alpha_1 < \alpha_2 < \alpha_3)$ according to the cost calculation formula. We can construct the graph by presenting the above costs, and then assign the tasks with priorities (e.g., T_3) by solving the constructed graph.

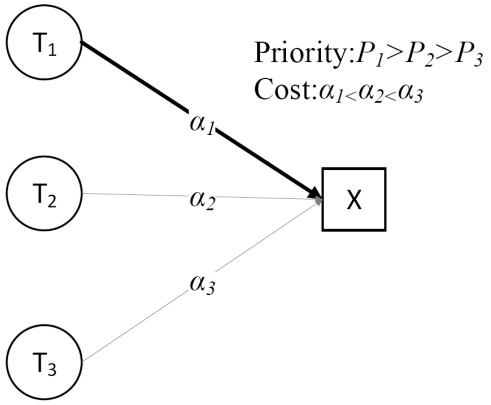

FIGURE 4. Graph construction for priority

3. **Scheduling Graph Solution.** We can solve the problem of minimum cost max flow as follows.

Target function:

$$\min \sum_{(w,v)\in E} c(w,v) f(w,v),$$

Constraint conditions:

$$0 \leq f(w,v) \leq u(w,v) \quad \forall (w,v) \in E,$$

$$\sum_{(w,v)\in E} f(w,v) - \sum_{(v,w)\in E} f(v,w) = b(v) \quad \forall u \in V,$$

where w and v present the node of task requirements and the node of physical resources in the graph; $c(w,v)$ and $f(w,v)$ present the cost and flow of edges when allocating task w to resource v; $u(w,v)$ is the capacity of edge (w,v).

We solve the problem by finding a minimum cost with iterations in the constraint of a maximum flow, or finding a maximum flow with iterations in the constraint of a minimum cost. Existing approaches (e.g., cost scaling [4], and network simplex [5]) of solving the problem often have a high computation complexity and a few changes in the graphic structure, so they cannot adapt to dynamic application scenarios. Thus, we propose an incremental approach of solving the graph, which speeds up the calculation by caching temporary results and reuse them in each iteration. As shown in Algorithm 1, we adjust the local graphic structure of the constructed graph in the constraint of a minimum cost to increase the flow, and incrementally find an optimum graph with iterations.

Algorithm 1: Incremental Graph Solution

Input: Graph $G(V, E, U, C)$
Output: Minimum cost flow f of graph G

1. Procedure Incremental *CostScaling*(G)
2. Initiate f = *CostScaling*(G)
3. While detecting a Event do
4. Update graph G according to Event
5. f' = *CostScaling*(G, f)
6. Execute scheduling
7. $f = f'$
8. End While
9. End
10. Procedure *CostScaling*(G, f)
11. $\epsilon = C$, $p(i) = 0\ \forall i \in V$, $f' = f$
12. While $\epsilon \geq \frac{1}{n}$ do
13. (ϵ, f, p) = *refine*(ϵ, f', p)
14. End While
15. return f'
16. End
17. Procedure refine(ϵ, f, p)
18. $\epsilon = \epsilon/\alpha$
19. $\forall (v, w) \in E$, ***if*** $c_p(v, w) < 0$ ***then*** $f'(v, w) = u(v, w)$
20. While existing a push or a relabel operation that applies
21. Select such an operation and apply it
22. End While
23. return (ϵ, f', p)
24. End

Algorithm 1 detects the events (e.g., task submission, and task failure) of global changes in the graph with an event driven mechanism (lines 3-7); updates the graphic structure with the results of graphic solution in the last iteration, when some events occur (10-16); updates relaxation conditions with iterations (lines 17-24). The computation complexity is decided by the scale of changes in the graph, which is $O(VE)$ in the best condition, and is $O(V^2E\log(VE))$ in the worst condition, where V is the number of nodes, and E is the number of edges.

4. Evaluation.

4.1. Experimental environment.

As shown in Figure 5, the experiments include a real scenario to validate that BGScheduler can achieve various scheduling goals, and a simulated scenario to evaluate the scheduling delay of BGScheduler.

Real scenario: the experimental environment includes two racks; each rack has ten hosts; each host has 24 cores, 2.4GHz Intel Xeon CPU and 24G DDR3 memory, CentOS 7 and Docker 1.0; each host in the first rack has a GPU; racks and clusters deploy Gigabit Ethernet routers.

Simulated scenario: we use a Google's open dataset [6] which records the historical monitoring data of Google's clusters including ten thousand physical nodes, each of which is configured with 24 cores, 64G, memory and 10Gbps networks.

We evaluate BGScheduler with resource utilization, the average time of processing tasks, the average time of waiting tasks, and system normalized performance (SNP).

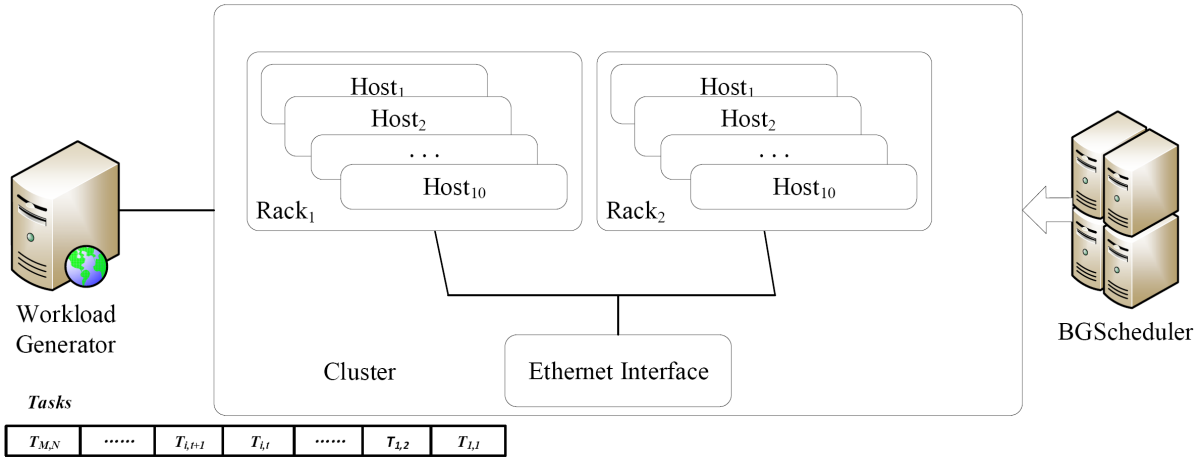

FIGURE 5. Experimental testbed

We use batch workloads and interactive workloads to evaluate our approach. For batch workloads, we use CPU sensitive MergeSort jobs, memory sensitive PageRank jobs, network sensitive TPC-H jobs, and GPU sensitive jobs image analysis. For interactive workloads, we use HTTP service Httpd, SQL database MySQL, and distributed cache Redis operations.

4.2. **Scheduling validation.** We compare BGScheduler with well-known open source scheduling framework Firmament [7] implementing Quincy supporting fairness, Swarm [8] implementing Alsched supporting constraint, and Kubernetes [9] implementing Borg supporting priority. Each of the above three frameworks supports only one scheduling strategy. The experiments in this subsection aim at validating that BGScheduler can support all the above three scheduling strategies flexibly.

4.2.1. *Fairness.* We compare BGScheduler with Firmament supporting fairness and a typical random strategy using batch, interactive and mix workloads. As shown in Figure 6, the SNP of BGScheduler is similar to that of Firmament, and twice than that of Random. The experimental results demonstrate that BGScheduler can well support the fairness-based scheduling strategy.

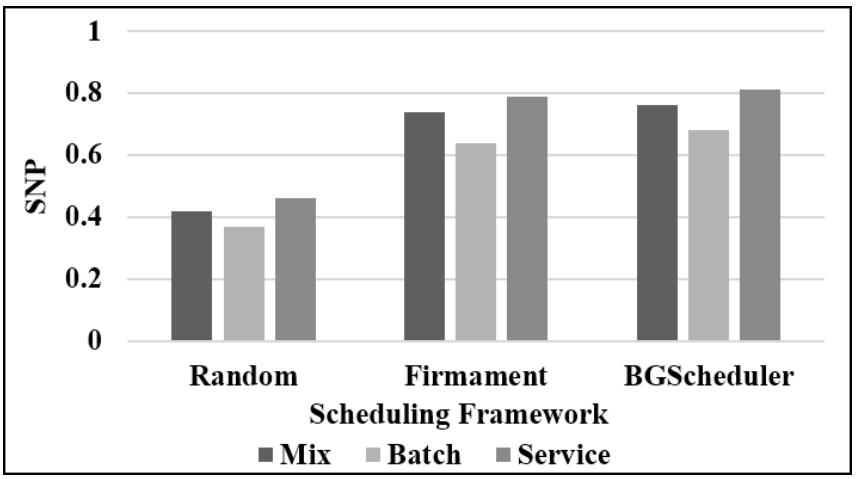

FIGURE 6. SNP of different task schedulers

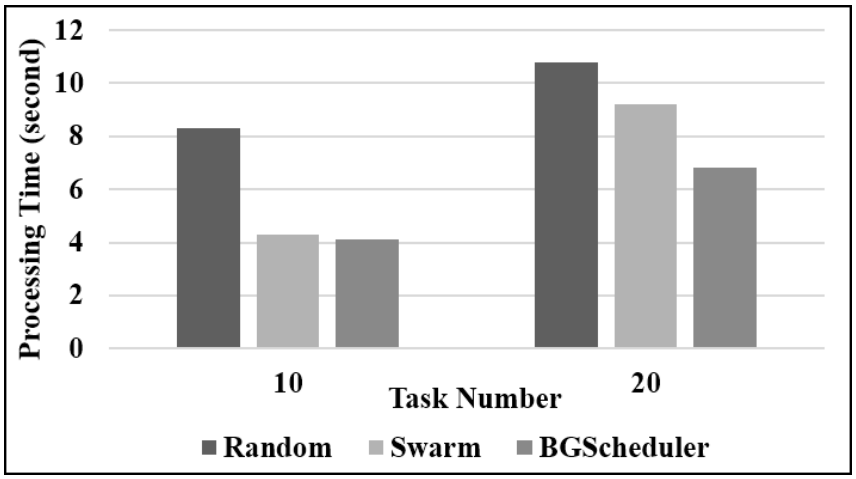

FIGURE 7. Average scheduling times of different schedulers

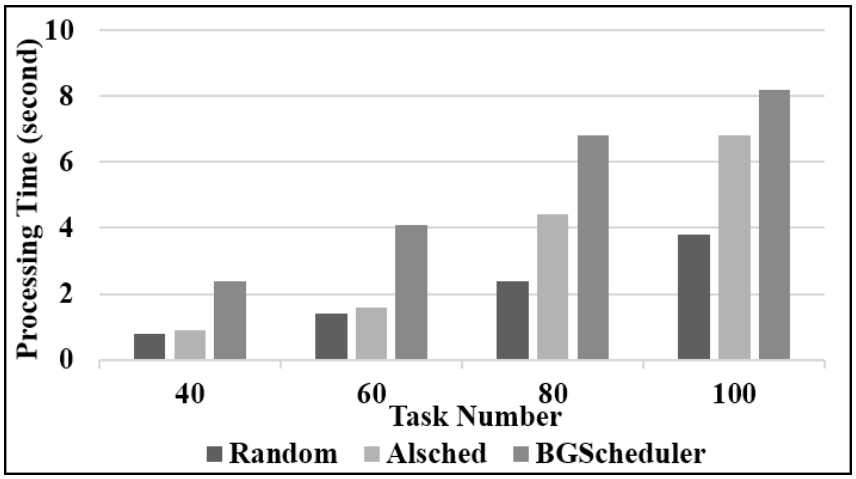

FIGURE 8. Average processing times of different schedulers

4.2.2. *Constraint.* We compare BGScheduler with Swarm supporting constraint and a typical random strategy in the average time of processing tasks using image analysis workloads. We set the number of tasks as ten and twenty, respectively. As shown in Figure 7, the average processing time of BGScheduler is similar to that of Swarm, and twice than that of Random, when the task number is ten; the average processing time of BGScheduler is the best, when the task number is twenty. As shown in Figure 8, the average processing time of BGScheduler is the least, when the task number increases. The experimental results demonstrate that BGScheduler can well support the constraint scheduling strategy, and is better than Swarm in performance.

4.2.3. *Priority.* We compare BGScheduler with Kubernetes supporting priority and a typical random strategy in the average time of processing tasks using three batch workloads.

As shown in Figure 9, the delays of TPC-H tasks with the same priority are more than five seconds, because short tasks are hungry when long tasks occupy resources. As shown in Figure 10, the delays of TPC-H tasks with a higher priority are less than one second. The experimental results demonstrate that BGScheduler, which is similar to Kubernetes in performance, can well support the constraint scheduling strategy.

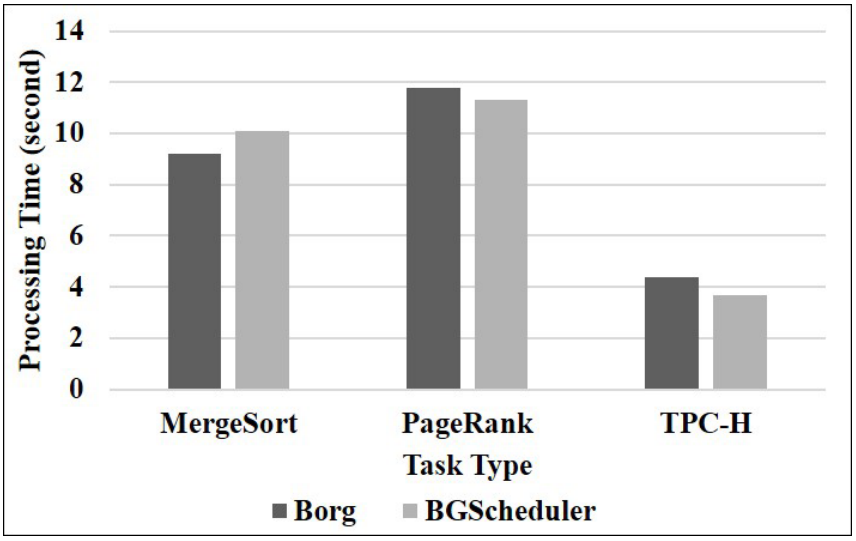

FIGURE 9. Average scheduling times of different schedulers in the same priority

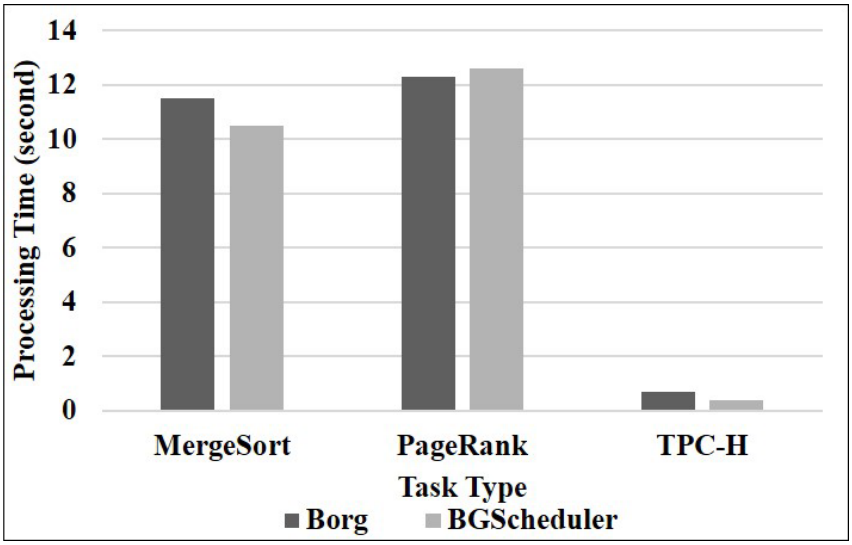

FIGURE 10. Average scheduling times of different schedulers in different priorities

4.3. **Performance evaluation.** We use a Google's open dataset, and use two scenarios with five thousand and ten thousand machines, respectively. We compare BGScheduler with a queue based Random algorithm and a Cost Scaling (CS) algorithm in scheduling delay. As shown in Figure 11 and Figure 12, the scheduling delays of three algorithms increase with the number of tasks in a near linearity. The delay of CS is much more than that of BGScheduler and that of Random, which increase rapidly with the number of tasks. The delay of BGScheduler is the least in the scenario with ten thousand nodes, which has the best performance. The results can be explained that our proposed method can online incrementally solve the bipartite graph with much less computation complexity.

5. **Conclusions.** This paper proposes a novel scheduling framework – BGScheduler for concurrent tasks in cloud computing, which considers the changing requirements of applications, and schedules tasks for an optimized goal. First, we use a bipartite graph to present tasks' requirements and physical resources. Second, we choose fairness, priority

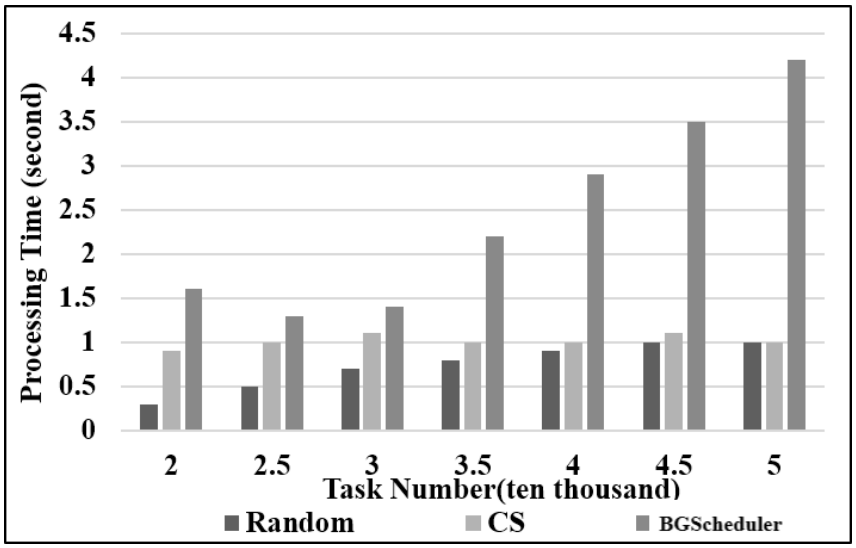

FIGURE 11. Processing times of different schedulers in five thousand simulated nodes

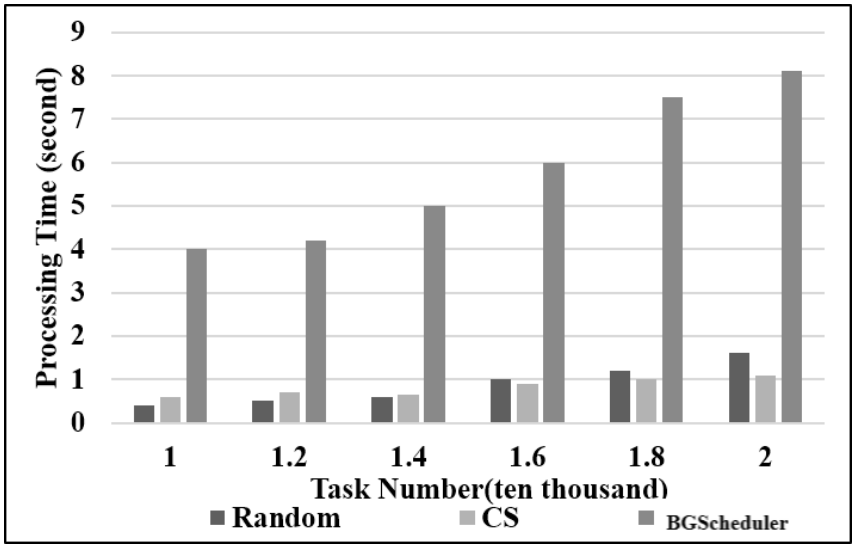

FIGURE 12. Processing times of different schedulers in ten thousand simulated nodes

and constraints as goals to match requirements and resources with a minimum cost maximum flow algorithm. Third, we propose an incremental optimization method to speed up the procedure of solving the bipartite graph. Finally, the experimental results demonstrate that BGScheduler has well flexibility to achieve various scheduling goals in a real scenario, and has efficiently decreased the scheduling delay in a simulated scenario.

In our future work, we plan to improve BGScheduler as follows. Currently we manually set parameters according to domain knowledge, but unsuitable parameters will affect the results. We plan to study automatic method to choose suitable parameters by learning from historical records. Furthermore, solving graphs requires high computation complexity, so we plan to speed up the procedure by filtering and merging redundant data.

Acknowledgment. This work is supported by Nanjing high-end talent team introduction project (No. 10072090) in China. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar and A. Goldberg, Quincy: Fair scheduling for distributed computing clusters, *Proc. of the 22nd ACM Symp. on Operating Systems Principles*, 2009.

[2] V. A. Tumano, J. Cipar, M. A. Kozuch and G. R. Ganger, Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds, *Proc. of the 3rd ACM Symp. on Cloud Computing*, 2012.

[3] A. Verma, L. Pedrosa, M. Kompolu, D. Oppenheimer, E. Tune and J. Wilkes, Large-scale cluster management at Google with Borg, *Prof. of the 10th European Conf. on Computer Systems*, 2015.

[4] A. V. Goldberg, An efficient implementation of a scaling minimumcost now algorithm, *Journal of Algorithms*, vol.22, no.1, pp.1-29, 1997.

[5] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[6] B. V. Cherkassky and A. V. Goldberg, On implementing the push-relabel method for the maximum flow problem, *Algorithmica*, vol.19, no.4, pp.390-410, 1997.

[7] C. Rejss, J. Wilkes and J. L. Hellerstein, Google cluster usage traces: Format schema, *Technical Report*, Google Inc., http://codegoogIecom/p/googleclusterdata/wiki/TraceVersion2, 2011.

[8] Firmament, *Firmament Quincy Scheduler*, http://firmament.io, 2015.

[9] Docker Swarm, *Docker Swarm Filters*, http://githubcom/docker/swam, 2014.

[10] M. Zaharia, D. Borthakur, S. J. Sarma, K. Elmeleegy, S. Shenker and I. Stoica, Delay scheduling: A simple technique for achieving locality and firmness in cluster scheduling, *Proc. of the 5th European Conf. on Computer Systems*, pp.265-278, 2010.

[11] K. Ousterhout, P. Wendell, M. Zaharia and I. Stoica, Sparrow: Distributed low latency scheduling, *Proc. of the 24th ACM Symp. on Operating Systems Principles*, pp.69-84, 2013.

[12] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker and I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, *NSDI 11*, pp.22-37, 2011.

[13] A. Tumanov, T. Zhu, M. A. Kozuch, M. Harchol-Balter and G. R. Ganger, Tetrisched: Space-time scheduling for heterogeneous datacenters, *Technical Report CMU-PDL-13-112*, Carnegie Mellon University, 2013.

[14] A. Goder, A. Spiridonov and Y. Wang, Bistro: Scheduling data-parallel jobs against live production systems, *Proc. of the 2015 USENIX Annual Technical Conf.*, pp.459-471, 2015.

[15] C. Delimitrou and C. Kozyrakis, Paragon: QoSAware scheduling for heterogeneous datacenters, *ACM SIGPLAN Notices*, vol.48, no.4, pp.77-88, 2013.

[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek and J. Wilkes, Omega: Flexible scalable schedulers for large compute clusters, *Proc. of the 8th European Conf. on Computer Syscems*, pp.351-364, 2013.

[17] C. Delimitrou and C. Kozrrakis, Quasar: Resource efficient and QoS aware cluster management, *ACM SIGPLAN Notices*, vol.49, no.4, pp.127-144, 2014.

[18] S. Venkataraman, A. Panda, G. Ananthanamyanan, M. J. Franklin and I. Stoica, The power of choice in data-aware cluster scheduling, *Proc of the 11th USENIX Symp. on Operating Systems Design and Implementation*, pp.301-316, 2014.

[19] J. A. dos Santos, J. P. C. Henriques, R. P. Mesquita, A. B. Lugli and M. M. D. Santos, Industrial supervisory system using cloud computing, *International Journal of Innovative Computing, Information and Control*, vol.13, no.1, pp.75-84, 2017.

[20] S. Yin, J. Liu, Y. Zhang and L. Teng, Cuckoo search algorithm based on mobile cloud model, *International Journal of Innovative Computing, Information and Control*, vol.12, no.6, pp.1809-1819, 2016.

[21] B. Wang, S. Jin and B. Qin, Batch arrival based performance evaluation of a VM scheduling strategy in cloud computing, *International Journal of Innovative Computing, Information and Control*, vol.14, no.2, pp.455-467, 2018.