IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received August 4, 2019, accepted August 16, 2019, date of publication August 23, 2019, date of current version September 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936820

Effective Parallel Computing via a Free Stale Synchronous Parallel Strategy

HANG SHI[1], YUE ZHAO[1], BOFENG ZHANG[1], KENJI YOSHIGOE[2], (Senior Member, IEEE), AND FURONG CHANG[1,3]

[1]School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China
[2]Faculty of Information Networking for Innovation and Design (INIAD), Toyo University, Tokyo 112-8606, Japan
[3]School of Computer Science and Technology, Kashi University, Xinjiang 844008, China

Corresponding author: Yue Zhao (yxzhao@shu.edu.cn)

This work was supported by the Xinjiang Natural Science Foundation under Grant 2016D01B010.

ABSTRACT As the data becomes bigger and more complex, people tend to process it in a distributed system implemented on clusters. Due to the power consumption, cost, and differentiated price-performance, the clusters are evolving into the system with heterogeneous hardware leading to the performance difference among the nodes. Even in a homogeneous cluster, the performance of the nodes is different due to the resource competition and the communication cost. Some nodes with poor performance will drag down the efficiency of the whole system. Existing parallel computing strategies such as bulk synchronous parallel strategy and stale synchronous parallel strategy are not well suited to this problem. To address it, we proposed a free stale synchronous parallel (FSSP) strategy to free the system from the negative impact of those nodes. FSSP is improved from stale synchronous parallel (SSP) strategy, which can effectively and accurately figure out the slow nodes and eliminate the negative effects of those nodes. We validated the performance of the FSSP strategy by using some classical machine learning algorithms and datasets. Our experimental results demonstrated that FSSP was 1.5-12× faster than the bulk synchronous parallel strategy and stale synchronous parallel strategy, and it used 4× fewer iterations than the asynchronous parallel strategy to converge.

INDEX TERMS Straggler, parallel strategy, parallel programming.

I. INTRODUCTION

The massive volumes of historical data that already exists and the sharp data generation speed have brought us more intractable challenges. The sheer volume of data has led to the interest in parallel computing and distributed systems. The leading example is the MapReduce based systems [1]. This kind of systems works on low-cost unreliable commodity hardware, and it is an extremely scalable cluster consist of redundant array of independent nodes [2]. However, the inefficiency in iterative calculations of MapReduce makes it no longer suitable for many current applications [3], [4].

With the development of distributed computing, many new solutions have emerged, such as GraphX [5], Hama [6], and so on [7]–[9]. These frameworks can show their strengths in most applications. However, due to the limitations of parallel computing strategies, they are not satisfactory in some cases. For example, the bulk synchronous parallel (BSP) strategy based frameworks, such as Hama, require all nodes to complete the tasks of current stage before proceeding to the next stage. The strict synchronous limitation of BSP induces the problem that several stragglers will drag down the efficiency of the whole system. In order to break the shortcoming of the strict synchronous restrictions and accelerate the computing speed, an asynchronous parallel (ASP) strategy has been proposed [10]-[12]. This kind of strategy eliminates the negative influence of the stragglers in BSP strategy by allowing each node to work without synchronous restriction and to use local parameters. The independent local computing makes it easy for each node to fall into a local optimal solution instead of a global optimal solution. Hence, the ASP strategy cannot guarantee calculation convergence. Ho *et al.* [13] proposes a stale synchronous parallel (SSP) strategy, which uses a staleness threshold to control the synchronization of work process. As long as the process gap between the fastest node and the slowest node is within the threshold (e.g., the number

The associate editor coordinating the review of this article and approving it for publication was Wenbing Zhao.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

of iterations of each node in an iterative work), the system switches to the asynchronous parallel computing. When the gap exceeds this threshold, the fastest node will wait for the slowest node to catch up. The SSP strategy is considered as a compromise between asynchronous parallel strategy and synchronous parallel strategy. Nevertheless, the efficiency of the SSP based system is still largely impacted by the stragglers [14]–[16].

Stragglers prevail in both heterogeneous and homogeneous systems. Due to the differentiated price-performance, cost, and energy consumption balance [17]–[21], there are often multiple small clusters combined into a big cluster as a whole. The heterogeneous environment caused by software configurations and hardware diversity leads to the different performance of nodes, and results in the speeds of task completion to be unequal [22]–[24]. The faster nodes need to wait for stragglers to complete their tasks before they can show the final result. The waiting process is a waste of computing resources. In a distributed environment, the communication latency, overheads, and bandwidth may change the load balance, the synchronization behavior, and the resource contention [25]-[28]. Even in a homogeneous environment, the node calculation speed will be different due to resource competition and load imbalance. The performance differences will affect the overall performance of the system.

The ASP strategy and the SSP strategy may be effective with the nodes caused by temporary slowdowns, but their performance are not satisfactory with stragglers maintain low efficiency during the work entirely. The proposed solutions work efficiently in MapReduce systems, but they are not applicable for other systems. Thus, we proposed a free stale synchronous parallel (FSSP) strategy to address this problem.

Based on SSP, we used a *penalty times* mechanism to distinguish the stragglers. The node in FSSP will record the number of times when it is too slow to make the fastest node to pause, we called it *slow times*. Once the *slow times* reach the preset *penalty times*, the slow node will be identified as a straggler and will be excluded from the subsequent process. After the exclusion, the system will be free from the impact of the stragglers. The job undertaken by the punished node will be borne by the other nodes.

In our previous work [29], we implemented FSSP on the parameter server [30], [31] and compared it with aforementioned BSP strategy, ASP strategy and SSP strategy on some classic algorithm models and data sets. The experimental results demonstrated that FSSP outperformed BSP, ASP, and SSP strategies altogether. However, our previous implementation did not simulate the real situation well, and was fragile in large scale computing situation. In our previous experiments, we set only one straggler, and the actual application situation may exist several stragglers. The job copy mechanism of our previous work was efficient in a small-scale computing task, but inefficient in large-scale tasks. As the scale increases, the problem of excessive memory usage will decrease the performance of the system. Therefore, we make several main additional contributions in this work

1) Explore the impact of the stragglers on the overall performance of the system and list the experimental results.
2) Expand the scale of stragglers to simulate the real situation.
3) Change the job copy mechanism from neighbor nodes to the fast node.

The rest of this paper is organized as follows. In Section2, we introduce the related works. In section3, we evaluate the influence of stragglers by several experiments. In Section4, we describe the free stale synchronous parallel (FSSP) strategy. In Section5, we evaluate FSSP and compare it with other strategies. In Section 6, we finally sum up our contributions.

II. RELATED WORK

A. COUNTERMEASURE OF THE STRAGGLERS IN MAPREDUCE

As an epoch-making pioneering theoretical technology, MapReduce has greatly promoted the development of big data, and has spawned many big data processing frameworks. The most famous one is Hadoop implemented by Yahoo. Nevertheless, the development of MapReduce and Hadoop has exposed lots of shortcomings. For example, the defects of MapReduce limit the efficiency of the performance of Hadoop on loop operations, and the programming paradigm of MapReduce limits the implementation of computations that cannot be converted into this paradigm. Zaharia *et al.* [32] points out that some of the implicit assumptions of the scheduler of Hadoop make Hadoop inefficient in environments with stragglers. For example, Hadoop implicitly assumes that nodes perform work at roughly the same rate, and the tasks progress at a constant rate throughout time. However, in the actual application environment, the work performance of nodes is generally not static. The nodes will exhibit different working efficiencies with different workloads. When the load is larger, the efficiency will be relatively lower. The scheduler of Hadoop treats the time cost for each phase as the same. For example, in a reduce task, the execution is divided into three phases, each of which accounts for 1/3 of the total time. It leads to the inaccurate judgment of the stragglers. MapReduce systems are overly idealized for homogeneous operating environments, and they perform inefficiently in heterogeneous environments. The aforementioned Hadoop, as well as Twister [33] and ST-Hadoop [34], do not address this problem satisfactorily.

To address this problem, Zaharia *et al.* [32] proposes the Longest Approximate Time to End (LATE) algorithm. This algorithm relies on the speculative execution mechanism of Hadoop. When a node is identified as a straggler in a task, MapReduce runs a speculative copy of its task on another machine to finish the computation faster. Hadoop uses a progress score between 0 and 1 to determine whether a node is a straggler. In this scoring mechanism, the proportion of each stage to the total time spent is the same, and it is scored by the

current stage of the node. When the progress score of a task is less than the average for its category minus 0.2, and the task has run for at least one minute, it is marked as a straggler. The LATE algorithm relies on this scoring mechanism to predict the longest possible execution time of each node, and always perform speculative execution on the nodes that may run the longest. These tasks, which belong to the predicted stragglers, are fulfilled by other nodes, thus reducing the impact of these slow nodes on the overall system. However, the LATE algorithm also has some shortcomings. Its forecasting mechanism relies on the Hadoop scoring mechanism, which is an overly idealized static mechanism. This static scoring mechanism is not robust against the various emergencies that may occur in actual operation.

Since the LATE algorithm still needs many improvements under dynamic conditions, Fadika *et al.* [22] designs a new MapReduce based system MARLA (MApReduce with adaptive Load balancing for heterogeneous and Load imbalAnced clusters) to deal with dynamic heterogeneous environments. In MARLA, the traditional task distribution mechanism of the existing MapReduce system is replaced by active requests of the work nodes. The master node registers the total number of tasks available to the nodes, and the work nodes are afforded a processing identification tag, which is used to request tasks. When the work node completes its own task, it can request a new one from the master node, so that the slow node will process fewer tasks accordingly. In that case, the impact of the stragglers on the overall system becomes less serious. Ahmad *et al.* [23] also designs a new MapReduce system Tarazu to address this problem. They analyze the reason for the performance differences in the workflows of MapReduce in detail, and the most important reasons includes the interaction between loads balancing, network traffic in Map and the reduce phase imbalance amplified by heterogeneity. Tarazu improves traditional workflows and performs more efficiently in the condition with stragglers.

Although these aforementioned solutions performed efficiently in the condition with stragglers, they are over-reliant on the mechanism of the MapReduce and not applicable for other systems [35]–[37].

B. EXISTING PARALLEL STRATEGIES

1) BSP STRATEGY

The most widely used parallel strategy is bulk synchronous parallel (BSP) strategy, which requires strict synchronous restriction. In BSP, it required all nodes to complete their jobs of current clock before entering the next clock. Nevertheless, the different performance of the nodes leads to the awkward situation that some stragglers drag down the overall efficiency, and this problem is more prominent in heterogeneous environments [22]. Another problem is that the strict synchronization requirements cause massive communication between nodes leading to large communication overheads. The time cost by communication may be a lot more than the calculation time [32], [38].

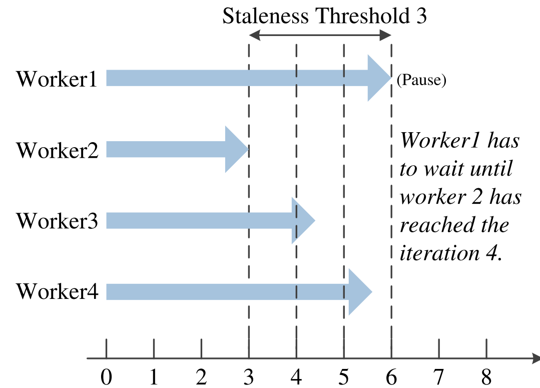

FIGURE 1. The SSP strategy.

2) ASP STRATEGY

In order to accelerate the computing speed and reduce the communication overheads in BSP, some improved BSP solutions have been proposed, such as [39], [40], but they are not sufficiently satisfactory. Thus, more loose strategies than BSP have been proposed, such as [10], [41], [42]. Take DistBelief [10] as an example. DistBelief includes a kind of asynchronous parallel (ASP) strategy called Downpour SGD, which allows each node updates part of the overall parameters. At the same time, in order to enable the nodes to complete their respective calculations, each node copies the entire model to local before calculation and updates the parameters asynchronously. This kind of strategy greatly increases the computing speed and reduces the communication overheads to some extent, but the independent local computing results in the calculated model to be a local optimal result not the global optimal result. Thus, in ASP, convergence cannot be guaranteed.

3) SSP STRATEGY

Taking into account the convergence guarantee of BSP and the calculation speed of ASP, Qirong Ho proposes a stale synchronous parallel (SSP) strategy [13], which is also widely used in applications [43]–[45]. This strategy contains a synchronous staleness threshold, and the fastest node cannot exceed the slowest one more than the predefined staleness threshold. Before the nodes reach this limit, they work asynchronously. Whenever the work process gap between the fastest node and the slowest node reaches the limit of stale threshold, the fastest node will pause and wait for the slowest one to catch up before entering the next work process, like what is shown in Figure 1.

In their work [13], the experimental results demonstrate that the larger the staleness threshold value is, the less time is used for communication consumption, and the closer to the asynchronous parallel strategy. The existence of synchronous boundary guarantees the convergence of SSP. Zhang *et al.* [46] proposes a method to dynamically adjust this staleness threshold in the calculation task. However, whether it is the original fixed synchronous boundary or the dynamic one, the stragglers will still drag down the efficiency of the

whole system, especially with the existence of the stragglers mentioned above. In order to address the problem that the slow nodes drag the system, we proposed the free stale synchronous parallel (FSSP) strategy.

III. THE STRAGGLERS

In this section, we discuss the causes of stragglers and demonstrate the effects of these stragglers through experiments.

A. THE CAUSE OF STRAGGLERS

The two main reasons for the performance difference between the nodes are heterogeneity and competition. Many practical applications are now built on virtual digital centers or cloud servers. For example, the New York Times rented 100 virtual machines for a day to convert 11 million scanned articles to PDFs [32]. These virtual machines may run on different physical hosts, and their performance differences are subject to aging and configuration. Some physical hosts will be in different computer rooms that are geographically far apart, which not only brings hardware differences, but also huge communication overheads.

Even in a cluster, virtual machines also compete for hard disk and network bandwidth. With clusters such as those established on Amazon Elastic Compute Cloud, their nodes may be distributed on different physical hosts, and the virtual machines of other users are also running on top of these physical hosts. Since the clusters of different users generally operate different tasks, the competition intensity of resources between the cluster of the same users and the different users are generally different. Differences in the level of competition lead to different resources available and performance differences.

Of course, in addition to these two main factors, there are many situations that can cause different node performance, such as system failures, program errors, load problems, and so on [47]–[49]. Since there are so many factors that cause stragglers, some reasonable methods are required to deal with this problem.

B. THE IMPACT OF STRAGGLERS

Jiang *et al.* [14] explore the extent of impact of stragglers on BSP systems, ASP systems, and SSP systems. In their experiments, they activate the *sleep()* function in 20% nodes to simulate the environment with stragglers, and they increase the sleep time to increase heterogeneity. Their experimental results show that when stragglers exist, the performance of the three types of systems decreases correspondingly. In the BSP system, the overall performance of the system is greatly affected by stragglers due to strict synchronization restrictions. When the running time of stragglers is twice that of other nodes, the time for the system to complete tasks is doubled correspondingly. In ASP system and SSP system, the calculation speed of stragglers is slow, which makes the speed of parameter updating different. The overall parameters do not conform to the objective function and the performance of the system is also decreased.

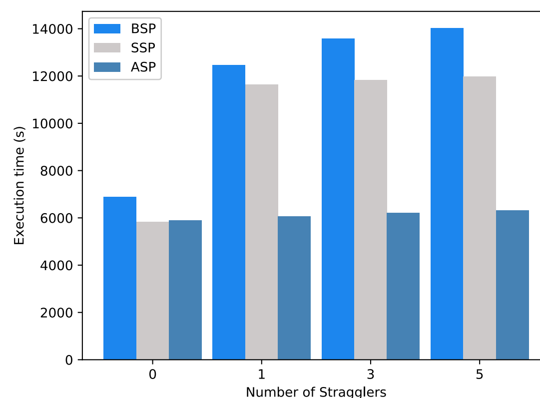

FIGURE 2. Performance of existing parallel strategies with different scale of stragglers.

In order to explore the impact of the stragglers more adequately, we supplemented the impact of different numbers of stragglers on the performance of the three types of systems. In this section, the dataset we used is Netflix prize data, and the benchmark algorithm is matrix decomposition. The experiment was implemented on MPI, which used 64 processors to perform the task. The specific hardware environment was given in Section 5. We refered to the experiment in [14], and artificially set some stragglers through the *sleep()* function. The experimental results were demonstrated in Figure 2.

1) RESULT

As shown in Figure 2, when the scale of stragglers increases, the execution time of the task using the three parallel strategies increases correspondingly. Thus the performance decreases. When execution time of the stragglers is set to be $2\times$ more of the normal node duration, the execution time under the BSP and SSP strategies is roughly $2\times$ more than before. And the execution time slightly increases when the scale of the stragglers becomes larger. The execution time under the ASP strategy has increased correspondingly with the increase in the scale of stragglers, but the change in execution time is not as obvious as BSP and SSP.

2) DISCUSSION

Due to the synchronization restrictions, when running under BSP and SSP strategies, the execution time will be seriously dragged when stragglers occurs, because synchronization requires all threads to keep their progress consistent or within a certain range. The slow computing speed of stragglers will drag down the efficiency of others. With the scale of stragglers increasing, the impact of the running speed is not as severe as the time when the first straggler occurs, for the synchronization overheads is more related to the execution time of the stragglers rather than the amount of the stragglers.

When running under the ASP strategy, the overall runtime did not change drastically because there is no synchronization limit. Nevertheless, the appearance of stragglers affects the convergence of the overall model, so that the model cannot meet the objective function, and the overall execution time is also extended. As the number of stragglers increases,

this problem becomes more severe and the corresponding execution time costs more.

IV. THE FREE STALE SYNCHRONOUS PARALLEL STRATEGY

A. DESIGN OF THE FSSP STRATEGY

As the amount of data grows larger and the calculations become more complicated, a cluster of multiple machines is generally used to accomplish the calculation task. Assume that we want to train a machine learning model with multiple parameters *P = {p1, p2,..., pn}*. In an iteratively optimized model, the parameter *P* is updated by iterations, and the objective function is used to evaluate the model. If the target requirement is not met, then the next iteration is performed to update the parameters, until the entire model gradually converges to the object. In the iterative process, each node or thread follows a certain parallel strategy to control the progress of the task. The whole process is represented by algorithm 1.

Algorithm 1 Typical Parallel Machine Learning Function

```
1:  Init_Model(P);
2:  for i = 0 to Max_iteration do
3:      Synchronization_Strategy();
4:      Read_Data(data);
5:      Param_Update(P, data);
6:      if Object_Func(P) <= Object then
7:          Break;
8:      end if
9:  end for
10: return Model(P).
```

As the existing parallel strategies perform inefficiently with the presence of the stragglers, we designed the free stale synchronous parallel (FSSP) strategy to distinguish the stragglers and reduce their damage to overall performance. FSSP contains two part of aspects: the mechanism of straggler detecting, and the mechanism of computing integrity protection.

1) STRAGGLER DETECTIVE MECHANISM OF FSSP

In this section, some important names and their definitions are listed below:

slow times: The times when a node is too slow to make the fastest node pause in SSP.

penalty times: This is a preset factor as the condition to judge the stragglers and trigger the punishment. When the *slow times* of one node reach the *penalty times*, it will be identified as a straggler.

SSP uses the *staleness threshold* to control the overall working process, and we used this mechanism to detect the stragglers. We found that a temporary slowdown is normal in actual application due to the work load and network delay, and performance fluctuation will lead to the misjudgement of the stragglers [50]. To avoid the misjudgement, we used the *penalty times* to control the strictness of evaluation criteria.

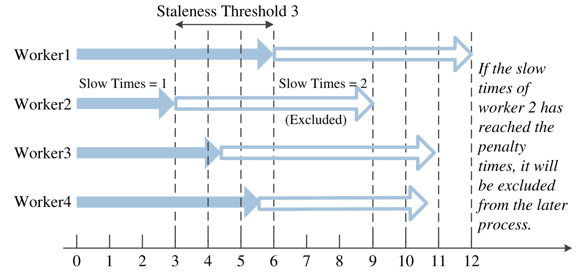

FIGURE 3. Straggler detective mechanism of the FSSP strategy.

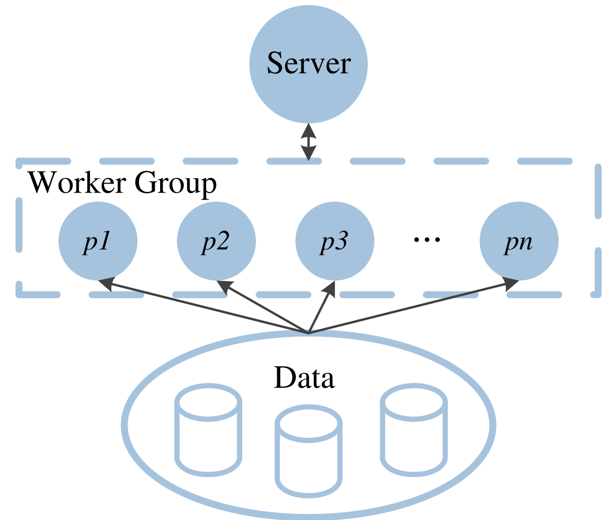

FIGURE 4. Traditional distributed task executing scenario.

A proper *penalty times* will improve the accuracy of straggler judgement.

In Figure 3, if we set the *penalty times* are 2, when the *slow times* of worker 2 reach the *penalty times*, worker 2 will be excluded from the subsequent calculation process.

2) COMPUTING INTEGRITY PROTECTION MECHANISM OF FSSP

Nevertheless, in the process of distributed computing tasks, each node is responsible for updating part of the parameters. If one node is stopped, this part of parameters will not be updated. Thus, a computing integrity protection mechanism is needed after stragglers being excluded.

In a traditional distributed task, the parameters updating scenario is like Figure 4. For the parameters *P={p1, p2,..., pn}*, the update process in worker group *W={w1, w2,..., wn}* is as follows:

$$p_i^{(t+1)} = p_i^{(t)} + \Delta(w_i, p_i^{(t)}) \tag{1}$$

In that case, once a machine stops working, some parameters will not be updated, and the model will not meet the requirement.

In our previous work [29], we copy the job from one node to its neighbour node like Figure 5 at the beginning of the task. In that case, once a node is excluded from the worker group, its job will not lost and finished by the neighbouring node.

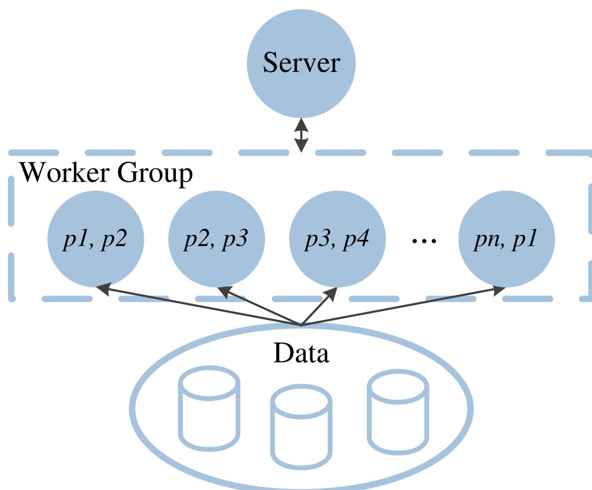

FIGURE 5. Job copy on neighboring node.

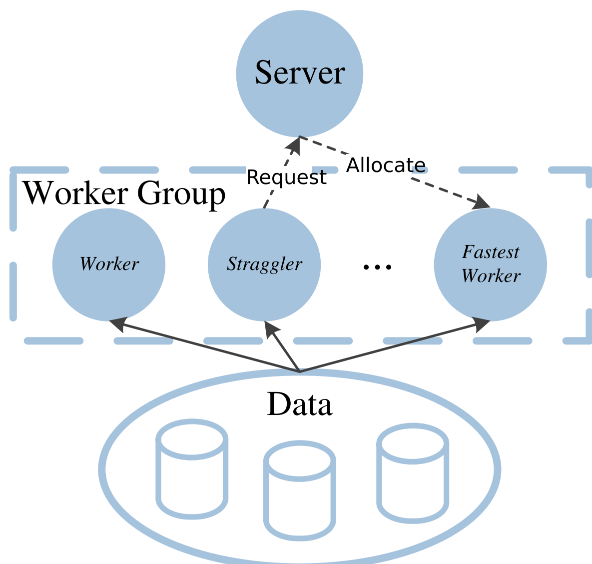

FIGURE 6. Job copy on the fastest node.

Although the neighbor-copy is efficient in a small scale task, it occupies too much memory, and the neighbouring nodes may not be the best choice for job copy. Thus, when the task scale becomes larger, the former implementation becomes more inefficiently than other parallel strategies.

To improve the performance under larger scale task, we changed the job copy mechanism from neighbouring node to the fastest node. SSP maintains the work process (more explicitly, their iterations) of the work nodes on server node to control their process synchronization, and FSSP also use the process information to judge the straggler. Once a node is identified as a straggler and be excluded from the worker group, its job will be copied to the fastest node according to their work process maintained by the server node. If the fastest node has been occupied, the job of the straggler will be copied to another unoccupied fastest node.

The overall flow of FSSP is represented by algorithm 2 and 3.

Algorithm 2 Parallel Machine Learning Function in FSSP on Workers

1: *Init_Model*(P);
2: $slow_times = 0$;
3: $penalty_times = n$;
// Set penalty_times.
4: **for** $i = 0$ to $Max_iteration$ **do**
5: *SSP_Control*(&*slow_times*);
// If the node reaches the threshold, its slow_times will increase.
6: **if** $slow_times >= penalty_times$ **then**
7: *Request_Job_Copy*();
// Send job copy request to server node, server node will assign the job.
8: *Stop*();
// When slow_times reaches the penalty_times, this node/process will be excluded from the subsequent process.
9: **else**
10: **if** *Check_Occupied*() **then**
11: *Allocation_Receive*();
// If this node is occupied, it will copy the job from the straggler according to the allocation.
12: **end if**
13: *Read_Data*(*data*);
14: *Param_Update*(P, *data*);
15: **if** *Object_Func*(P) $<=$ *Object* **then**
16: *Break*;
17: **end if**
18: **end if**
19: **end for**
20: **return** *Model*(P).

Algorithm 3 Job Copy Allocation on Server

1: *job_copy_request* = *Receive_Job_Copy_Request*();
// Receive the job copy request from the straggler.
2: *iteration_info* = *Gather_Iteration_Info*();
// Gather the iteration information of all the nodes.
3: *target* = *Find_Unoccupied_Worker*(*iteration_info*);
// Find the fastest unoccupied node.
4: *Occupy_Worker*(*target*);
// Occupy the target node.
5: *Allocation_Request*(*target*, *job_copy_request*);
// Allocate the job copy from the request node to the target node.

B. THEORETICAL ANALYSIS OF THE FSSP STRATEGY

In a distributed computing task, assume that we want to train a model M with an initial state of M_0, and node w is one of the nodes affected by the aforementioned stragglers in a SSP system. It can read the global model as the formula below in iteration i according to [13].

$$M_i = M_0 + U1_{0\sim(i-s)} + U2_{(i-s)\sim i} + U3_{i\sim(i+s-1)} \quad (2)$$

In (2), U represents for the update operation, and s denotes the *stale threshold*. This equation consists of three major parts:

1) $U1_{0\sim(i-s)}$ represents for the update accomplished by all nodes before iteration *(i-s)*. In the SSP strategy, this part can be read by all the nodes.
2) $U2_{(i-s)\sim i}$ represents for the update finished by node w from iteration *(i-s)* to *i*.
3) $U3_{i\sim(i+s-1)}$ represents for the update operation of all the nodes except node w range from iteration *(i-s)* to *(i+s-1)*. This part of update from other nodes can be read by node w because of the *stale threshold*.

Due to the synchronous restriction, node w will wait for the stragglers several times. In FSSP, the work is different before the time, when the slowest node being excluded, from the work after this time point. Suppose that the average time to complete an iteration for node w is c_w.

1) BEFORE THE NODE BEING EXCLUDED

In this period, the FSSP strategy runs in the SSP way. We use t_{wait} to denote the time node w costs on waiting. At time t, the number of iterations finished by node w is:

$$iter_w = (t - t_{wait})/c_w \tag{3}$$

And the update operations finished by node w are:

$$U_w = \sum_{i=0}^{iter_w} u_{(w,i)} \tag{4}$$

2) AFTER THE NODE BEING EXCLUDED

The cluster has been free from the impact of the slowest node since the exclusion time. t'_{wait} denotes the time node w costs on waiting. At time t, the number of iterations node w has finished is:

$$iter'_w = (t - t'_{wait})/c_w \tag{5}$$

And the update node finished by w is:

$$U'_w = \sum_{i=0}^{iter'_w} u_{(w,i)} \tag{6}$$

The stragglers has been excluded from the subsequent process, hence the time cost on waiting in this period is less than the same work in SSP. In that case, we get the formula as follows:

$$t_{wait} > t'_{wait} \tag{7}$$
$$iter_w > iter'_w \tag{8}$$
$$U_w \in U'_w \tag{9}$$

Compared with the SSP strategy, the FSSP strategy frees the fast nodes from the impact of the stragglers, and lets them spend more time on computing rather than waiting. Hence, the nodes accomplished more iterations than that in SSP strategy at the same time.

TABLE 1. The experiment datasets.

Name	Description	Rows/Nodes	Columns/Edges
Netflix prize data	Netflix movie evaluation dataset	480,189	17,770
ego-Twitter	Twitter data was crawled from public sources.	81,306	1,768,149
LiveJournal social network	LiveJournal online social network	4,847,571	68,993,773

Assume that we want to train a model M with an initial state of M_0 under the FSSP strategy. For node w, it can read the model as the formula below at time t:

$$M_t = M_0 + U1'_{0\sim(i-s)} + U2'_{(i-s)\sim i} + U3'_{i\sim(i+s-1)} \tag{10}$$

According to the inference mentioned above, the fast nodes spend more time on computing and accomplish more iterations than those in the SSP strategy. Hence, by comparing formula 2 with formula 1, we get the conclusions as follows:

$$U_1 \in U'_1 \tag{11}$$

In period U'_2 and U'_3, the nodes are working on the later iteration than the period in U_2 and U_3.

Therefore, compared with the SSP strategy, the FSSP strategy reduces the overheads on synchronization and takes full advantage of the computing resource.

V. EVALUATION

In this section, we evaluated the effectiveness of the FSSP strategy against the stragglers, and compared it with existing parallel strategies. The experimental results demonstrated that the FSSP strategy outperformed the other parallel strategies.

A. EXPERIMENTAL SETUP

1) EXPERIMENT ENVIRONMENT

The computing platform is consist of 140 IBM PureFlex x240 blade nodes. Each node is configured with 64GB of RAM and two 2.9GHz Intel E5-2690 CPUs with 8 cores, running on Ubuntu 16.04. It is a public computing platform carrying many computing tasks from different users. The resource competition among nodes is unpredictable and makes the experimental environment more closer to the reality.

2) BENCHMARKING APPLICATION AND DATASET

We chose matrix factorization and PageRank as the benchmarking applications to evaluate the FSSP strategy. The information of the datasets are shown in Table 1. For matrix factorization, we used the method of stochastic gradient descent to find the optimal model by iterations. And we used full batch of ego-Twitter and Netflix dataset. For PageRank, we implemented it by the method of power iteration, and used 10% minibatch on LiveJournal social network per iteration.

TABLE 2. The information of the experiment groups.

Group	Benchmark Application	Dataset	Processors	Stragglers	Slow Degree of The Stragglers
1	Matrix Factorization	ego-Twitter	64	6	1.5x
2	Matrix Factorization	Netflix prize data	32	3	1.5x
3	PageRank	LiveJournal social network	64	6	2.0x
4	PageRank	LiveJournal social network	32	3	2.0x

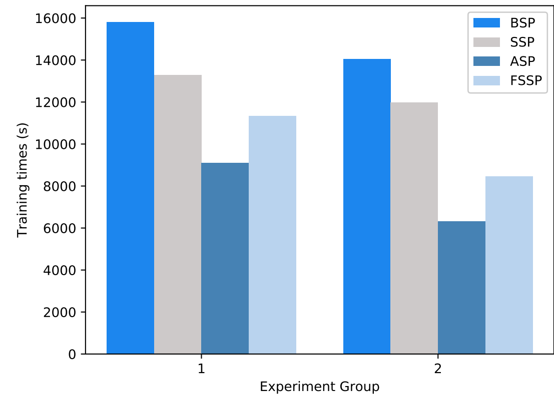

FIGURE 7. The comparison of the execution time on matrix factorization (experiment group 1 and 2).

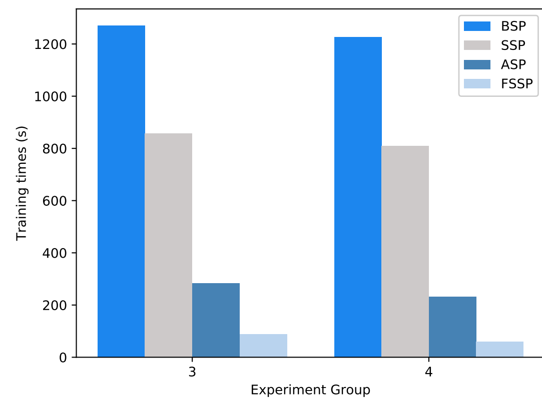

FIGURE 8. The comparison of the execution time on pagerank (experiment group 3 and 4).

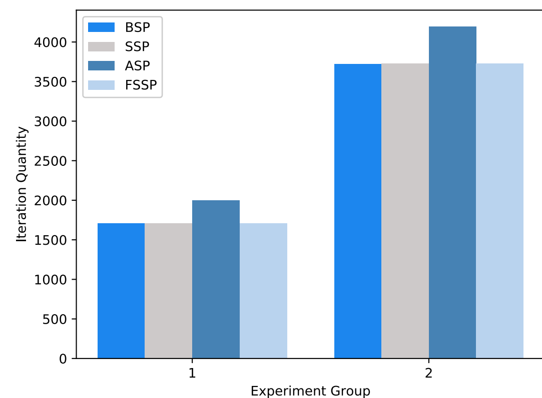

FIGURE 9. The comparison of the iteration quantity on matrix factorization (experiment group 1 and 2).

3) EXPERIMENT DESIGN

We implemented the benchmark applications by MPI on the aforementioned computing platform. The unpredictable resource competition may lead to the differential performance of the processors, but it cannot guarantee the appearance of the stragglers. Following [14], we random picked 5% of the processors to be stragglers by using *sleep()* function, and we chose one of the processors to be the root processor to maintain the iteration information of all the processors. Every processor compare its work process with the overall work process to evaluate its performance. When it is too slow to make the fastest node to pause, its *slow times* will increase until reaching the *penalty times*. When the processor reaching the *penalty times*, it will send a request to the root processor and prepare to be excluded. The root processor will appoint an unoccupied fastest processor to inherit the job of the straggler.

We implemented four experiment groups to evaluate the performance of the FSSP strategy.

B. EXPERIMENT RESULT AND DISCUSSION

1) EXECUTION TIME

In figure 7 and 8, we compared the execution time of all the parallel strategies. The experimental result demonstrated that the ASP strategy and the FSSP strategy finished the task faster than the BSP strategy and the SSP strategy in all of the experiment groups. The fast processors in ASP did not have the synchronous overheads as the processors in BSP and SSP. The FSSP strategy reduced the critical synchronous overheads caused by the stragglers through its straggler exclusion, and free the system from the impact from the stragglers. The performance of ASP and FSSP was different in figure 7 and 8. Relating to figure 10, in experiment group 3 and 4, the ASP strategy cannot converge the loss and finished the work until reaching the max iteration, thus the performance of ASP was unstable. Nevertheless, the FSSP strategy was designed on the basis of the SSP strategy, thus the parallel synchronization happened throughout the entire task. The parallel synchronization made the performance of FSSP to be stable.

2) ITERATION QUANTITY AND QUANTITY

In figure 9 and 10, we compared the iteration quantity (lower the better) when the task finishing of all the parallel strategies. Figure 9 and 10 revealed that the iteration quantity of the ASP strategy was the most among all of the strategies due to its poor convergence performance. The ASP strategy cannot guarantee the convergence of the object, thus ASP need more iterations of calculations to finish

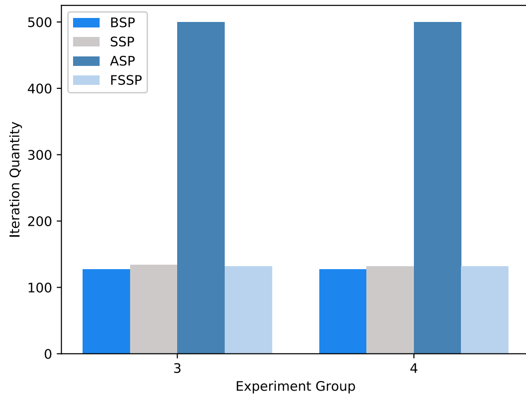

FIGURE 10. The comparison of the iteration quantity on pagerank (experiment group 3 and 4).

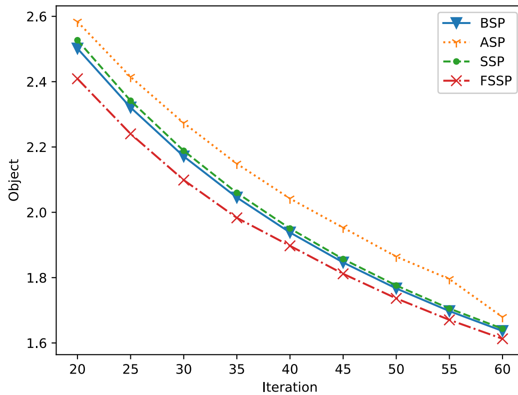

FIGURE 11. The comparison of the convergence on iteration quantity of experiment group 1.

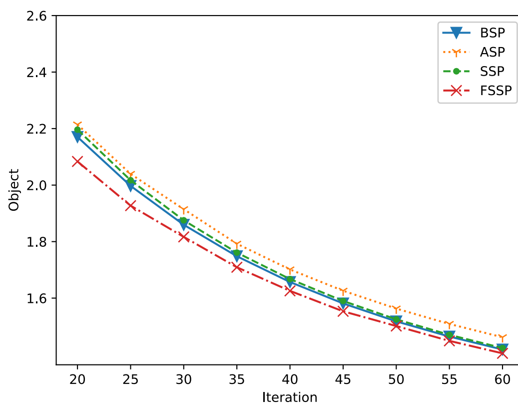

FIGURE 12. The comparison of the convergence on iteration quantity of experiment group 2.

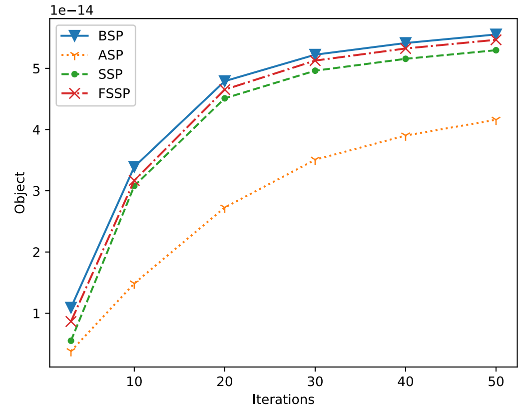

FIGURE 13. The comparison of the convergence on iteration quantity of experiment group 3.

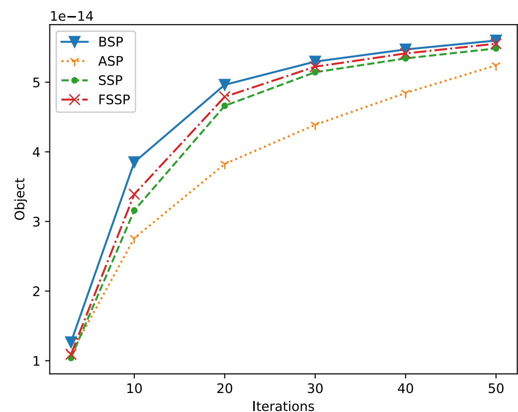

FIGURE 14. The comparison of the convergence on iteration quantity of experiment group 4.

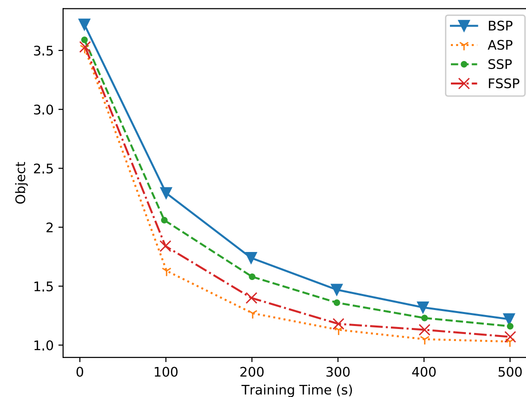

FIGURE 15. The comparison of the convergence on execution time of experiment group 1.

the task. Figure 11-14 demonstrated that the convergence quality of the ASP strategy was unsatisfactory in detail. The strategies with synchronization restriction converged better than the ASP strategy. The FSSP strategy inherited the synchronous mechanism of the SSP strategy, thus the convergence quality of FSSP was much more satisfactory than ASP. And the straggler exclusion reduced the negative effect of the stragglers and speeded up the computing, thus iteration quality of the FSSP strategy was better than the BSP strategy and the SSP strategy.

3) CONVERGENCE PERFORMANCE

Figure 11-18 demonstrated that performance comparison of the convergence of all the strategies. In figure 11-14, from the iteration dimension, the ASP strategy performed worst, and the strategies with synchronization restriction performed better. The convergence performance of ASP was the worst, because it increased the computational speed at the expense of convergence guarantee. From the time dimension, in figure 15 and 16, the ASP strategy converged

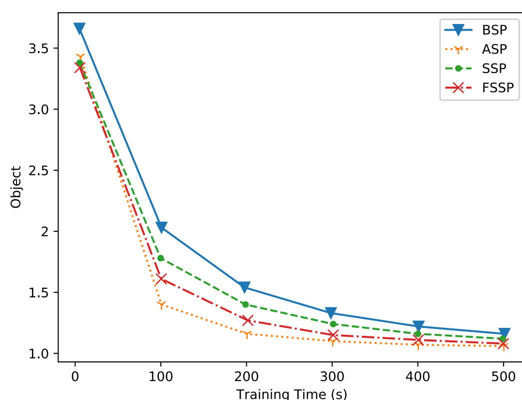

FIGURE 16. The comparison of the convergence on execution time of experiment group 2.

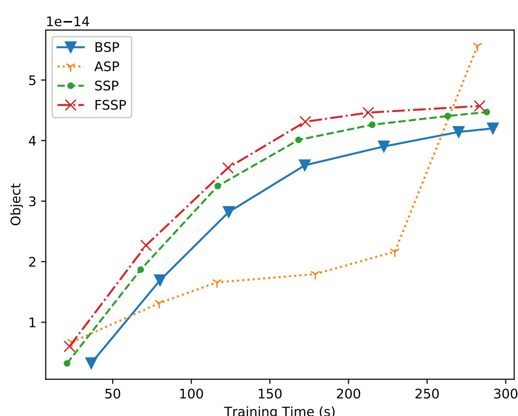

FIGURE 17. The comparison of the convergence on execution time of experiment group 3.

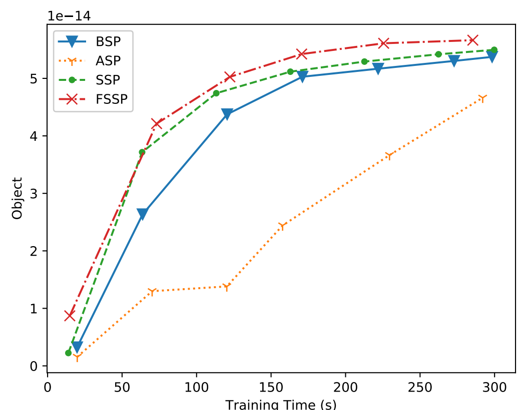

FIGURE 18. The comparison of the convergence on execution time of experiment group 4.

fastest due to its independent computing mode and no synchronization restriction, but its performance was the worst in figure 17 and 18. Figure 15-18 demonstrated that a strategy without synchronization restriction like ASP was fragile on a task optimized by iteration. The FSSP strategy reduced the negative effect of the stragglers but still keep the synchronization mechanism, thus FSSP converged faster than BSP and SSP. And the synchronization mechanism inherited from SSP made the convergence quality of FSSP much better than ASP. FSSP both take care of the convergence quality and computing speed, thus the comprehensive performance was the best among the strategies.

VI. CONCLUSION

The existing parallel strategies work well in normal scenario, but they are inefficient with the existence of the stragglers. To address this problem, we proposed the free stale synchronous parallel (FSSP) strategy. Based on the SSP strategy, the FSSP strategy combines the *staleness threshold* with the *penalty times* mechanism to detect the stragglers, and excludes the stragglers from the subsequent work. The job copy mechanism of FSSP can protect the integrity of the task after the stragglers being excluded. The experimental results demonstrated that, with the existence of the stragglers, the FSSP strategy was 1.5-12× faster than the bulk synchronous parallel strategy and stale synchronous parallel strategy, and it used 4× fewer iterations than the asynchronous parallel strategy to converge.

ACKNOWLEDGMENT

The authors would like to thank the High Performance Computing Center of the School of Computer Science of Shanghai University for their support for this work.

REFERENCES

[1] I. A. T. Hashem, N. B. Anuar, M. Marjani, E. Ahmed, H. Chiroma, A. Firdaus, M. T. Abdullah, F. Alotaibi, W. K. M. Ali, I. Yaqoob, and A. Gani, "MapReduce scheduling algorithms: A review," *J. Supercomput.*, pp. 1–31, Dec. 2018. doi: 10.1007/s11227-018-2719-5.

[2] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Comput*, vol. 46, no. 1, Oct. 2013, Art. no. 11. doi: 10.1145/2522968.2522979.

[3] R. Patgiri, "Taxonomy of big data: A survey," Aug. 2018, *arXiv:1808.08474*. [Online]. Available: https://arxiv.org/abs/1808.08474

[4] S. Liu, D.-G. Zhang, X.-H. Liu, T. Zhang, J.-X. Gao, C.-L. Gong, and Y.-Y. Cui, "Dynamic analysis for the average shortest path length of mobile ad hoc networks under random failure scenarios," *IEEE Access*, vol. 7, pp. 21343–21358, 2019. doi: 10.1109/ACCESS.2019.2896699.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. OSDI*, Berkeley, CA, USA, 2014, pp. 599–613.

[6] K. Siddique, Z. Akhtar, Y. Kim, Y.-S. Jeong, and E. J. Yoon, "Investigating apache Hama: A bulk synchronous parallel computing framework," *J. Supercomput.*, vol. 73, no. 9, pp. 4190–4205, Sep. 2017.

[7] D.-G. Zhang, "A new approach and system for attentive mobile learning based on seamless migration," *Appl. Intell.*, vol. 36, no. 1, pp. 75–89, Jan. 2012.

[8] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "LightGraph: Lighten communication in distributed graph-parallel processing," in *Proc. IEEE Int. Congr. Big Data*, Jun./Jul. 2014, pp. 717–724.

[9] Y. Zhao, K. Yoshigoe, M. Xie, J. Bian, and K. Xiong, "L-PowerGraph: A lightweight distributed graph-parallel communication mechanism," *J. Supercomput.*, pp. 1–30, Apr. 2018. doi: 10.1007/s11227-018-2359-9.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Proc. NIPS*, Lake Tahoe, NV, USA, 2012, pp. 1223–1231.

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, New York, NY, USA, 2016, pp. 1928–1937.

[12] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, Berkeley, CA, USA, 2016, pp. 265–283.

[13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, ''More effective distributed ML via a stale synchronous parallel parameter server,'' in *Proc. NIPS*, Lake Tahoe, NV, USA, 2013, pp. 1223–1231.

[14] J. Jiang, B. Cui, C. Zhang, and L. Yu, ''Heterogeneity-aware distributed parameter servers,'' in *Proc. SIGMOD*, New York, NY, USA, 2017, pp. 463–478.

[15] D.-G. Zhang, H.-L. Niu, and S. Liu, ''Novel PEECR-based clustering routing approach,'' *Soft Comput.*, vol. 21, no. 24, pp. 7313–7323, Dec. 2017. doi: 10.1007/s00500-016-2270-3.

[16] D.-G. Zhang, J.-X. Gao, X.-H. Liu, T. Zhang, and D.-X. Zhao, ''Novel approach of distributed & adaptive trust metrics for MANET,'' *Wireless Netw.*, vol. 25, no. 6, pp. 3587–3603, 2019. doi: 10.1007/s11276-019-01955-2.

[17] D.-G. Zhang, T. Zhang, J. Zhang, Y. Dong, and X.-D. Zhang, ''A kind of effective data aggregating method based on compressive sensing for wireless sensor network,'' *EURASIP J. Wireless Commun. Netw.*, vol. 2018, p. 159, Dec. 2018. doi: 10.1186/s13638-018-1176-4.

[18] T. Zhang, D. Zhang, J. Qiu, X. Zhang, P. Zhao, and C. Gong, ''A kind of novel method of power allocation with limited cross-tier interference for CRN,'' *IEEE Access*, vol. 7, no. 1, pp. 82571–82583, 2019. doi: 10.1109/ACCESS.2019.2921310.

[19] D.-G. Zhang, K. Zheng, T. Zhang, and X. Wang, ''A novel multicast routing method with minimum transmission for WSN of cloud computing service,'' *Soft Comput.*, vol. 19, no. 7, pp. 1817–1827, Jul. 2015.

[20] D.-G. Zhang, S. Zhou, and Y.-M. Tang, ''A low duty cycle efficient MAC protocol based on self-adaption and predictive strategy,'' *Mobile Netw. Appl.*, vol. 23, no. 4, pp. 828–839, May 2017. doi: 10.1007/s11036-017-0878-x.

[21] D.-G. Zhang and X.-D. Zhang, ''Design and implementation of embedded un-interruptible power supply system (EUPSS) for Web-based mobile application,'' *Enterprise Inf. Syst.*, vol. 6, no. 4, pp. 473–489, Oct. 2011.

[22] Z. Fadika, E. Dede, J. Hartog, and M. Govindaraju, ''MARLA: MapReduce for heterogeneous clusters,'' in *Proc. CCGRID*, Washington, DC, USA, May 2012, pp. 49–56.

[23] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, ''Tarazu: Optimizing mapreduce on heterogeneous clusters,'' *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 61–74, Mar. 2012.

[24] D. Zhang, H. Ge, T. Zhang, Y.-Y. Cui, X. Liu, and G. Mao, ''New multi-hop clustering algorithm for vehicular ad hoc networks,'' *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 4, pp. 1517–1530, Apr. 2019. doi: 10.1109/TITS.2018.2853165.

[25] J. A. Rico-Gallego, J. C. Díaz-Martín, R. R. Manumachu, and A. L. Lastovetsky, ''A survey of communication performance models for high-performance computing,'' *ACM Comput. Surv.*, vol. 51, no. 6, Feb. 2019, Art. no. 126.

[26] S. Liu and X. H. Liu, ''Novel dynamic source routing protocol (DSR) based on genetic algorithm-bacterial foraging optimization (GA-BFO),'' *Int. J. Commun. Syst.*, vol. 31, no. 18, p. e3824, 2018. doi: 10.1002/dac.3824.

[27] D.-G. Zhang, T. Zhang, Y. Dong, X.-H. Liu, Y.-Y. Cui, and D.-X. Zhao, ''Novel optimized link state routing protocol based on quantum genetic strategy for mobile learning,'' *J. Netw. Comput. Appl.*, vol. 122, pp. 37–49, Nov. 2018. doi: 10.1016/j.jnca.2018.07.018.

[28] D.-G. Zhang, X. Wang, and X.-D. Song, ''New medical image fusion approach with coding based on SCD in wireless sensor network,'' *J. Elect. Eng. Technol.*, vol. 10, no. 6, pp. 2384–2392, Nov. 2015.

[29] H. Shi, Y. Zhao, B. Zhang, K. Yoshigoe, and A. V. Vasilakos, ''A free stale synchronous parallel strategy for distributed machine learning,'' in *Proc. ACM BDE*, Hong Kong, 2019, pp. 23–29.

[30] M. Li, D. G. Andersen, A. Smola, and K. Yu, ''Communication efficient distributed machine learning with the parameter server,'' in *Proc. NIPS*, Cambridge, MA, USA, 2014, pp. 19–27.

[31] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, ''Scaling distributed machine learning with the parameter server,'' in *Proc. OSDI*, Berkeley, CA, USA, 2014, pp. 583–598.

[32] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, ''Improving MapReduce performance in heterogeneous environments,'' in *Proc. OSDI*, San Diego, CA, USA, 2008, pp. 29–42.

[33] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, ''Twister: A runtime for iterative MapReduce,'' in *Proc. HPDC*, New York, NY, USA, 2010, pp. 810–818.

[34] L. Alarabi, M. F. Mokbel, and M. Musleh, ''ST-Hadoop: A MapReduce framework for spatio-temporal data,'' *GeoInformatica*, vol. 22, no. 4, pp. 785–813, Oct. 2018.

[35] D. Zhang, T. Zhang, and X. Liu, ''Novel self-adaptive routing service algorithm for application in VANET,'' *Appl. Intell.*, vol. 49, no. 5, pp. 1866–1879, 2019. doi: 10.1007/s10489-018-1368-y.

[36] D.-G. Zhang, X. Wang, X. Song, and D. Zhao, ''A novel approach to mapped correlation of ID for RFID anti-collision,'' *IEEE Trans. Services Comput.*, vol. 7, no. 4, pp. 741–748, Oct. 2014.

[37] D.-G. Zhang, K. Zheng, D.-X. Zhao, X.-D. Song, and X. Wang, ''Novel quick start (QS) method for optimization of TCP,'' *Wireless Netw.*, vol. 22, no. 1, pp. 211–222, Jan. 2016.

[38] D.-G. Zhang, Y.-N. Zhu, C.-P. Zhao, and W.-B. Dai, ''A new constructing approach for a weighted topology of wireless sensor networks based on local-world theory for the Internet of Things (IOT),'' *Comput. Math. Appl.*, vol. 64, no. 5, pp. 1044–1055, Sep. 2012.

[39] C. Liu, D. Zeng, H. Yao, X. Yan, L. Yu, and Z. Fu, ''An efficient iterative graph data processing framework based on bulk synchronous parallel model,'' *Concurrency Comput., Pract. Exper.*, p. e4432, Jan. 2018. doi: 10.1002/cpe.4432.

[40] S. Wang, W. Chen, A. Pi, and X. Zhou, ''Aggressive synchronization with partial processing for iterative ML jobs on clusters,'' in *Proc. 19th Int. Middleware Conf.*, New York, NY, USA, 2018, pp. 253–265.

[41] H.-F. Yu, C.-J. Hsieh, H. Yun, S. V. N. Vishwanathan, and I. S. Dhillon, ''A scalable asynchronous distributed algorithm for topic modeling,'' in *Proc. WWW*, Geneva, Switzerland, 2015, pp. 1340–1350.

[42] L. Cannelli, F. Facchinei, V. Kungurtsev, and G. Scutari, ''Asynchronous parallel nonconvex large-scale optimization,'' in *Proc. ICASSP*, New Orleans, LA, USA, Mar. 2017, pp. 4706–4710. doi: 10.1109/ICASSP.2017.7953049.

[43] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, ''Petuum: A new platform for distributed machine learning on big data,'' *IEEE Trans. Big Data*, vol. 1, no. 2, pp. 49–67, Jun. 2015. doi: 10.1109/TBDATA.2015.2472014.

[44] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, ''Managed communication and consistency for fast data-parallel iterative analytics,'' in *Proc. SoCC*, New York, NY, USA, 2015, pp. 381–394.

[45] X. Shi, B. Cui, Y. Shao, and Y. Tong, ''Tornado: A system for real-time iterative analysis over evolving data,'' in *Proc. SIGMOD*, New York, NY, USA, 2016, pp. 417–430.

[46] J. Zhang, H. Tu, Y. Ren, J. Wan, L. Zhou, M. Li, and J. Wang, ''An adaptive synchronous parallel strategy for distributed machine learning,'' *IEEE Access*, vol. 6, pp. 19222–19230, 2018. doi: 10.1109/ACCESS.2018.2820899.

[47] D. Zhang, G. Li, K. Zheng, X. Ming, and Z.-H. Pan, ''An energy-balanced routing method based on forward-aware factor for wireless sensor networks,'' *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 766–773, Feb. 2014.

[48] D.-G. Zhang, C. Chen, Y.-Y. Cui, and T. Zhang, ''New method of energy efficient subcarrier allocation based on evolutionary game theory,'' *Mobile Netw. Appl.*, vol. 9, pp. 1–14, Sep. 2018. doi: 10.1007/s11036-018-1123-y.

[49] D.-G. Zhang, S. Liu, T. Zhang, and Z. Liang, ''Novel unequal clustering routing protocol considering energy balancing based on network partition & distance for mobile education,'' *J. Netw. Comput. Appl.*, vol. 88, no. 15, pp. 1–9, 2017. doi: 10.1016/j.jnca.2017.03.025.

[50] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, ''Addressing the straggler problem for iterative convergent parallel ML,'' in *Proc. SoCC*, New York, NY, USA, 2016, pp. 98–111.

HANG SHI received the B.S. degree in information management and information system from Ningbo University, Ningbo, China, in 2018. He is currently pursuing the M.S. degree with the School of Computer Engineering and Science, Shanghai University. His research interests include big data analytics, distributed computing, and high-performance computing.

YUE ZHAO received the Ph.D. degree in integrated computing from the University of Arkansas at Little Rock, Little Rock, in 2015. He is currently a Lecturer with the School of Computer Engineering and Science, Shanghai University. He used to work as a Research Fellow at the Epithelial Systems Biology Laboratory, National Heart Lung and Blood Institute (NHLBI), National Institutes of Health (NIH), MD, USA. His research interests include big data analytics, distributed computing, high-performance computing, and bioinformatics.

BOFENG ZHANG received the B.S., M.S., and Ph.D. degrees from the Northwestern Polytechnical University of Technology, Xi'an, China, in 1991, 1994, and 1997, respectively.

From 1997 to 1999, he was a Postdoctoral Researcher with Zhejiang University, where he was promoted to an Associate Professor. Since 1999, he has been on the faculty of the School of Computer Engineering and Science, Shanghai University. From August 2006 to August 2007, he was a Visiting Professor with the University of Aizu, Japan. From September 2013 to September 2014, he was a Visiting Professor with Purdue University Calumet, USA. He is currently the Vice Dean of the School of Computer Engineering and Science, Shanghai University. He has published three books and about 150 research papers in national and international journals and major conference proceedings. His current research interests include intelligent information processing, data science and technology, and intelligent human–computer interaction. He serves as a Steering Committee Member, the Workshop Chair, and a Program Committee Member for several important international conferences.

KENJI YOSHIGOE (S'98–M'04–SM'13) received the Ph.D. degree in computer science and engineering from the University of South Florida, Tampa, FL, USA, in 2004. He is currently a Professor with the Faculty of Information Networking for Innovation and Design (INIAD), Toyo University, Tokyo, Japan. His current research interests include the reliability, security, and scalability of various interconnected systems ranging from high-performance computing system to resource-constrained wireless sensor networks to dynamically evolving social networks.

FURONG CHANG received the B.S. degree in information and computation science from Jingdezhen Ceramic Institute, Jiangxi, China, in 2009, and the M.S. degree in computer software and theory from Northwest University for Nationalities, Gansu, China, in 2012. She is currently pursuing the Ph.D. degree in computer application technology with Shanghai University, Shanghai, China. Since 2012, she has been a Lecturer with the School of Computer Science and Technology, Kashi University, Xinjiang, China. Her research interests include complex networks and data mining.