Yu H, Jiang XY, Zhao J *et al.* Toward high-performance delta-based iterative processing with a group-based approach. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(4): 797–813 July 2022. DOI 10.1007/s11390-022-2101-1

Toward High-Performance Delta-Based Iterative Processing with a Group-Based Approach

Hui Yu (余　辉), *Student Member*, *CCF*, Xin-Yu Jiang (姜新宇), *Student Member*, *CCF*
Jin Zhao (赵　进), *Student Member*, *CCF*, Hao Qi (齐　豪), *Student Member*, *CCF*
Yu Zhang* (张　宇), *Member*, *CCF*, *ACM*, *IEEE*
Xiao-Fei Liao (廖小飞), *Distinguished Member*, *CCF*, *Member*, *ACM*, *IEEE*
Hai-Kun Liu (刘海坤), *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*, Fu-Bing Mao (毛伏兵), *Member*, *CCF*, *ACM*, *IEEE*, and Hai Jin (金　海), *Fellow*, *CCF*, *IEEE*, *Member*, *ACM*

National Engineering Research Center for Big Data Technology and System, Huazhong University of Science and Technology, Wuhan 430074, China

Service Computing Technology and System Laboratory, Huazhong University of Science and Technology Wuhan 430074, China

Cluster and Grid Computing Laboratory, Huazhong University of Science and Technology, Wuhan 430074, China

School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

E-mail: {huiy, xinyujiang, zjin, theqihao, zhyu, xfliao, hkliu, fbmao, hjin}@hust.edu.cn

Received December 21, 2021; accepted June 29, 2022.

Abstract　Many systems have been built to employ the delta-based iterative execution model to support iterative algorithms on distributed platforms by exploiting the sparse computational dependencies between data items of these iterative algorithms in a synchronous or asynchronous approach. However, for large-scale iterative algorithms, existing synchronous solutions suffer from slow convergence speed and load imbalance, because of the strict barrier between iterations; while existing asynchronous approaches induce excessive redundant communication and computation cost as a result of being barrier-free. In view of the performance trade-off between these two approaches, this paper designs an efficient execution manager, called Aiter-R, which can be integrated into existing delta-based iterative processing systems to efficiently support the execution of delta-based iterative algorithms, by using our proposed group-based iterative execution approach. It can efficiently and correctly explore the middle ground of the two extremes. A heuristic scheduling algorithm is further proposed to allow an iterative algorithm to adaptively choose its trade-off point so as to achieve the maximum efficiency. Experimental results show that Aiter-R strikes a good balance between the synchronous and asynchronous policies and outperforms state-of-the-art solutions. It reduces the execution time by up to 54.1% and 84.6% in comparison with existing asynchronous and the synchronous models, respectively.

Keywords　iterative algorithm, delta-based execution model, efficiency

1　Introduction

Because many iterative algorithms have been recently proposed to analyze large-scale data in many domains, such as data mining and scientific computing, scalable and cheap distributed platforms, e.g., the cloud infrastructure, become promising platforms to support large-scale iterative processing. However, because iterative algorithms need to repeatedly handle the same large-scale data iteration by iteration until the results satisfy a user-given convergence or stopping condition, iterative algorithms still suffer from long time to converge on distributed platforms.

For efficient execution of large-scale iterative algo-

Regular Paper

Special Section of Xia Peisu Young Scholars Forum 2021

This paper is supported by the National Natural Science Foundation of China under Grant Nos. 61832006, 61825202, 62072193, and 61929103.

*Corresponding Author

rithms on distributed platforms, the Delta-based Iterative Execution (DIE) [1–7] model was proposed to reduce the execution time of a diverse set of iterative graph algorithms, or called delta-based iterative algorithms, by exploiting their sparse computational dependencies to avoid processing convergent states and also reduce synchronization cost. The delta-based iterative algorithms are the iterative algorithms which can be correctly executed by using the DIE model. These algorithms are prevalent in Internet applications and scientific computing, etc. The PageRank algorithm [1, 8–10], for example, is a well-known delta-based iterative algorithm widely used in web search engines. Other examples, such as adsorption [1, 11] and expected hitting time [1, 12], can be found in many real-world applications such as link prediction and recommendation systems. The Jacobi algorithm [1, 13] is also delta-based and has been adopted to solve large systems of linear equations in scientific computing.

The DIE model [1–7] can be executed in either a synchronous or an asynchronous way, while converging to the same final result. However, the synchronous DIE model not only needs to process large numbers of state changes for convergence because of the slow propagation of its (important) state changes, but also suffers from network jitters [14] and load imbalance [15] in the distributed platform for strict synchronization between iterations. The asynchronous DIE model, on the other hand, uses no barrier and converges more quickly for its speculative execution, but incurs high redundant communication and computation cost, because it generates excessive trigger actions, where a trigger action is a user given function invoked by each state change of the data item to handle this state change.

We observe that the synchronous DIE model and the asynchronous DIE model represent two extreme policies—the former has lower cost but converges more slowly, while the latter has a faster convergence speed but incurs higher cost to handle useless trigger actions. With this observation, we propose a group-based iterative execution model along with a heuristic scheduling algorithm, which can correctly and efficiently balance between the cost and benefits of the above two DIE models. The key idea of our proposed approach is to adaptively combine a group of state changes for each data item, and then use an effective scheduling algorithm to determine the optimal processing strategy of the combined state changes before pushing the results to the other data items. This approach has two main advantages. First, it enables delta-based iterative algorithms to efficiently switch between the synchronous DIE model and the asynchronous DIE model. Second, it enables much less communication and computation than the asynchronous model due to fewer trigger actions, and also faster convergence and lower synchronization cost than the synchronous model. To evaluate the efficiency of our approach, we also develop an efficient execution manager Aiter-R, which can be integrated into existing delta-based iterative processing systems so as to make them transparently and efficiently support the execution of delta-based iterative algorithms. In comparison with the cutting-edge asynchronous DIE model and the synchronous DIE model, experimental results show that Aiter-R can improve the performance by up to 2.18 times and 6.52 times, respectively.

In summary, this paper has three contributions.

1) We propose a group-based iterative execution model that seeks the trade-off between the synchronous DIE model and the asynchronous DIE model for large-scale delta-based iterative algorithms. It provides a means for the delta-based iterative algorithm to tune the trade-off between the benefits of reduced cost and the benefits of fast state propagation.

2) We propose a heuristic scheduling algorithm for each algorithm to further choose its own optimal trade-off point for maximum efficiency.

3) We design an efficient execution manager, i.e., Aiter-R, which can be integrated into existing delta-based iterative processing systems so as to make them transparently and efficiently support the execution of delta-based iterative algorithms.

4) We conduct extensive experiments to demonstrate its advantages. The results validate the efficiency of our Aiter-R by the fact that it achieves a higher performance improvement than the existing solutions over a cluster with 256 cores on 16 nodes.

The remainder of the paper is organized as follows. Section 2 gives a brief survey of related work. Section 3 and Section 4 describe the motivation and the main idea of our proposed group-based execution model and a heuristic scheduling algorithm. Section 5 describes its implementation details, followed by performance analysis in Section 6 and a comprehensive experimental evaluation in Section 7. Section 8 finally concludes the whole paper.

2 Related Work

With the explosive growth of data, many frameworks have been recently proposed to support iterative algorithms based on synchronous iterative processing or asynchronous iterative processing.

Synchronous Iterative Processing Frameworks. To support iterative algorithms, Twister[16] tries to buffer the results of each iteration in the main memory aiming to reduce high data access cost in each iteration. HaLoop[17] also tries to buffer the results to spare the irregular data access cost using a loop-aware task scheduler. For efficient data consistency and fault-tolerance, Piccolo[18] proposes Resilient Distributed Dataset (RDD). Based on Piccolo, Spark[19] further proposes and implements more optimizations, which can efficiently support irregular applications, such as iterative algorithms. However, these frameworks usually need global synchronization between iterations for iterative algorithms and suffer from high synchronization cost. Note that a series of dedicated frameworks are also designed to support iterative algorithms over graphs. These typical graph processing systems include Pregel[20], Chaos[21], and Powerlyra[22]. However, they can only support graph processing, instead of general iterative algorithms, e.g., Jacobi algorithm[1, 13].

Asynchronous Iterative Processing Frameworks. Previous work[23–27] shows that many iterative algorithms can also be executed in an asynchronous way and use the most recent state of data items because the state updates the other data items within the current iteration. Thus, the new state of the data item can be propagated more quickly in this asynchronous way than in the synchronous way, getting a faster convergence speed. CIEL[28] proposes to construct a dynamic task graph to support efficient execution of iterative algorithms based on data-flow. Domino[29] provides a novel programming model along with an efficient runtime system for iterative algorithms. It can correctly and efficiently support asynchronous execution of iterative algorithms. Meanwhile, priority scheduling[29, 30] is also proposed to accelerate the convergence of iterative algorithms instead of using the default round-robin scheduling algorithm. However, for the unawareness of the sparse computational dependencies in iterative algorithms, these systems, such as Domino, suffer from high runtime overhead. Based on this observation, the delta-based iterative execution model[1–7] has been recently proposed to exploit these sparse computational dependencies for better performance of iterative algorithms. With the execution model, data items of iterative algorithms are able to converge only based on delta data and the results on the previous iteration. Thus, it enables much low runtime overhead because it only needs to handle delta data. More importantly, priority scheduling algorithms[1, 6] can be more efficiently implemented on this delta-based iterative execution model for better performance of iterative algorithms.

3 Motivation

This section first describes the background and then discusses the inefficiency of existing solutions.

3.1 Delta-Based Iterative Execution Model

An iterative algorithm can be realized in several ways. Recent work[1–7] shows that a broad class of iterative algorithms (ranging from PageRank[8] of graph processing to the Jacobi algorithm[13] of scientific computing) can be correctly executed in the Delta-based Iterative Execution (DIE) model. These iterative algorithms are called delta-based iterative algorithms, which are conducted as follows:

$$\begin{cases} \boldsymbol{R}^{n+1} = \boldsymbol{R}^n \oplus \Delta\boldsymbol{R}^n, \\ \Delta\boldsymbol{R}^{n+1} = F(\Delta\boldsymbol{R}^n), \end{cases} \tag{1}$$

where $F()$ is a user-defined function and has distributive property. $\oplus$ is a user given general sum operation, which has commutative and associative properties. $\boldsymbol{R}^n = (\boldsymbol{R}^n(1), \ldots, \boldsymbol{R}^n(j))$, where $\boldsymbol{R}^n(j)$ is the j-th element to be processed at the n-th iteration. $\Delta\boldsymbol{R}^n = (\Delta\boldsymbol{R}^n(1), \ldots, \Delta\boldsymbol{R}^n(j))$, where $\Delta\boldsymbol{R}^n(j)$ is the state change of the element $\boldsymbol{R}^n(j)$, and $\boldsymbol{R}^{n+1}(j) = \boldsymbol{R}^n(j) \oplus \Delta\boldsymbol{R}^n(j)$. $\boldsymbol{R}^0$ and $\Delta\boldsymbol{R}^0$ are user-given initial constant vectors. Different from traditional execution models that iteratively update the state of a data item based on the previous states of other data items, the DIE model updates the state of a data item by only accumulating state changes of others. By such means, through employing the sparse computational dependencies, it can efficiently reduce the runtime overhead caused by the re-computation of data items that have been converged.

The DIE model can be executed either synchronously or asynchronously. In the synchronous model, the trigger actions are executed round by round with a global barrier. In the asynchronous DIE model, the trigger action is triggered when an element, for example $\boldsymbol{R}^n(j)$ of vector $\boldsymbol{R}^n$, has changed its state. In other words, a set of workers iteratively process elements of the vector in an asynchronous way. When a worker receives a state change $\Delta\boldsymbol{R}^n(j)$, it will process it. The whole process is finished when the user-defined condition is met, such as $\Delta\boldsymbol{R}^{n+1}(j)$ of all elements of $\boldsymbol{R}$

satisfying $\|\Delta \boldsymbol{R}^{n+1}(j)\| < \epsilon$ (where ϵ is a user given convergence condition). It has been demonstrated that the asynchronous DIE model converges to the same results as the synchronous DIE model. Besides, the processing order of the state change of each element, such as $\Delta \boldsymbol{R}^{n}(j)$, does not affect the convergence results[1]. It allows us to schedule the processing order of these state changes. Table 1 lists the notations used in this paper.

Table 1. Description of Notations

Notation	Meaning
$\oplus, \varepsilon$	User-given general sum operation and convergence condition respectively
$\boldsymbol{R}^n(j)$	The j-th element to be processed at the n-th iteration
$G(j)$	A group containing several state changes
$F()$	User-defined function
Δ	State change of the element
τ, T_a	Waiting time, a period used in our method respectively

3.2 Inefficiency of Existing Execution Models

However, the above models all suffer from suboptimal performance. In the following part, we take the asynchronous PageRank[8] as an example to illustrate it, where Algorithm 1 describes the trigger action function of the asynchronous PageRank. For the asynchronous PageRank, each page j accumulates the received delta ranking scores of its neighbors (e.g., $\Delta \boldsymbol{R}(k)$ of the page k) and then updates its ranking score $\boldsymbol{R}(j)$. Then, the delta ranking score $\Delta \boldsymbol{R}(j)$ of the page j is sent to j's neighbor pages, and $\Delta \boldsymbol{R}(j)$ is then set to 0. Each worker of the distributed environment is assigned with a set of web pages and updates the ranking scores of web pages until there are no more trigger actions. Note that, for other delta-based iterative algorithms, they have the same problem for the same reasons.

Algorithm 1. Trigger Action of Asynchronous PageRank /*Executed on the worker owning web page j*/

```
1:  procedure USER OPERATION($j$, $\Delta \boldsymbol{R}(j)$)
2:      $\boldsymbol{R}(j) \leftarrow \boldsymbol{R}(j) + \Delta \boldsymbol{R}(j)$
3:      if $\Delta \boldsymbol{R}(j) > \epsilon$ then
4:          $links \leftarrow$ look up outlinks of web page $j$
5:          for each link $<j, i> \in links$ do
6:              $\Delta \boldsymbol{R}(j) \leftarrow d \times \Delta \boldsymbol{R}(j)/\deg(j)$
7:              /*Propagate $\Delta \boldsymbol{R}(j)$ to the worker owing web page
    $i$ for its state update. $d$ is the damping factor. $\deg(j)$ represents the degree of $j$ */
8:              Diffuse($i$, $\Delta \boldsymbol{R}(j)$)
9:          end for
10:     end if
11: end procedure
```

We now discuss the inefficiency of PageRank with existing solutions. Assume that two state changes, i.e., $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$, are sent to worker 1 from two other workers, i.e., worker 2 and worker 3. $\Delta \boldsymbol{R}(j)'$ may arrive at worker 1 much later than $\Delta \boldsymbol{R}(j)''$ due to network jitters[14], or due to load imbalance between workers 2 and 3. For the synchronous DIE model, worker 1 has to wait until the arrival of both $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$ before processing them and propagating the results, and it results in much idle time. In the asynchronous DIE model, the trigger action for the processing of $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$ can be activated when it is received. Thus, it has no synchronization cost and can quickly propagate the state change of each data item to others following it along the directed paths of the dependency graph of all data items, without having to wait until the end of the current round. This phenomenon is called cascade effect. Therefore, it can converge more quickly than the synchronous DIE model. In the asynchronous DIE model, unlike the default round-robin scheduling algorithm[1], the priority scheduling algorithm[1, 30] is proposed to schedule the processing order of state changes according to their importance with regard to convergence. The priority of each state change is specified by a user-defined function. For example, the priority of $\Delta \boldsymbol{R}(j)$ can be evaluated via the value of $\Delta \boldsymbol{R}(j)$ itself. In this way, $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$ may not be processed until the convergence of the algorithm, when their priorities are small. Then, all trigger actions caused by them can be spared, getting better performance.

However, in order to quickly propagate the state changes of web pages, each state change in the asynchronous DIE model is processed individually and causes many trigger actions in subsequent iterations. Such a speculative execution approach may induce many redundant trigger actions, incurring unnecessary cost. For example, as described in Fig.1, assume web page j receives state changes $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$ from the web pages 1 and 2, respectively. After the processing of $\Delta \boldsymbol{R}(j)'$, the web page j will propagate its state change to $1, 2, \ldots, 5$ and causes trigger actions on them. Meanwhile, the echoing results from 1 and 2 will again cause trigger actions on j. This process proceeds until the termination condition is met, generating many trigger actions. Similarly, $\Delta \boldsymbol{R}(j)''$ will also cause the state change of j and its propagation to other web pages. In this process, $\Delta \boldsymbol{R}(j)'$ and $\Delta \boldsymbol{R}(j)''$ induce many similar trigger actions, i.e., redundant trigger actions. In reality, we can make the web page j wait a moment and combine $\Delta\boldsymbol{R}(j)'$ and $\Delta\boldsymbol{R}(j)''$ before their processing. Then, many similar actions caused by $\Delta\boldsymbol{R}(j)'$ and $\Delta\boldsymbol{R}(j)''$ in subsequent iterations are spared.

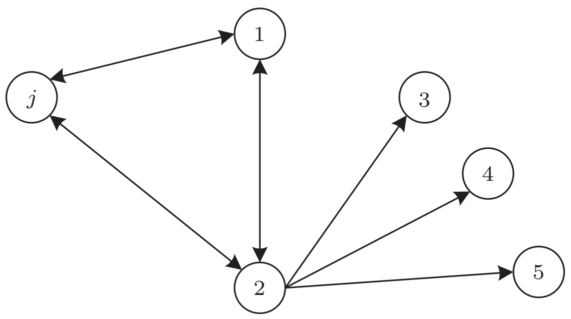


Fig.1. Illustration of redundant trigger actions in asynchronous DIE.

We can observe that both the synchronous DIE model and the asynchronous DIE model are two extreme policies and have their limitations: the synchronous model converges slowly, while the asynchronous model has higher redundant computation and communication cost. It motivates us to investigate whether and how we can efficiently explore the middle ground of these two extremes for better performance.

4 Our Approach

From the above discussion, we can also observe that the unprocessed state changes for the same data item can be combined before their processing and propagation. Based on this observation, we propose a group-based iterative execution model that tries to adaptively combine state changes for each data item before their processing. It opens an opportunity for the delta-based iterative algorithm to choose a trade-off point between the quick state propagation of asynchronous DIE model and the low runtime cost of synchronous DIE model. A efficient heuristic scheduling algorithm is also further proposed to allow the delta-based iterative algorithm to adaptively and efficiently choose its trade-off points for the maximum efficiency.

4.1 Group-Based Iterative Execution Model

The proposed group-based iterative execution model runs as described in Fig.2. When a state change $\Delta\boldsymbol{R}_j$ is received, it adds this state change into the related group $\boldsymbol{G}(j)$ using a user-defined function $\oplus$, which is a general sum operation[1]. $\boldsymbol{G}(j)= \{\Delta\boldsymbol{R}^k(j)|k = 1,2,\ldots\}$ is a group containing several state changes for the data item $\boldsymbol{R}(j)$. Meanwhile, the groups are periodically extracted in an asynchronous way. When the waiting time τ ($\tau \geqslant 0$ and its initial value is given by the user) runs out, several groups are extracted for user-defined operations (such as Algorithm 1) to process according to a scheduling algorithm. In this way, it allows us to efficiently trade off between the synchronous model and the asynchronous model by only setting τ and get the maximum efficiency by using a scheduling algorithm. Note that the asynchronous model is the special case with $\tau = 0$, while the synchronous one is the special case with τ=$\tau_{\rm syn}$. There, $\tau_{\rm syn}$ is the interval between two successive iterations for the synchronous model.

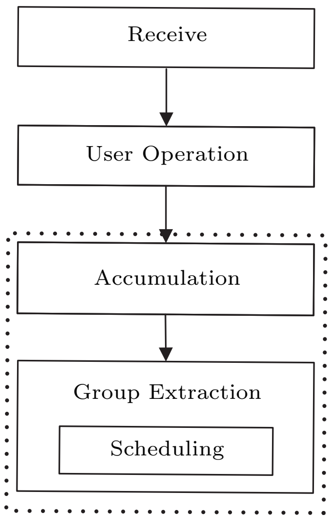


Fig.2. Dataflow of group-based iterative execution.

The average number of state changes gathered in a group may be very small under some conditions when τ is set to a constant. For better performance, we thus need an adaptive scheme to adjust the value of τ, especially because the execution environment of the distributed platforms varies with time. For this goal, we uses a function $B = B(\tau)$ to denote the gained benefits when τ is a specific value. Then, we can adjust τ to approximately get a maximum value of $B()$. Note that each worker can have its own τ according to its own execution environment. Frequent adjustment of τ may cause a high runtime overhead. Therefore, τ in our approach is adjusted with a period of T_a, considering that the execution environment changes at different rates for different algorithms. The value of T_a is given by the user and is often set larger than τ. If the execution environment changes frequently, T_a can be set smaller in order to make the value of τ quickly adapt to the environment. Otherwise, T_a is set larger aiming to reduce the runtime overhead. After setting T_a, the algorithm dynamically adjusts τ in the following way. Assume $\boldsymbol{C}(j)$ is the set of groups that were handled between the two successive processing times of $\boldsymbol{G}(j)$. The state changes gathered by a group $\boldsymbol{G}(j)$ are the state changes sent to this group within the time interval:

$$\begin{cases} T_{\text{interval}}(j) = \tau \times (1 + \xi(j)) + \pi(j), \\ \pi(j) = \sum_{t \in \boldsymbol{C}(j)} \sigma(t), \end{cases} \tag{2}$$

from its previous processing time, where $\sigma(t)$ is the time to process state changes contained in a group $\boldsymbol{G}(t)$, and $\xi(j)$ is the size of set $\boldsymbol{C}(j)$, i.e., the times that the group $\boldsymbol{G}(j)$ has been ruled out for processing from its previous processing. Thus, τ can be increased to make a group wait longer when fewer state changes have been gathered. On the contrary, τ can be decreased to reduce the waiting time, when sufficient state changes are gathered.

We can see that the function $B = B(\tau)$ only has one extreme point $\tau_{\max}$, and its value $B(\tau_{\max})$ is the maximum value of function $B(\tau)$. To find $\tau_{\max}$ according to the change of the execution environment, thus, it always increases τ with $\Delta\tau$ (given by the user) after a period of T_a, and the value of $\Delta\tau$ is set with its opposite number when

$$B(\tau_p + \Delta\tau) < B(\tau_p), \tag{3}$$

for the previous adjustment of τ, where $B(\tau_{\text{p}})$ and $B(\tau_{\text{p}+\Delta\tau})$ are the benefits gained in previous two intervals of T_a. Specifically, if deciding to further increase τ with $|\Delta\tau|$, it should be expected to get more benefits after this increment of τ. Otherwise, it decreases τ with $|\Delta\tau|$ for the new interval. Note that the value of $|\Delta\tau|$ should be set with a suitable value. Otherwise, it is difficult to get the value of $\tau_{\max}$ for τ when it is too large, or it needs a long time to get $\tau_{\max}$ for τ when it is too small. To efficiently get the approximate value of $B(\tau)$ for $\tau = \tau_{\text{p}}$ and $\tau = \tau_{\text{p}} + \Delta\tau$, in practice, we evaluate $B(\tau)$ using the number of state changes processed by the local worker during the interval of T_a.

4.2 Heuristic Scheduling Algorithm

From (2), we can find that the processing order of the groups has many important impacts on the value of $\xi(j)$. It means that the scheduling algorithm of our model affects the average number of state changes collected by its groups. Based on this observation, a heuristic scheduling algorithm is further proposed to tune the trade-off between the benefits of faster propagation of state changes and the benefits of more gathered state changes within the groups at the cost of more waiting time.

4.2.1 *Factors of Priority Definition*

In reality, the above trade-off can be evaluated with the following two factors.

Importance to Convergence (*ITC*). $ITC(\Delta\boldsymbol{R}^k(j))$ is used to evaluate the importance of a state change $\Delta\boldsymbol{R}^k(j)$ on the convergence speed of an iterative algorithm. The privileged processing of the state change with a larger ITC can accelerate the convergence. It is because an iterative algorithm may have been converged although many state changes with lower ITC are not handled. To calculate $ITC(\Delta\boldsymbol{R}^k(j))$ with low overhead, its value can be evaluated according to the value of $\Delta\boldsymbol{R}^k(j)$ itself. It requires the user to specify "±" with a user-defined function as priority scheduling [30]. Taking the asynchronous PageRank algorithm as an example, $ITC(\Delta\boldsymbol{R}^k(j)) = +\Delta\boldsymbol{R}^k(j)$ because a larger $\Delta\boldsymbol{R}^k(j)$ can make it converge more quickly. Therefore, $ITC(\boldsymbol{G}(j))$ of a group $\boldsymbol{G}(j)$ can be evaluated with the sum of ITC of all its state changes, i.e.,

$$\begin{cases} ITC(\boldsymbol{G}(j)) = \sum_{\Delta\boldsymbol{R}^k(j) \in \boldsymbol{G}(j)} ITC(\Delta\boldsymbol{R}^k(j)), \\ ITC(\Delta\boldsymbol{R}^k(j)) = \pm\Delta\boldsymbol{R}^k(j). \end{cases} \tag{4}$$

Cost to Grouping (*CTG*). Before the definition of CTG, we first discuss the heuristic hint for it. Considering the algorithm in Subsection 3.2, we assume a worker has two unprocessed groups $\boldsymbol{G}(j)$ and $\boldsymbol{G}(h)$ with the same ITC and the scheduler of this worker first selects $\boldsymbol{G}(j)$ to process. However, in the future, it is possible that the number of trigger actions saved by $\boldsymbol{G}(j)$ is much larger than that by $\boldsymbol{G}(h)$ because of two reasons. First, the number of state changes combined by $\boldsymbol{G}(j)$ in the future may be much larger than that of $\boldsymbol{G}(h)$. Second, the average number of trigger actions caused by the state changes of $\boldsymbol{G}(j)$ may also be much larger than that of $\boldsymbol{G}(h)$. Thus, $\boldsymbol{G}(j)$ is processed after more state changes are gathered. $CTG(\boldsymbol{G}(j))$ denotes the total cost of the number of trigger actions spared by a group $\boldsymbol{G}(j)$ in the future when processing $\boldsymbol{G}(j)$. In reality, CTG is leveraged to prolong $T_{\text{interval}}(j)$ and to gather more state changes by increasing $\xi(j)$ (described in (2)). $CTG(\boldsymbol{G}(j))$ can be expressed as

$$CTG(\boldsymbol{G}(j)) = \sum_{\Delta\boldsymbol{R}^k(j) \in \boldsymbol{G}(j)} Num(\Delta\boldsymbol{R}^k(j)), \tag{5}$$

where $\Delta\boldsymbol{R}^k(j)$ is the state change combined by the group $\boldsymbol{G}(j)$, and $Num(\Delta\boldsymbol{R}^k(j))$ is the number of trigger actions caused by the state change $\Delta\boldsymbol{R}^k(j)$ in future. To evaluate $CTG(\boldsymbol{G}(j))$, we also first introduce two factors.

1) $CR(h,\ j)$, which is the ratio of the number of processed state changes to the total number of state changes needed to be processed at the h-th round for the j-th element of vector $\boldsymbol{R}$. Note that a round means all elements of $\boldsymbol{R}$ are handled once. The value of $CR(h,\ j)$ can be approximately calculated as

$$CR(h,j) = \frac{N_P(h,j)}{N_T(h,j)}, \qquad (6)$$

where $N_P(h,\ j)$ and $N_T(h,\ j)$ are the number of state changes already processed and the number of state changes that need to be processed for element $\boldsymbol{R}(j)$ in the h-th round, respectively. Because $N_T(h,\ j)$ is unknown in advance and most of the vector $\boldsymbol{R}$'s elements have the same $N_T(h,\ j)$, we can approximately treat it as a constant. Then we can get that

$$CR(h,j) = \frac{N_P(h,j)}{T}, \qquad (7)$$

where T is a constant value gained from runtime.

2) $RN(\Delta\boldsymbol{R}^k(j))$, which is the round number of state change $\Delta\boldsymbol{R}^k(j)$. It is the number of hops from the initial state change value of $\boldsymbol{R}(j)$, i.e., $\Delta\boldsymbol{R}^0(j)$, to $\Delta\boldsymbol{R}^k(j)$ in the dependency graph. The value of $RN(\Delta\boldsymbol{R}^k(j))$ is calculated in the following way. $RN(\Delta\boldsymbol{R}^k(j))$ is set with h, if $\Delta\boldsymbol{R}^k(j)$ is the processed results of a state change with $RN = h - 1$.

Obviously, the number of state changes generated within the h-th round for the processing of $\boldsymbol{R}(j)$ is dependent on $CR(h,j)$. A round with a smaller CR may have more unprocessed state changes. Then, more state changes may arise in this round. In other words, when a round has a smaller value of CR, a state change is more likely to come from this round in future. In addition, because iterative computation is a process of refinement, the state change with a larger RN causes fewer trigger actions. Thus, the number of trigger actions caused by the state change $\Delta\boldsymbol{R}^k(j)$ in future is dependent on $RN(\Delta\boldsymbol{R}^k(j))$ and $CR(RN(\Delta\boldsymbol{R}^k(j)), j)$. The value of $CTG(\boldsymbol{G}(j))$ is negatively correlated with the value of CR and the value of RN and can be approximately evaluated via

$$CTG(\boldsymbol{G}(j)) = -\sum_{h\in S(j)} h \times CR(h,j), \qquad (8)$$

where $S(j) = \{h | RN(\Delta\boldsymbol{R}^k(j)) = h \bigwedge \Delta\boldsymbol{R}^k(j) \in \boldsymbol{G}(j)\}$.

4.2.2 *Priority Definition and Group-Based Scheduling*

Based on the values of ITC and CTG, the priority of each group in our heuristic scheduling algorithm can be calculated as follows. Because a group with a larger ITC can accelerate the propagation of important state changes, the priority of group $\boldsymbol{G}(j)$, i.e., $Pri(\boldsymbol{G}(j))$, is set to be positively correlated with the value of $ITC(\boldsymbol{G}(j))$. On the other hand, $Pri(\boldsymbol{G}(j))$ should be negatively correlated with $CTG(\boldsymbol{G}(j))$ in order to make the group with a larger value of CTG wait longer. Then, it has more opportunities to gather more state changes, thus reducing the cost of state propagation. Consequently, based on (4), (7), and (8), $Pri(\boldsymbol{G}(j))$ can be approximately evaluated in the following linear form for low profiling overhead.

$$\begin{cases} Pri(\boldsymbol{G}(j)) = \sum_{\Delta\boldsymbol{R}^k(j)\in\boldsymbol{G}(j)} Pri(\Delta\boldsymbol{R}^k(j)), \\ Pri(\Delta\boldsymbol{R}^k(j)) = ITC(\Delta\boldsymbol{R}^k(j)) + \beta \times RN(\Delta\boldsymbol{R}^k(j)), \end{cases}$$

where β is a constant. When a state change $\Delta\boldsymbol{R}^k(j)$ is gathered by a group $\boldsymbol{G}(j)$, $Pri(\boldsymbol{G}(j))$ can be efficiently updated incrementally using the priority of $\Delta\boldsymbol{R}^k(j)$. In this way, the priority of each group can be approximately obtained in real time. Besides, the unique constant β for the evaluation of $Pri(\Delta\boldsymbol{R}^k(j))$ can be easily set by the user or automatically determined at runtime. The value of β reflects the trade-off between the benefits from quick state propagation and the benefits of the spared cost by grouping more states.

For automatic setting of β, the runtime system decreases or increases β until its value is suitable, according to the profiled average priority of the groups processed by it.

5 Implementation of Aiter-R

Existing delta-based iterative processing systems always distribute the data over the workers of different nodes for parallel processing. Each worker maintains a subset of data items and receives state changes to concurrently update them according to the received state changes.

To make existing delta-based iterative processing systems transparently and efficiently support the execution of delta-based iterative algorithms, an efficient execution manager Aiter-R is developed by us, which can be integrated into these existing systems. Each worker of existing systems has an Aiter-R engine. Each Aiter-R engine has a local table, called GroupTable. It is used to manage and schedule the groups, where the unprocessed state changes with the same key are put into the same group. GroupTable is indexed by the key of each group. Each item of this table contains three fields: the key value j of a group, the priority value of this group for scheduling, and the delta value to accumulate those state changes gathered by this group. Each Aiter-R engine also has a Group extractor. The details are described in Algorithm 2. When a new unprocessed state change is received by the accumulator, it accumulates this state change's value with the delta value of a group according to the state change's key using the user-given function $\oplus$ (line 3). Meanwhile, the priority of this group is updated according to this state change's priority in real time with a low overhead (lines 4 and 5).

Algorithm 2. Details of Group Extractor

```
1: procedure ACCUMULATOR(DataItem $j$, $\Delta \boldsymbol{R}^k(j)$)
2:     /*Accumulate $\Delta \boldsymbol{R}^k(j)$ into $\boldsymbol{G}(j)$ for $j$.*/
3:     $\boldsymbol{G}(j).\Delta v \leftarrow \boldsymbol{G}(j).\Delta v \oplus \Delta \boldsymbol{R}^k(j)$
4:     $Pri(\Delta \boldsymbol{R}^k(j)) \leftarrow ITC(\Delta \boldsymbol{R}^k(j)) + \beta \times RN(\Delta \boldsymbol{R}^k(j))$
5:     $Pri(\boldsymbol{G}(j)) \leftarrow Pri(\boldsymbol{G}(j)) + Pri(\Delta \boldsymbol{R}^k(j))$
6: end procedure
7: procedure SCHEDULING ALGORITHM
8:     /*Get $N_g$ groups with the highest priorities
9:     from the local worker $W_l$.*/
10:    $\boldsymbol{G}_{\text{set}} \leftarrow GetGroups(W_l, N_g)$
11:    for each group $\boldsymbol{G}(j) \in \boldsymbol{G}_{\text{set}}$ do
12:        /*Output the state changes accumulated in
13:        group $\boldsymbol{G}(j)$ to the user-defined operation
14:        for the processing of the data item $\boldsymbol{R}(j)$.*/
15:        Output($j$, $\boldsymbol{G}(j).\Delta v$)
16:    end for
17: end procedure
```

To efficiently assign the processing order of the groups owned by each Aiter-R engine, it also has a scheduler. When the waiting time has elapsed, the scheduler selects several highest priority groups and outputs them to the user-defined operation, e.g., the function described in Algorithm 1, so as to allow them to be first processed by existing systems integrated with Aiter-R.

To reduce the cost caused by frequent group extraction, we make Aiter-R engine on each worker extract an adjustable number of groups each time. In this way, it can trade off between the benefits of prioritized execution and the runtime overhead by adjusting the number of groups extracted each time, i.e., N_g.

In reality, as described in Algorithm 3, we approximately extract the top N_g records by sorting the samples, where N is the number of groups stored in GroupTable. The idea behinds this algorithm is that the priority distribution in a small set of samples in reality can reflect the status of GroupTable. In this way, it only takes $O(N)$ cost to extract the top N_g groups rather than $O(N \times \log N)$. Note that the number of samples, i.e., s, is also important to the performance. A large s may introduce high runtime cost, while a small one may reduce the benefits gained from prioritization.

Algorithm 3. Get Groups with the Highest Priorities

```
1: procedure GETGROUPS(Worker $W$, $N_g$)
2:     /*Randomly select $s(s \ll N)$ records.*/
3:     $Set_{\text{sample}} \leftarrow RandomGet(W.\text{GroupTable})$
4:     /*Sort samples in priority-descending order.*/
5:     Descending($Set_{\text{sample}}$)
6:     /*Use threshold $T_t$ to extract $N_g$ groups
7:      with the largest priority.*/
8:     $T_t \leftarrow Pri(Set_{\text{sample}}[\frac{s \times N_g}{N}])$
9:     for each $\boldsymbol{G}(j) \in W$.GroupTable do
10:        if $Pri(\boldsymbol{G}(j)) \geqslant T_t$ and $N_g > 0$ then
11:            /*Insert $\boldsymbol{G}(j)$ into group set $\boldsymbol{G}_{\text{set}}$.*/
12:            $InsertGroup(\boldsymbol{G}_{\text{set}}, \boldsymbol{G}(j))$
13:            $N_g \leftarrow N_g - 1$
14:        end if
15:    end for
16:    Return $\boldsymbol{G}_{\text{set}}$
17: end procedure
```

Now, we analyze the optimal value of N_g that gives the best performance. The execution time of asynchronous iterative computation is composed of two parts: the time to process trigger actions and the runtime cost. We derive these two parts as follows. First, let $f(N_g)$ be the total number of trigger actions needed for convergence when the number of groups extracted each time is set to N_g. Let T_p be the average processing time of each trigger action, including the time for computation and the time for communication. Thus, the total time spent on handling trigger actions is $f(N_g) \times T_p$. Second, the overhead time is dominated by the time used to extract groups from GroupTable, which is linear with the size of GroupTable, i.e., N. Let T_o be the average time to scan a record in GroupTable. Then, the total overhead time in asynchronous iterative computation is $\frac{f(N_g)}{N_g} \times N \times T_o$, where $\frac{f(N_g)}{N_g}$ is the times to extract groups. Because $f(N_g)$ can be approximately estimated as a linear function of N_g, we can get the total execution time T_{total} of asynchronous iterative computation as follows:

$$\begin{cases} T_{\text{total}} = f(N_g) \times T_p + \dfrac{f(N_g)}{N_g} \times N \times T_o, \\ f(N_g) = \alpha \times N_g + \theta. \end{cases}$$

Therefore, to get the minimum value of T_{total}, we have $N_g = (\frac{\theta \times N \times T_o}{\alpha \times T_p})^{1/2}$, where the value of N_g can be provided by the user or be set at runtime. Section 7 gives the performance of benchmarks with different values of

N_g and demonstrates that it can improve the performance over a wide range of N_g. To set N_g at runtime, it needs to get T_o, T_p, α, and θ. T_o and T_p can be gained by profiling the execution of the previous iteration, and α and θ can be estimated by online analysis as well. Specifically, it can use the above sampling method to approximate the total number of trigger actions, i.e., $f(N_g)$, needed for convergence of the whole dataset by only evaluating the number of trigger actions for the samples. Then, α and θ can be calculated by resolving two linear equations about α and θ.

6 Performance Analysis

This section gives an analysis of different execution models. In reality, the execution time of different schemes of the DIE model all can be evaluated via

$$T_{\text{total}} = T_{\text{trigger}} + T_{\text{cost}},$$

where T_{trigger} is the time to process the trigger actions needed for convergence and T_{cost} is the runtime overhead. Because the processing time of each trigger action is the same, then

$$T_{\text{total}} = T_p \times N_{\text{trigger}} + T_{\text{cost}},$$

where N_{trigger} is the number of trigger actions needed for convergence and T_p is the processing time of each trigger action, including the time for both computation and communication.

Assume the priority asynchronous approach needs $N^{(2)}_{\text{trigger}}$ trigger actions for convergence. Then the cost is

$$T^{(2)}_{\text{cost}} = T_o \times N \times \frac{N^{(2)}_{\text{trigger}}}{N_g},$$

at least, where N_g is the number of data items extracted each time and T_o is the average time for scanning a record in the table with size N. Thus, the total execution time of the priority asynchronous approach is

$$\begin{aligned} T^{(2)}_{\text{total}} &= T_p \times N^{(2)}_{\text{trigger}} + T^{(2)}_{\text{cost}} \\ &= T_p \times N^{(2)}_{\text{trigger}} \times \left(1 + T_o \times \frac{N}{T_p \times N_g}\right). \end{aligned}$$

The approximate execution time of the round robin asynchronous approach is

$$\begin{aligned} T^{(3)}_{\text{total}} &= T_p \times N^{(3)}_{\text{trigger}} + T_o \times N^{(3)}_{\text{trigger}} \\ &= T_p \times N^{(3)}_{\text{trigger}} \times \left(1 + \frac{T_o}{T_p}\right), \end{aligned}$$

where $N^{(3)}_{\text{trigger}}$ is the number of trigger actions needed for its convergence. $N^{(3)}_{\text{trigger}}$ is usually larger than $N^{(2)}_{\text{trigger}}$ because the priority asynchronous way converges quicker than the round robin asynchronous way [1].

However, the number of trigger actions needed for the convergence of our approach is only $N^{(1)}_{\text{trigger}}$, where

$$\frac{N^{(2)}_{\text{trigger}}}{N_{\text{Group}}} < N^{(1)}_{\text{trigger}} < \frac{N^{(3)}_{\text{trigger}}}{N_{\text{Group}}},$$

and N_{Group} is the average number of state changes gathered in each group of our approach. The runtime overhead of our approach is

$$T^{(1)}_{\text{cost}} = (\tau + T_o \times N) \times \frac{N^{(1)}_{\text{trigger}}}{N_g},$$

where N_g is the number of groups extracted each time and T_o is the average time to scan a record in the GroupTable with the size of N. Therefore, the total execution time of our approach is

$$T^{(1)}_{\text{total}} = T_p \times N^{(1)}_{\text{trigger}} \times \left(1 + \frac{\tau + T_o \times N}{T_p \times N_g}\right).$$

It means that

$$T'^{(2)}_{\text{total}} < T^{(1)}_{\text{total}} < T'^{(3)}_{\text{total}},$$

where

$$T'^{(2)}_{\text{total}} = T^{(2)}_{\text{total}} \times \frac{1 + \frac{\tau}{T_p \times N_g + T_o \times N}}{N_{\text{Group}}},$$

and

$$T'^{(3)}_{\text{total}} = T^{(3)}_{\text{total}} \times \frac{T_p \times N_g + T_o \times N + \tau}{(T_p \times N_g + T_o \times N_g) \times N_{\text{Group}}}.$$

On the other hand, assume $L_{\max}$ and L_{avg} are the maximum and the mean computation load of all workers, respectively. Then, the computational imbalance degree λ_L can be expressed as

$$\lambda_L = \frac{L_{\max}}{L_{\text{avg}}} - 1.$$

Then, the total execution time of the synchronous approach is

$$\begin{aligned} T^{(4)}_{\text{total}} &= T_p \times N^{(4)}_{\text{trigger}} \times (1 + \lambda_L) + T^{(4)}_{\text{cost}} \\ &= T_p \times N^{(4)}_{\text{trigger}} \times (1 + \lambda_L) + T_o \times N^{(4)}_{\text{trigger}} \\ &= T_p \times N^{(4)}_{\text{trigger}} \times \left(1 + \lambda_L + \frac{T_o}{T_p}\right), \end{aligned}$$

where $N_{\text{trigger}}^{(4)}$ is the number of trigger actions needed for convergence and is usually larger than $N_{\text{trigger}}^{(2)}$ because the asynchronous approach can propagate a new state of the data item more quickly [1].

In summary, by comparing the total execution time of each execution model, we can conclude that our execution model is superior to the other execution models, i.e., the total time spent by our execution model is the lowest, because our approach can propagate the new state of the data item more efficiently. Therefore, we theoretically demonstrate the effectiveness of our method.

7 Experimental Evaluation

The hardware platform is a cluster with 16 nodes, where each node has two octuple-cores Intel® Xeon® E5-2670 CPUs at 2.60 GHz with 64 GB memory, running a Linux operation system with kernel version 2.6.32. The nodes are interconnected by a 2-Gigabit Ethernet. Each node spawns 16 workers to run the benchmarks. The algorithms are compiled by gcc v4.7.2. Four typical benchmarks from data mining and scientific computing are implemented:

1) RISMF [31], which is used to profile the relationships between users and items in a recommender system;

2) PageRank [8], which is a popular algorithm to rank web pages;

3) Adsorption [11], which is a graph-based label propagation algorithm and provides personalized recommendation for contents;

4) Jacobi algorithm [13], which is employed to solve linear equations in scientific computation.

Table 2 shows the datasets used for these benchmarks. The first three datasets are real-world ones①–③ and the fourth one is a generated sparse matrix as Maiter [1].

Table 2. Statistics of Dataset

Benchmark	Dataset
RISMF①	Rows: 1.8 million; columns: 0.136 million
PageRank②	Nodes: 3.5 billion; edges: 128.7 billion
Adsorption③	Nodes: 133.6 million; edges: 5.5 billion
Jacobi algorithm③	Rows: 1 million; columns: 1 million

In order to evaluate the performance of our approach and also understand the advantages of our approach, Aiter-R is integrated with Maiter and then is compared with the following four schemes implemented on Maiter. They differ only in the execution model and the scheduling algorithm. Note that we still call the version of Maiter integrated with Aiter-R as Aiter-R in the following experiments:

1) Asyn-RR: the asynchronous DIE model with round robin scheduling [1], which is the default scheduling algorithm;

2) Asyn-Pri: the asynchronous DIE model with priority scheduling [1], which is the scheme employing ITC as the priority and the value of ITC is specified by a user-defined function;

3) Syn-with/Syn-without: the synchronous DIE model [1] with or without a combiner, respectively;

4) Group-RR: the group-based iterative execution model with round robin scheduling.

Note that we choose Maiter because it is demonstrated to outperform other existing systems for the execution of delta-based iterative algorithms [1]. In addition, Aiter-R is also compared with other cutting-edge systems, i.e., Spark v3.2.0 [19] and Domino [29], which support iterative processing on distributed platforms. Their performance is tried to be tuned to be the best. In the experiments, for Aiter-R, we set $\beta = 5 \times 10^{-5}$, $|\Delta\tau| = 0.0005$ and $T_a = 0.5$, $\frac{N_g}{\sqrt{N}} = 20$ and $s = 1000$ for both PageRank and Adsorption, where N is the number of data items processed by different benchmarks. In Aiter-R, $\frac{N_g}{\sqrt{N}}$ and s are set to 4 and 200 for the RISMF and Jacobi algorithm respectively. All experimental results are the average of 10 repetitive runs.

7.1 Number of Trigger Actions for Convergence

First, we evaluate the impacts of the two factors, i.e., ITC and CTG, on the reduction of the number of triggered actions. To get this goal, we first set $\beta = 0$ and then evaluate how many trigger actions can be reduced with only ITC. We then evaluate the influence of CTG in a similar way by setting β to its maximum in order to ignore the impact of ITC. The results are normalized with respect to their sum and presented in Fig.3. From this figure, we have two observations. First, ITC and CTG are both important to the reduction of redundant trigger actions. Taking the RISMF

①http://webscope.sandbox.yahoo.com/catalog.php?datatype=r, Apr. 2022.

②http://webdatacommons.org/hyperlinkgraph/, Apr. 2022.

③http://law.di.unimi.it/datasets.php, Apr. 2022.

algorithm as an example, only taking ITC and CTG as the factor of the priority reduces the number of trigger actions by up to 51.1% and 48.9%, respectively. Second, we can observe that ITC always reduces more trigger actions than CTG for the four benchmarks. It means that ITC is more important than CTG on the reduction of trigger actions.

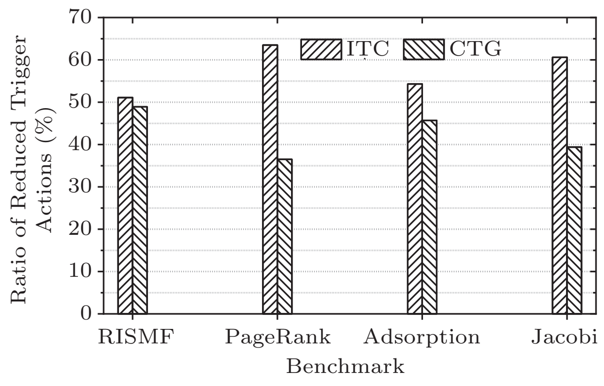


Fig.3. Ratio of saved trigger actions by using only ITC or CTG as priority, respectively.

Fig.4 depicts the number of trigger actions needed by different schemes for convergence normalized to the number of trigger actions needed by Asyn-RR. We can observe that the synchronous approach needs much more trigger actions than the asynchronous approach, because the latter can propagate the new value of the data item more quickly to help convergence. Taking the RISMF algorithm as an example, the trigger actions needed by Syn-without are 4.56 times as many as those by Asyn-RR. Meanwhile, we can observe that, in the Jacobi algorithm, 86.0% of trigger actions of Syn-without can be reduced by Syn-with. It means that a combiner can be used in the synchronous approach so as to reduce trigger actions for better performance.

Fig.4 also shows that Asyn-Pri can dramatically reduce the number of trigger actions needed for the convergence of Asyn-RR. Taking the Jacobi algorithm as an example, Asyn-Pri can reduce the number of trigger actions of Asyn-RR up to 67.2%. It is because that the most important state changes with regard to its convergence are more quickly propagated by the priority scheduling algorithm of Asyn-Pri in comparison with Asyn-RR. As a result, Asyn-Pri can converge faster than Asyn-RR. Meanwhile, as shown in the figure, Asyn-Pri still has more trigger actions than Aiter-R. Taking RISMF as an example, Aiter-R can reduce 73.5% of the trigger actions of Asyn-Pri. It mainly has two reasons. First, state changes for the same data item can be effectively grouped and processed together in Aiter-R. To demonstrate it, we have also evaluated the number of trigger actions needed by Group-RR. For the Jacobi algorithm, it can be observed that 63.8% of the trigger actions needed by Asyn-RR are spared by Group-RR. Second, many trigger actions can be reduced by our proposed scheduling method. For example, Aiter-R only needs 25.4% of the trigger actions of Group-RR in the Jacobi algorithm.

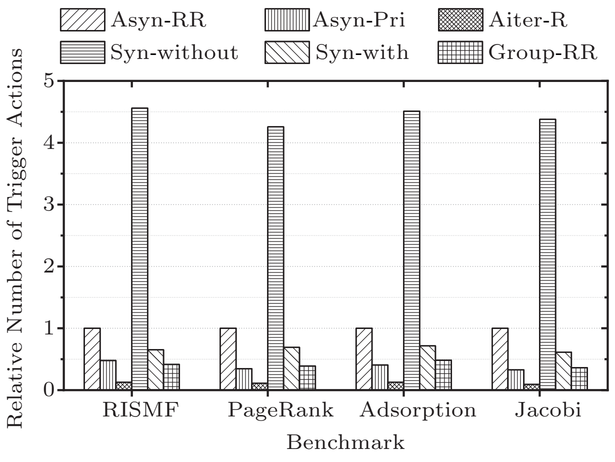


Fig.4. Number of trigger actions needed for convergence under different schemes normalized to that of Asyn-RR.

Finally, Table 3 shows the convergence condition of Aiter-R and Asyn-RR on different benchmarks by profiling the time of them to get a fixed value of ED. Note that ED is the Euclidean distance between the result vector of Aiter-R or Asyn-RR and that of Syn-without. From this table, we can observe that Aiter-R converges to the same final result as Asyn-RR and Syn-without in a faster way.

Table 3. Convergence Time (s) of Asyn-RR and Aiter-R on Different Algorithms

ED	RISMF		PageRank		Adsorption		Jacobi Algorithm	
	Asyn-RR	Aiter-R	Asyn-RR	Aiter-R	Asyn-RR	Aiter-R	Asyn-RR	Aiter-R
10^{-1}	168.9	53.7	64.9	34.9	42.1	20.0	334.4	133.1
10^{-2}	383.0	119.3	150.0	76.7	94.7	43.8	730.4	279.7
10^{-3}	830.1	250.6	337.9	166.5	205.2	92.0	1 531.2	559.9
10^{-4}	1 695.8	501.2	699.2	333.4	423.4	182.5	3 107.2	1 062.8
10^{-5}	3 373.2	977.9	1 366.8	633.6	808.7	346.2	5 788.4	1 913.4

7.2 Runtime Overhead

Fig.5 shows the execution time breakdown of Asyn-RR, Asyn-Pri and Aiter-R. Although the overhead (i.e., the time spent on non-computational tasks) occupies 78.7% of the total execution time of Aiter-R, significant reduction of trigger actions yet is obtained by Aiter-R as the following discussed. Note that Aiter-R needs much lower runtime overhead than Asyn-Pri. Fig. 6 shows the computational overhead of different schemes, where the time to handle redundant trigger actions is not included in this computational overhead. Fig. 6 shows that the computational overheads of Syn-with is less than that of Asyn-RR. For example, in the Jacobi algorithm, the computational overhead of Syn-with is only 61.5% of that of Asyn-RR. Meanwhile, we find that Domino also suffers from higher runtime overhead than Asyn-RR because Domino is based on a costly distributed locking engine. In RISMF, the computational overhead of Domino is 8.03 times higher than that of Asyn-RR.

In addition, from Fig.6, we see that Asyn-Pri has a higher overhead than Asyn-RR in scheduling the processing order according to the importance of different trigger actions with regard to its convergence. For the Jacobi algorithm, Asyn-Pri has a 4.81 times higher computational overhead than Asyn-RR. In the Jacobi algorithm for example, the computational overhead of Asyn-Pri is 1 950.7 seconds, while it is only 1 418.1 seconds for Aiter-R. Meanwhile, we can find that the computational overhead of the data item extraction algorithm in Asyn-Pri is high. For example, the computational overhead of Group-RR is only 694.5 seconds for the Jacobi algorithm.

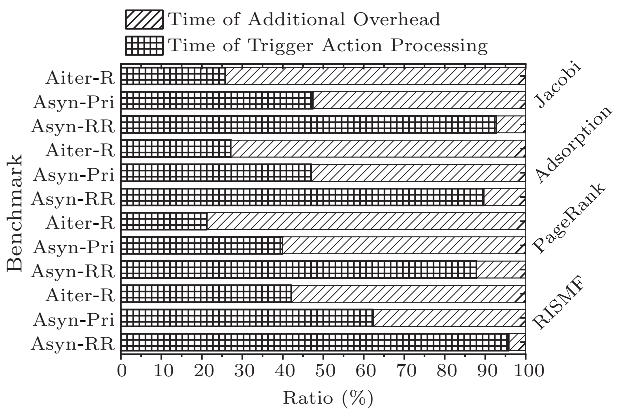


Fig.5. Execution time breakdown of Asyn-RR, Asyn-Pri and Aiter-R.

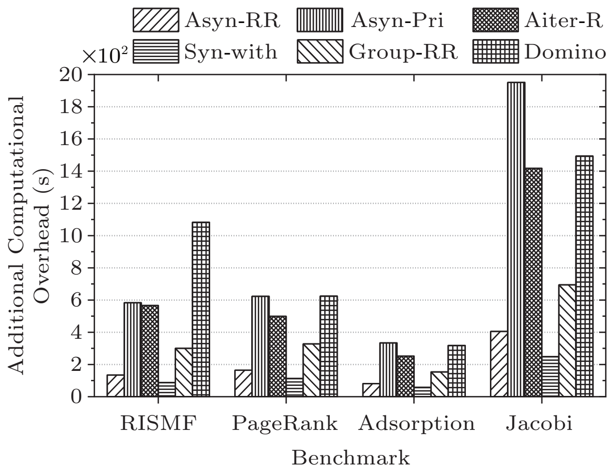


Fig.6. Computational overheads of different schemes.

Finally, the communication overhead of different schemes is evaluated. Fig. 7(a) and Fig. 7(b) present the volume of traffic and the number of communication operations for different schemes normalized to that of

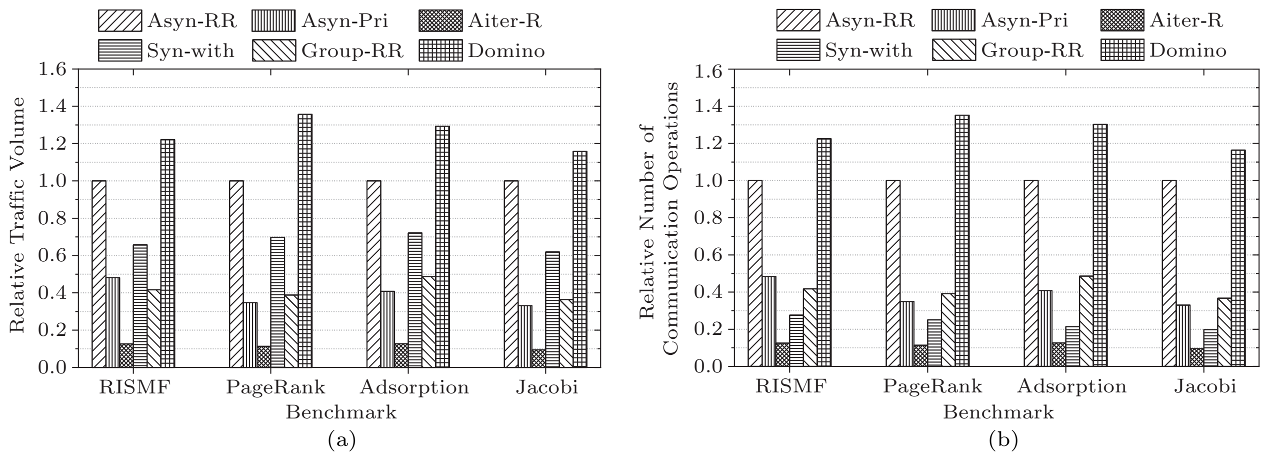


Fig.7. Communication overheads of different schemes normalized to that of Asyn-RR. (a) Traffic volume. (b) Number of communication operations.

Asyn-RR respectively. We can observe that, for the Jacobi algorithm, Aiter-R can reduce both the volume of traffic and the number of communication operations by up to 71.6% in comparison with Asyn-Pri, and by up to 90.6% in comparison with Asyn-RR due to much fewer trigger actions. It also means that Aiter-R has better scalability for lower communication cost. Meanwhile, we see that the synchronous approach can reduce the number of communication operations. Taking Adsorption as an example, the number of communication operations needed by Syn-with is only 21.4% of that by Asyn-RR, and the traffic volume of Syn-with is 72.1% of that of Asyn-RR.

7.3 Execution Time

In Fig.8, we have also evaluated the performance of Aiter-R on a platform of AliCloud with 80 Gbps network and 32 nodes. Fig.8 and Fig.9 give the execution time of the benchmarks by using different schemes. It can be observed that the synchronous approach performs worse than the asynchronous approach because of a slower convergence speed and more significant load imbalance (a worker in the synchronous approach may even take 3.58 times as much load as the average load of all workers). For example, the execution time of the Jacobi algorithm is as long as 11 891.5 seconds for Syn-with, although it uses combiners to reduce network traffic. However, with Asyn-RR, the Jacobi algorithm only needs 5 788.4 seconds, because the asynchronous approach requires much fewer trigger actions to converge than the synchronous method, and also has no synchronization cost.

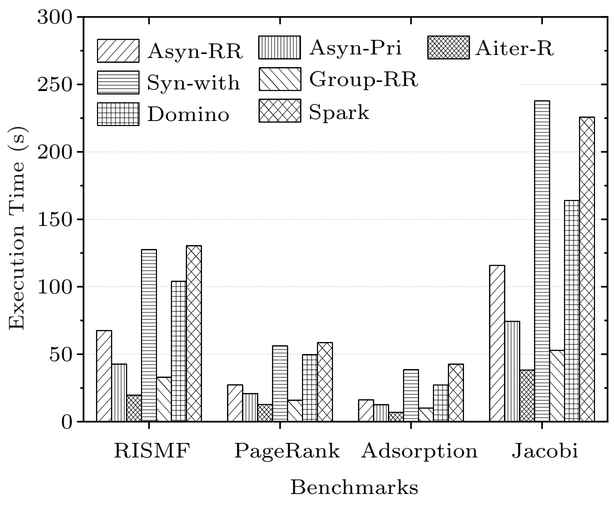


Fig.8. Execution time of different schemes on AliCloud.

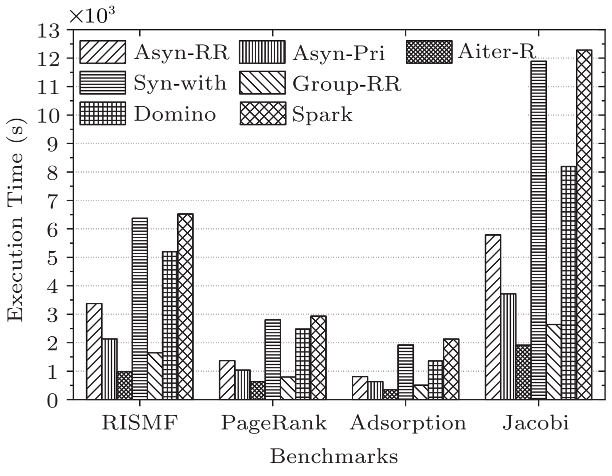


Fig.9. Execution time of different schemes on 16 nodes cluster.

However, Asyn-RR still performs poorly due to many redundant trigger actions. For the Jacobi algorithm, Asyn-Pri only needs 3 716.7 seconds, while Aiter-R even only takes 1 913.4 seconds because of the reduction of redundant trigger actions with low overhead. In other words, Aiter-R achieves a speedup of 1.94 and 3.03 in comparison with Asyn-Pri and Asyn-RR, respectively. For the RISMF algorithm, Aiter-R can even improve the performance of Asyn-Pri by 2.18 times and that of Syn-with by 6.52 times. There are two main reasons for the superior performance of Aiter-R in comparison with the above two asynchronous execution schemes, namely Asyn-RR and Asyn-Pri. First, the adaptive group scheme of Aiter-R can be used to spare many trigger actions. As shown in Fig.9, the execution time of the RISMF algorithm is only 1 644.1 seconds for Group-RR, while 3 373.2 seconds for Asyn-RR. It also means that the benefit from our group-based approach is still much more than the overhead caused by it. Second, from Fig. 9, we can observe that the scheduling algorithm of Aiter-R can further improve the performance because of significantly reduced trigger actions for faster state propagation, although it incurs higher runtime overhead. For example, the execution time of the RISMF algorithm over Aiter-R is only 977.9 seconds, much lower than 1 644.1 seconds over Group-RR.

Aiter-R is also compared with two state-of-the-art systems, i.e., Domino and Spark. As shown in Fig. 9, both Domino and Spark suffer from poorer performance than Aiter-R. Domino has high runtime overhead in realizing its execution model with a costly distributed locking engine. Spark suffers from a slow convergence speed and high synchronization cost because of its global barrier between iterations. Therefore, Aiter-R can improve the performance of RISMF by up to 5.32 times and 6.67 times in comparison with Domino and Spark, respectively, yet with the same accuracy of results. Finally, Fig.10 also evaluates the scalability of Aiter-R. Fig.10 shows that Aiter-R can achieve good scalability. It is because Aiter-R can spare much communication cost caused by redundant trigger actions.

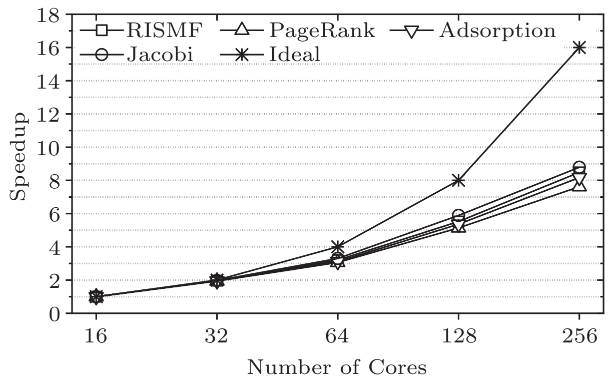


Fig.10. Scalability of Aiter-R for various cores.

7.4 Impacts of Various Parameters

Fig.11(a) gives the performance of Aiter-R normalized to that of Group-RR with different values of N_g. This figure shows that too large N_g may degrade the benefits of the scheduling algorithm of Aiter-R. In the extreme case, i.e., the value of N_g is the same as the size of GroupTable, no benefits will be gotten from the scheduling algorithm for the iterative computation. On the other hand, setting N_g too small may lead to frequent group extraction, which incurs considerable runtime overhead. In addition, as shown in Fig.11(a), we can observe that, over a wide range of N_g, the method that more benefits are gotten by the extraction of top groups with the highest priorities than by randomly outputting N_g groups without considering any priority. It is because that Aiter-R can more efficiently propagate state changes than Group-RR and accelerate its convergence speed. Meanwhile, more state changes can be gathered in each group for data items of Aiter-R, reducing the communication and computation cost of Group-RR.

In order to show the impact of sample rate s on the performance of Aiter-R, Fig.11(b) shows the execution time of Aiter-R with different values of s normalized to that of the baseline condition with $s = 40$. Note that, for the baseline, the execution time of the

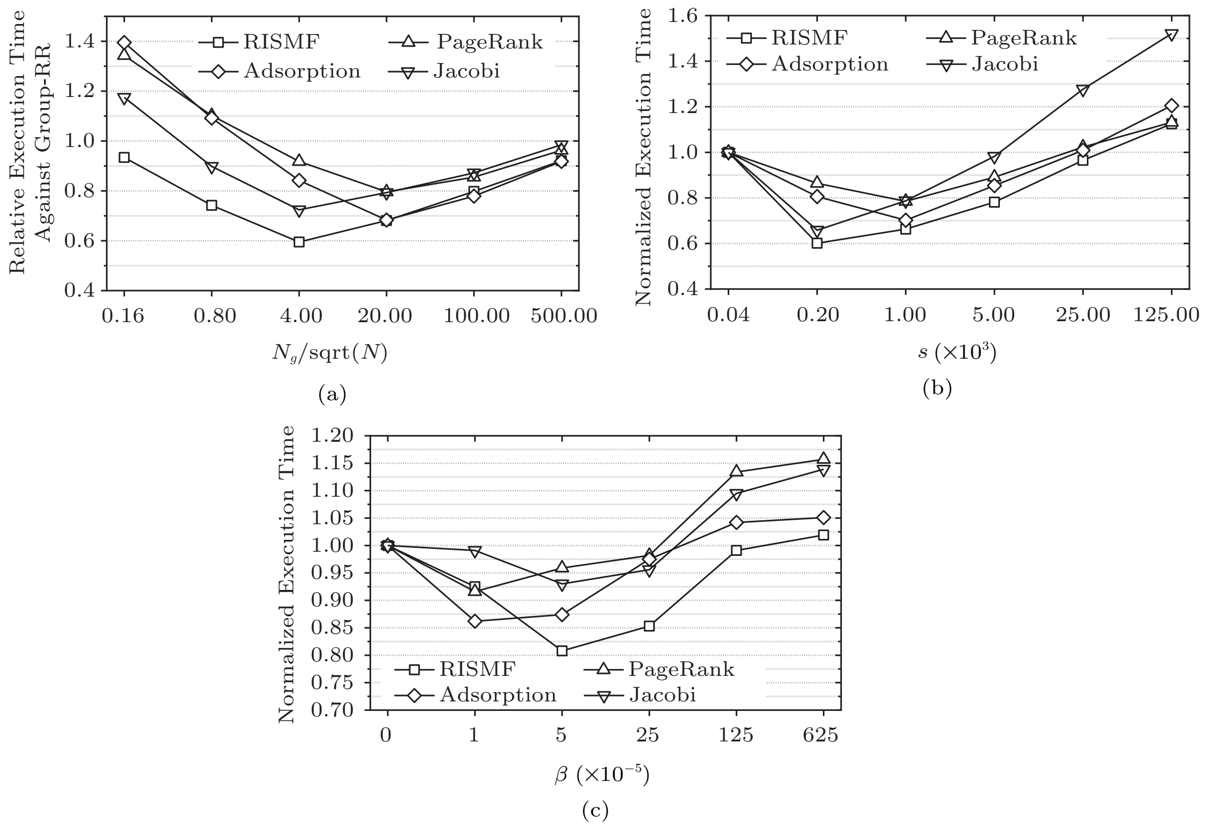


Fig.11. Impacts of (a) N_g, (b) s and (c) β on the performance of Aiter-R.

RISMF, PageRank, Adsorption, and Jacobi algorithms is 1 627.1, 807.1, 493.1, and 2 907.9 seconds, respectively. We can find a phenomenon that too small s may degrade the effect of prioritization. On the other hand, it may incur considerable overhead to arrange the samples. Fig. 11(c) depicts the execution time of Aiter-R with different values of β normalized to that of the baseline condition with $\beta = 0$, where the execution time of the RISMF, PageRank, Adsorption, and Jacobi algorithms is 1 210.2 s, 660.6 s, 396.1 s, and 2 057.4 s, respectively. Fig. 11(c) shows that whether the parameter β is set too large or too small will may lead to performance losses. It is because β represents a trade-off between the gain from quick state propagation and the benefits of cost reduction by more aggressive state grouping. Finally, Table 4 also shows the impacts of T_a and $\Delta\tau$ on the performance of Aiter-R. We can observe that either too small or too large value of both T_a and $\Delta\tau$ may result in worse performance for Aiter-R.

Table 4. Impacts of T_a and $\Delta\tau$ on the Execution Time (s) of PageRank under Aiter-R

T_a	$\|\Delta\tau\|$			
	0.000 1	0.000 5	0.002 5	0.012 5
0.10	648.4	652.5	667.6	684.2
0.25	642.5	642.2	655.9	672.5
0.50	646.8	633.6	647.0	661.7
1.00	655.2	640.3	647.7	660.1

8 Conclusions

This paper proposed a group-based iterative execution model, along with an efficient heuristic scheduling algorithm, to efficiently support large-scale delta-based iterative algorithms on the distributed platforms. Comprehensive experimental results showed that our approach achieves better performance than existing solutions. In the future, we will investigate how to efficiently optimize our approach to support the processing of large-scale streaming data.

References

[1] Zhang Y, Gao Q, Gao L, Wang C. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2014, 25(8): 2091-2100. DOI: 10.1109/TPDS.2013.235.

[2] Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012, pp.17-30.

[3] Mihaylov S R, Ives Z G, Guha S. REX: Recursive, delta-based data-centric computation. *Proc. the VLDB Endowment*, 2012, 5(11): 1280-1291. DOI: 10.14778/2350229.2350246.

[4] Yu W, Lin X, Zhang W. Fast incremental SimRank on link-evolving graphs. In *Proc. the 30th IEEE International Conference on Data Engineering*, Mar. 31-Apr. 4, 2014, pp.304-315. DOI: 10.1109/ICDE.2014.6816660.

[5] Zhang Y, Chen S, Wang Q, Yu G. i^{2}MapReduce: Incremental MapReduce for mining evolving big data. *IEEE Transactions on Knowledge and Data Engineering*, 2015, 27(7): 1906-1919. DOI: 10.1109/TKDE.2015.2397438.

[6] Zhang Y, Liao X, Jin H, Gu L, Zhou B B. FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE Transactions on Knowledge and Data Engineering*, 2018, 30(5): 895-907. DOI: 10.1109/TKDE.2017.2781241.

[7] Zhang Y, Liao X, Shi X, Jin H, He B. Efficient disk-based directed graph processing: A strongly connected component approach. *IEEE Transactions on Parallel and Distributed Systems*, 2018, 29(4): 830-842. DOI: 10.1109/TPDS.2017.2776115.

[8] Hou G, Chen X, Wang S, Wei Z. Massively parallel algorithms for personalized PageRank. *Proc. the VLDB Endowment*, 2021, 14(9): 1668-1680. DOI: 10.14778/3461535.3461554.

[9] Chen H, Jin H, Cui X. Hybrid followee recommendation in microblogging systems. *Science China Information Sciences*, 2017, 60(1): Article No. 012102. DOI: 10.1007/s11432-016-5551-7.

[10] Liao X, Chen Y, Zhang Y *et al.* An efficient incremental strongly connected components algorithm for evolving directed graphs. *Scientia Sinica Informationis*, 2019, 49(8): 988-1004. DOI: 10.1360/N112018-00125. (in Chinese)

[11] Baluja S, Seth R, Sivakumar D *et al.* Video suggestion and discovery for YouTube: Taking random walks through the view graph. In *Proc. the 17th International Conference on World Wide Web*, Apr. 2008, pp. 895-904. DOI: 10.1145/1367497.1367618.

[12] Liben-Nowell D, Kleinberg J. The link prediction problem for social networks. In *Proc. the 12th International Conference on Information and Knowledge Management*, Nov. 2003, pp.556-559. DOI: 10.1145/956863.956972.

[13] Shroff G M. A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix. *Numerische Mathematik*, 1990, 58(1): 779-805. DOI: 10.1007/BF01385654.

[14] Zhang Y, Liao X, Jin H, Min G. Resisting skew-accumulation for time-stepped applications in the cloud via exploiting parallelism. *IEEE Transactions on Cloud Computing*, 2015, 3(1): 54-65. DOI: 10.1109/TCC.2014.2328594.

[15] Zhang Y, Liao X, Jin H, Tan G, Min G. Inc-part: Incremental partitioning for load balancing in large-scale behavioral simulations. *IEEE Transactions on Parallel and Distributed Systems*, 2015, 26(7): 1900-1909. DOI: 10.1109/TPDS.2014.2333511.

[16] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S, Qiu J, Fox G. Twister: A runtime for iterative MapReduce. In *Proc. the 19th ACM International Symposium on High Performance Distributed Computing*, Jun. 2010, pp.810-818. DOI: 10.1145/1851476.1851593.

[17] Bu Y, Howe B, Balazinska M, Ernst M D. HaLoop: Efficient iterative data processing on large clusters. *Proc. the VLDB Endowment*, 2010, 3(1): 285-296. DOI: 10.14778/1920841.1920881.

[18] Power R, Li J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. the 9th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2010, pp.293-306.

[19] Zaharia M, Chowdhury M, FranklinM J, Shenker S, Stoica I. Spark: Cluster computing with working sets. In *Proc. the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, Jun. 2010.

[20] Malewicz G, Austern M H, Bik A J C, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In *Proc. the 2010 ACM SIGMOD International Conference on Management of Data*, Jun. 2010, pp.135-146. DOI: 10.1145/1807167.1807184.

[21] Roy A, Bindschaedler L, Malicevic J, Zwaenepoel W. Chaos: Scale-out graph processing from secondary storage. In *Proc. the 25th Symposium on Operating Systems Principles*, Oct. 2015, pp.410-424. DOI: 10.1145/2815400.2815408.

[22] Chen R, Shi J, Chen Y, Chen H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. the 10th European Conference on Computer Systems*, Apr. 2015, Article No. 1. DOI: 10.1145/2741948.2741970.

[23] Chazan D, Miranker W. Chaotic relaxation. *Linear Algebra and Its Applications*, 1969, 2(2): 199-222. DOI: 10.1016/0024-3795(69)90028-7.

[24] Baudet G M. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 1978, 25(2): 226-244. DOI: 10.1145/322063.322067.

[25] Bertsekas D P. Distributed asynchronous computation of fixed points. *Mathematical Programming*, 1983, 27(1): 107-120. DOI: 10.1007/BF02591967.

[26] Liu H K, Chen D, Jin H, Liao X F, He B S, Hu K, Zhang Y. A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions. *Journal of Computer Science and Technology*, 2021, 36(1): 4-32. DOI: 10.1007/s11390-020-0780-z.

[27] Lv X Q, Xiao W, Zhang Y, Liao X F, Jin H, Hua S Q. An effective framework for asynchronous incremental graph processing. *Frontiers of Computer Science*, 2019, 13(3): 539-551. DOI: 10.1007/s11704-018-7443-z.

[28] Murray D G, Schwarzkopf M, Smowton C, Smith S, Madhavapeddy A, Hand S. CIEL: A universal execution engine for distributed data-flow computing. In *Proc. the 8th USENIX Conference on Networked Systems Design and Implementation*, Mar. 30-Apr. 1, 2011, pp.113-126.

[29] Dai D, Chen Y, Kimpe D, Ross R B. Trigger-based incremental data processing with unified sync and async model. *IEEE Transactions on Cloud Computing*, 2021, 9(1): 372-385. DOI: 10.1109/TCC.2018.2830348.

[30] Zhang Y, Gao Q, Gao L, Wang C. PrIter: A distributed framework for prioritized iterative computations. In *Proc. the 2nd ACM Symposium on Cloud Computing*, Oct. 2011, Article No. 13. DOI: 10.1145/2038916.2038929.

[31] Takács G, Pilászy I, Németh B, Tikk D. Scalable collaborative filtering approaches for large recommender systems. *Journal of Machine Learning Research*, 2009, 10: 623-656.

Hui Yu is currently a Ph.D. candidate in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. His current research interests include graph processing and graph neural network.

Xin-Yu Jiang is currently a Master student in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his B.S. degree in software engineering at Hunan University, Changsha, in 2020. His current research interests include graph processing and graph neural network.

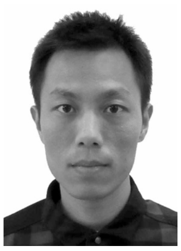
Jin Zhao is currently a Master student in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. he received his B.S. degree in software engineering at Hunan University, Changsha, in 2020. His current research interests include graph processing and graph neural network.

Hao Qi is currently a Ph.D. candidate in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. His current research interests include graph processing, system software, and architecture.

Yu Zhang received his Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, in 2016. He is currently an associate professor with the School of Computer Science and Technology, HUST, Wuhan. His research interests include computer architecture, system software, runtime optimization, programming model, and big data processing. He is a member of CCF, ACM, and IEEE.

Xiao-Fei Liao received his Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), Wuhan, in 2005. He is now the vice dean of the School of Computer Science and Technology at HUST, Wuhan. He has served as a reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P system, cluster computing and streaming services. He is a distinguished member of CCF and a member of ACM and IEEE.

Hai-Kun Liu received his Ph.D. degree from Huazhong University of Science and Technology, Wuhan. He is an associate professor with the School of Computer Science and Technology, HUST, Wuhan. His current research interests include in-memory computing, virtualization technologies, and cloud computing. He is a senior member of CCF, and a member of ACM and IEEE.



Fu-Bing Mao received his Ph.D. degree from Nanyang Technological University, Singapore. He is currently an assistant professor in School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan. His research interests include computer architecture, system software, FPGA physical design and optimization algorithms.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST), Wuhan. Jin received his Ph.D. degree in computer engineering from HUST, Wuhan, in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at the University of Hong Kong, Hong Kong, between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is a fellow of CCF and IEEE and a member of ACM. He has co-authored 15 books and published over 600 research papers. His research interests include computer architecture and virtualization.