

TSSA: A Two Step Scheduling Algorithm for the Event-driven Clusters

Yu Chen*, Mingming Sun*

*China Unionpay, Shanghai, China

Email: smilessun@hotmail.com, sunmingming@unionpay.com

Abstract—The event-driven programming model has been proposed to efficiently process iterative applications and incremental applications. In clusters based the event-driven model, applications are structured as a series of short triggers, each of which will be invoked when associate events are triggered. And framework assigns a newly submitted trigger to a node where the relevant datasets set. Unfortunately it may lead to load imbalance because associate events may occur by chance. Numerous triggers in a node may be simultaneously invoked but other nodes have no triggers running. To the end, we provide TSSA, a new two step event-driven for the event-driven clusters to maximize improve the utilization of node resources. Our results indicate TSSA performs well, and minimizes total execution time of applications.

Index Terms—event-driven clusters, event-driven, iterative applications

I. INTRODUCTION

Along with the popularity of cloud platforms such as Amazon EC2 and Windows Azure, more and more applications are turning to a distributed computation fashion to solve problems brought by 'Big Data'. They generally can be classified into two cases: iterative applications and incremental applications. Unfortunately, the synchronous data-flow model such as popular MapReduce [1], Dryad [2] and their variants is deficient for such applications. For example, incremental applications only need to process on the portion of data updates each time to achieve continuous correct results, like PageRank, distributed crawlers. However, MapReduce model must run over the entire dataset for every increment, incurring a significant performance penalty.

To address the needs of such applications, the event-driven programming model is proposed as an asynchronous computation model such as Percolator [4], Oolong [5] and Domino [6] et al. The method follows the Event-Condition-Action (ECA) rule[3], which has been widely researched in active database field. ECA rules are straightforward: when the event occurs, evaluate the condition; if the condition is fulfilled, execute the action automatically. Applications are structured as a series of triggers; each trigger completes a computation task and creates more works for "downstream" triggers by updating datasets. This method partitions the large-scale computing tasks into a number of smaller parallel triggers. For example, incremental applications only need to process the updated dataset and some relevant data which is affected by the updated dataset, instead of entire dataset. Therefore, the event-driven model over performs the synchronous data-flow model on iterative and incremental applications.

In clusters based event-driven model, it is possible for some nodes to be heavily loaded while others are lightly loaded since specific events are needed to invoke triggers, because the specific events occur by chance. So framework needs a load balancer to achieve better utilization of node resources and a high system throughput. Many load balancing algorithms which support distributed systems have been proposed and reviewed in the literature. But few of them have been designed to support the event-driven frameworks. Jobs composed of short, sub-second trigger tasks present a different load balancing challenge. In this article we propose TSSA (two step scheduling algorithm), a novel, yet practical, scheme which attempts to optimally balance the load for the event-driven cluster. TSSA contains two steps: 1. it assigns the trigger to one node where the relevant datasets set; 2. it use distributed random two sample load balancing algorithm to balance load. The goal in solving this performance problem is to achieve maximize improve the utilization of node resources. In summary, this paper main contributions are:

- We analyzed load imbalancing of the event-driven cluster in depth.
- We proposed a new dynamic random sample load balancing strategy that coordinates and redistributes triggers invoked based on the current load state of the groups and nodes. It adopts threshold-based strategy, which focuses on relative lightly loaded workers, to make no node in idle state, instead of roughly equalizing the workload among all nodes.
- Trigger group mechanism is to divide triggers invoked into groups as the basic unit of transfer triggers supposing the high-workload worker is chose. Because trigger may be short second or sub-second task. This can reduce the number of network I/O.
- We have implemented and deployed our algorithm on Domino, which is a trigger-based programming framework for writing general distributed high performance iterative or incremental applications. And it is available open source. Our results indicate that our load balancing performs well.

This paper is organized as follows. Section 2 describes our scheme. We implement and evaluate our algorithm in section 3 and section 4. We survey related work in section 5 and end with conclusion section 5.

II. ALGORITHM DESIGN

In this section, we present the TSSA design issues in detail, and discuss the techniques required to achieve our goal. Our design seeks to minimize total execution time of applications and improve the utilization of clusters' resources.

A. TSSA framework

In the event-driven programming model, applications are structured as a series of triggers; each trigger completes a computation task and creates more works for "downstream" triggers by updating datasets. An external process invokes the first trigger in the chain by modifying initial dataset. Triggers should terminated when final results have been calculated or results converge.

The framework of existing event-driven clusters such as Percolator, Oolong and Domino contains one Trigger Master node, which performs framework management, and many Trigger Workers. Trigger Master is responsible for assigning triggers to Trigger Workers. Trigger Workers periodically send heartbeat message to Trigger Master.

With a view to each trigger be invoked when its relevant dataset is updated. Thus the first step of TSSA is that scheduler assigns the trigger to Trigger Workers where the relevant datasets set, instead of adding the trigger to task waiting queue in Trigger Master. There is an advantage that once the dataset changes the corresponding triggers can be immediately invoked without the reaction delay from scheduler. The scheduling policy avoids frequent communication with Trigger Master and reduces response time.

However it may lead to load imbalance because triggers are invoked when specific events occur and during runtime the cluster resource usage cannot be determined. A Trigger Worker executes more triggers than it can run concurrently, then it queues new trigger until existing tasks complete. But, other workers may be idle. It is hard to schedule tasks and avoid hot workers when trigger submitted. Load imbalance distribution reduces the performance of clusters. Load unbalance must be detected and measured, and an appropriate migration strategy can be made during runtime based on current workload. So the second step of TSSA is that coordinate unbalanced load in Trigger Workers.

B. First Step: assign triggers

Scheduler in Trigger Master assigns the new trigger to Trigger Workers based on the location information of relevant dataset. Each Trigger Worker scans storage system for changed dataset and invokes the corresponding triggers. Each Trigger Worker has a trigger waiting queue, so invoked triggers will be add into trigger waiting queue. Workload of Trigger Worker can be considered as the sum of execution time of triggers in trigger waiting queue. Trigger Worker periodically sends itself workload to Trigger Master by heartbeat message.

C. Second Step: load balance

The event-driven programming model follows the Event-Condition-Action (ECA) rule. As specific events of triggers

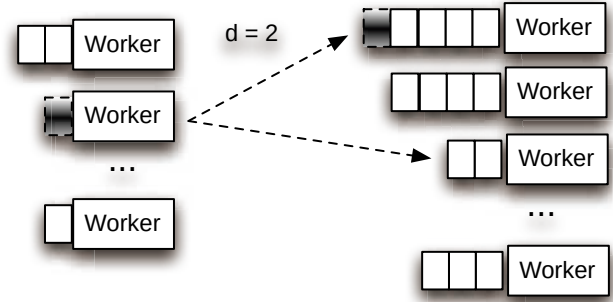


Fig. 1. the power of two choices

occur by chance, it is probably that workload in Trigger Workers is unbalanced. It is very important to redistribute triggers in cluster. We adopt a decentralized, randomized sampling load balancing approach to obtain near-optimal performance.

The goal of our load balancing policy is to ensure optimal use of distributed resources whereby no node remains in idle state, instead of roughly equalizing the workload among all nodes. TSSA focuses on Trigger Workers, whose workload drops below a threshold called *Tunder*, called as light-workload worker. *Tunder* is greater than the sum of load balancing policy time, the heartbeat interval (default as 5 second) and network transmission time of a trigger group.

$$T_{under} > 2 * T_{ask} + T_{heart} + T_{tran} \quad (1)$$

One of the benefits is that during the transferring triggers, light-workload worker can be still handle with triggers in trigger waiting queue. This method reduces the number of load balancing and avoids the drawbacks of frequently network transmission.

TSSA adopts the power of two choices[26] to get heavy-workload worker for light-workload worker(Fig.1). The light-workload worker random obtains two workers, and get some workload from heavier workload worker.

Algorithm 1 Load Balancing Algorithm for Light Worker

```

while  $n.load < T_{under}$  do
   $B, C < -random\_two\_worker()$ ;
   $a < -heaviest\_load(B, C)$ ;
  if  $U(a, n) > 0$  then
     $group\_fetch(a, n)$ ;
  else
     $D, F < -random\_two\_worker()$ ;
     $b < -heaviest\_load(D, F)$ ;
    if  $U(b, n) > 0$  then
       $group\_fetch(b, n)$ ;
    else
       $sendheartbeattoTriggerMaster$ ;
       $c < -response\_mater()$ ;
       $group\_fetch(c, n)$ ;
    end if
  end if
end if
end while

```

Algorithm 1 illustrates the process of load balancing algorithm of TSSA for light-workload worker. Each worker periodically check itself workload. Once a light-workload worker n is below the T_{under} , load balancer will be invoked. Firstly, load balancer selects two random workers, and thus chooses heaviest workload worker a . Secondly, load balancer computes $U(a, n)$ to determine whether the triggers needs to be transferred from a . If $U(a, n)$ is positive, load balancer will request triggers from a . Otherwise, it is not necessary to transfer tasks.

$$U(a, n) = a_{load} - TriGroup - (n_{load} + TriGroup + T_{tran}) \quad (2)$$

$TriGroup$ contains some triggers as the basic unit of transfer. T_{tran} is communication cost with two workers for transferring a $TriGroup$ since routers or hubs. If failed to obtain a heavy workload worker, worker n try again to random select two other worker, and so on. Unfortunately, when the second choices also is failed to choose a heavy workload worker, load balancer will request Trigger Master to get a heaviest workload worker by heartbeat messages. Trigger Master has total workload of workers by heartbeat messages. Worker n will obtain $TriGroup$ from the heaviest workload worker. Of course, if workers in cluster is idle, Trigger Master response no worker by heartbeat messages. In summary, a light-workload worker can get $TriGroup$ from relative heavy workload worker to make all worker busy.

D. Macro-control strategy

When most workers in cluster are in idle status, light-workload workers will continuously request workload to the same busy worker. The busy worker needs to continuously response to requests, leading to aggravating the pressure. To solve this problem, we design a central macro-control strategy. Exactly, Trigger Master can easy obtain all workload of workers by heartbeat messages. Trigger Master computes idle rate, which is used to determine whether the load balancing initiator is a light-load or heavy-load worker. The idle rate q means the percentage of light-load workers.

$$Y = q * n * \frac{C_{n-2}^{2d-1}}{C_{n-1}^{2d}} \quad (3)$$

$$Y = q * n * \frac{2d}{(n-2d-1)(n-1)} \quad (4)$$

$$q = \frac{Y * (n-2d-1)(n-1)}{2d * n} \quad (5)$$

d is the number of random choices. In this paper, d is 2. Y is the number of requests, which heavy-workload worker receives from light-workload workers. In practice, we can define Y , and then get q . When current idle rate is lower than defined q , the load balancing initiator is light-load worker, otherwise, heavy-load worker.

Algorithm 2 illustrates the process of load balancing algorithm of TSSA for heavy-workload worker. Once heavy-workload worker receive the transfer instruction from Trigger Master. T_{min} is the average value of all workers' workload. In

Algorithm 2 Load Balancing Algorithm for Heavy Worker

```

while  $m.load < T_{min}$  do
   $B, C < -random\_two\_worker()$ ;
   $a < -lightest\_load(B, C)$ ;
  if  $U(m, a) > 0$  then
     $group\_fetch(m, a)$ ;
  else
     $D, F < -random\_two\_worker()$ ;
     $b < -lightest\_load(D, F)$ ;
    if  $U(n, b) > 0$  then
       $group\_fetch(m, b)$ ;
    else
       $sendheartbeattoTriggerMaster$ ;
       $c < -response\_mater()$ ;
       $group\_fetch(m, c)$ ;
    end if
  end if
end while

```

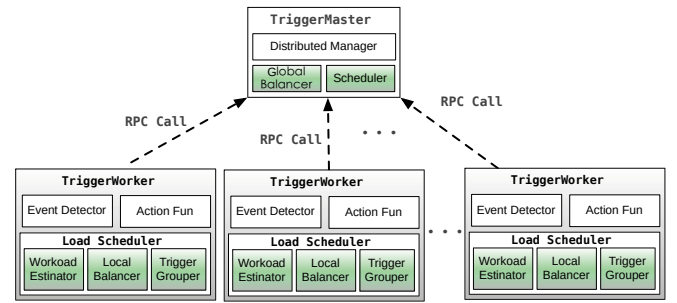


Fig. 2. Architecture of TSSA on Domino

the same way, heavy-workload worker selects lighter-workload worker by the power of two choices. If heavy-workload worker can not obtain a lighter-workload worker, it will request Trigger Master to get one worker.

III. IMPLEMENTATION

A. TSSA on Domino

TSSA consists of five main modules: a Scheduler and a Global Balancer in Trigger Master, a Local Balancer and Workload Estimator and Trigger Grouper in each Trigger Worker in Fig.2. Scheduler is responsible for the first step of scheduling algorithm. Scheduler assigns each new trigger to Trigger Worker where the relevant datasets set. Global Balancer analyzes workload of all Trigger Worker, and then redistributes workload to balance load.

Workload Estimator automatically predicts the execution time of trigger and evaluates current total workload. In this paper, Each TriggerWorker workload is the sum of the number of all triggers invoker in waiting queue. Because most triggers are smile, second tasks. Trigger waiting queue length is a popular indicator of workload[19]. Each Trigger Worker send itself workload to Trigger Master by heart beat mechanism. Local Balancer coordinates and redistributes triggers invoked according to the decision of Global Balancer. It is the key of TSSA. Trigger Grouper picks out several suitable triggers to

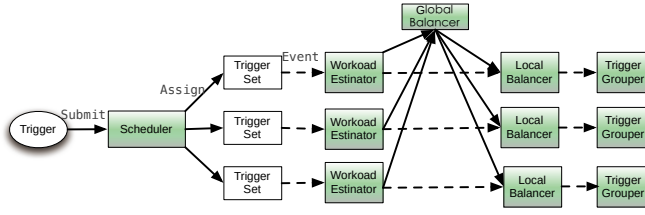


Fig. 3. Workflow of TSSA

become a group as the basic unit of transfer. As trigger in the event-driven clusters may be short second or sub-second task, we divide triggers invoked into groups to avoid repeatedly network transmission and reduce pressure.

Fig.?? shows the workflow of TSSA. A application is structured as a series of triggers. When a application is submitted, firstly Scheduler will assign all triggers to Trigger Workers where the relevant datasets of triggers set. And then Trigger Workers save their own triggers into Trigger Set. Once the relevant datasets are updated, trigger will be invoked and be add into trigger waiting queue. Workload Estimator automatically computes the workload of Trigger Worker. Trigger Worker send itself workload to Global Balancer in Trigger Master. Global Balancer in Trigger Master contains the map with Trigger Worker id and its workload. Global Balancer also computes idle rate of cluster q to determine

According to the decision of Global Balancer, Local Balancer coordinates and redistributes triggers invoked. For example, Local Balancer of the light-workload worker will notice a high-load worker transfer instructions. The high-load worker receives transfer instructions, selects one or more trigger to structure a *TriGroup*, and sends *TriGroup* to light-workload worker.

IV. EVALUATION

We are going to evaluate TSSA in term of performance and scaling. The performance metric is measured as the speedup over default Domino and an ideal load balancer. The scaling metric is measured as a different number of cluster nodes.

A. TSSA on Domino

In order to evaluate TSSA using a realistic workload, we ported TSSA to Domino by writing a Domino scheduling plugin. Domino is designed and implemented in JAVA and integrates with HBase storage system [7]. The Domino instance runs along with the HBase instance on each server. Each Trigger Worker contains a Event Detector to detect updates on HBase table and Action Fun to execute triggers. Its source can be found at <https://github.com/daidong/DominoHBase>.

B. Experimental Setup

To measure TSSA, we have built a Domino cluster, which has one master and 12 machines, each of which has Xeon dual-core processor and 6GB memory. All machines are connected by a common Gigabit Ethernet switch.

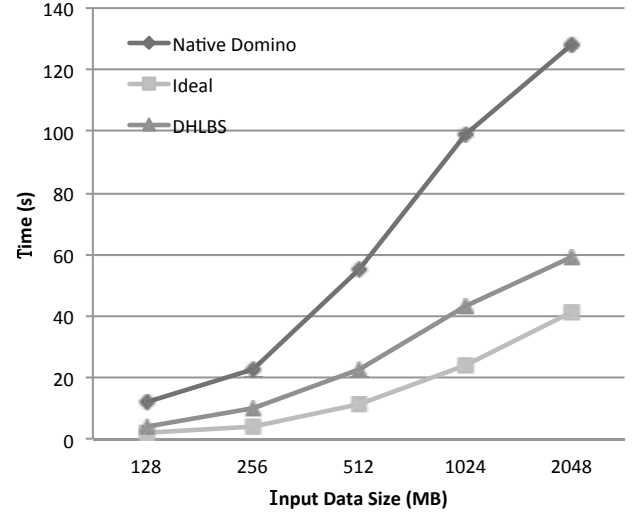


Fig. 4. WordCount under different input data sizes.

We measure TSSA performance on state-of-the-art distributed applications PageRank, and WordCount. In Domino, PageRank application contains two triggers: *PageRankDist* and *PageRankSum*. Any change on rank value of a page will trigger *PageRankDist* to recalculate rank values for all its relevant out-edge pages and write these changes to HBase. Then duo to these changes *PageRankSum* of such relevant out-edge pages will be invoked to sum up all the rank values generated from in-edge pages and write the result back into HBase. PageRank will converge until the difference between new rank value and old rank value less than a tolerable error. For more information, you can refer to [6]. For different evaluations, we create a large number of static input datasets and different changing data sets for both applications.

C. Performance Evaluation

Firstly, we designed this test to evaluate TSSA performance. Native Domino employs HBase's load balancer. However, the load balancer of HBase focuses on coordinating the data capacity of datanodes. To evaluate its performance, we also compare TSSA with an ideal load balancer that always transfers all triggers with zero I/O time and assumes data locality for all triggers. To compute the ideal completed time, we first run applications on a cluster with a single worker, and then divided this execution time by the number of workers in test cluster.

We evaluated WordCount on a synthetic dataset. The overall input datasets are 20 GB; each time we added a new dataset (from 128 M to 2 G) as the changing set. We can see that TSSA achieves slightly better performance than native Domino in Fig 4 and Fig 5. Performance speedup is lower at job of 2 G input size than other jobs because the changed dataset is fairly good duo to native load balancer of HBase, and thus trigger distribution also may be fairly good. However, in practice, when many jobs share the same cluster, dataset is not be evenly distributed. TSSA performed lower than an ideal scheme since an ideal scheme insures all triggers with data locality.

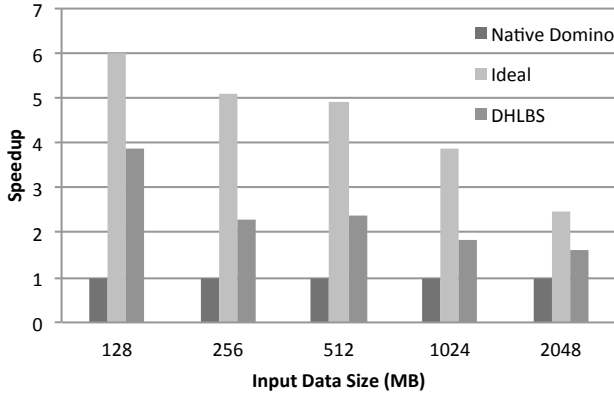


Fig. 5. Performance speedup with native Domino for WordCount under different input data sizes.

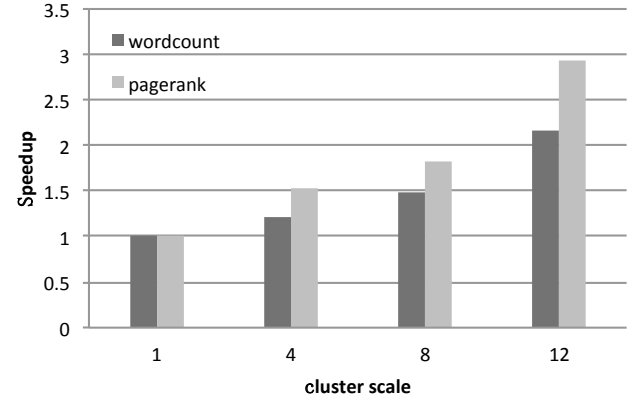


Fig. 8. Applications performance while the number of cluster nodes increases

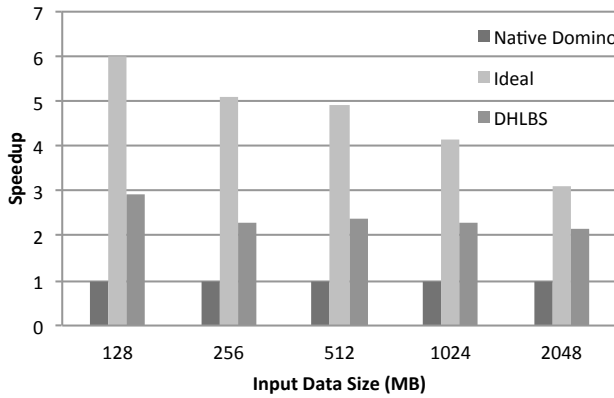


Fig. 6. PageRank under small different changed sizes.

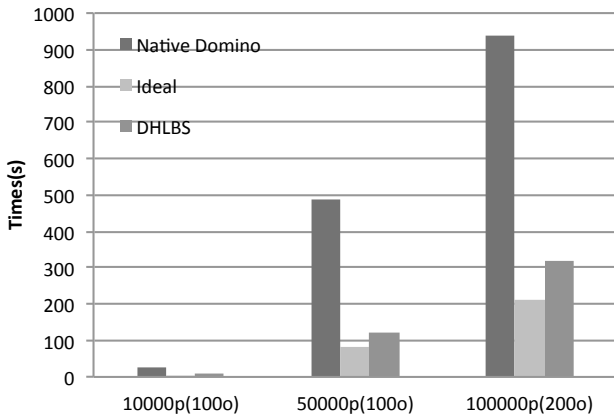


Fig. 7. PageRank under big different changed sizes.

Fig 6 and Fig 7 show PageRank runs on a synthetic web graph with increasing input size (10^6) with increasing input size ranging from 10^3 to 10^5 pages. PageRank is iterative. We labeled each input data set in the x-axis: p means the page number and o means the out-edges of each page. Fig 7 suggest that as the input changing data increasing, TSSA significantly

outperforms native Domino. The performance gain is mainly based on maintaining all workers busy as much as possible.

D. Scaling Performance

We explored how well TSSA scales as the number of nodes increasing. Fig 8 shows both application performance speedup as the number of nodes increases from 1 to 12 for the default input size in 12 nodes. We can see that TSSA archives better performance in comparison with native Domino. In small cluster size, the speedup is smaller than big cluster size. Because the load balancer of HBase can achieve relatively fair distribution. Of course, the only way to conclusively evaluate TSSA performance at scale will be to deploy it on a large cluster. But in light of this trend, our experiments suggest that our method performs well at cluster size increases to tens of thousands of cores.

V. RELATED WORK

In the event-driven clusters, dynamic load balancing is necessary since when triggers to be invoked is unpredictable. Decentralized load balancing algorithm is much more complicated than centralized methods. Sender Initiated Diffusion (SID) [12], [14] makes use of near-neighbor load information to apportion surplus load from heavily loaded processors to underload neighbors in the system. Receiver Initiated Diffusion (RID) [12], [14] is the converse of the SID policy. Random strategy is to select a remote node randomly and transfer the job [25]. Probing strategy is to select a random subset of nodes and find a suitable destination node, such as Sparrow [16]. Gradient Model [13] employs a gradient map of the proximities of underload processors in the system to guide the migration of tasks between overloaded and underload processors. These algorithms have conflicting scheduling problem: multiple local schedulers sampling in parallel may experience race conditions. However, our method reduces the problem by centralized group scheduling decisions.

Another decentralized strategy, hierarchical balancing is an asynchronous, global, approach which organizes the system into a hierarchy of subsystems. Willebeek [12] shows a binary-tree hierarchical balancing method, which is to maintain all

nodes' load difference within a prespecified amount. Dynamic biasing algorithm [11] uses biasing process to achieve load balancing. Round Robin algorithm [11] distributes arrival tasks orderly on the groups at the specific time slices. Also each local load balancer distributes arrival tasks of group to the nodes at specific time slices. Our method focuses on lightly loaded workers to make no node in idle state, instead of roughly equalizing the workload among all nodes. It can reduce the number of network transmission. And our method divides triggers into group as a transfer unit.

VI. CONCLUSION

Due to characteristics of the event-driven model, it is necessary to research load balancing algorithm. To the end, we present a two step scheduling scheme for the event-driven cluster frameworks. It avoids the performance bottleneck compared with centralized method. And it has lower network I/O than decentralized strategy. Since triggers may be second or sub-second task, we proposed a trigger group mechanism.

REFERENCES

- [1] Dean, Jeffrey and Ghemawat, Sanjay, MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008.
- [2] Isard, Michael and Budiu, Mihai and Yu, Yuan and Birrell, Andrew and Fetterly, Dennis, Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review, 2007
- [3] McCarthy, Dennis and Dayal, Umeshwar, The architecture of an active database management system. ACM Sigmod Record, 1989
- [4] Peng, Daniel and Dabek, Frank, Large-scale Incremental Processing Using Distributed Transactions and Notifications. OSDI, 2010
- [5] Mitchell, Christopher and Power, Russell and Li, Jinyang, Oolong: asynchronous distributed applications made easy. Proceedings of the Asia-Pacific Workshop on Systems, 2012
- [6] Dai, Dong and Li, Dino: Trigger-based Programming Framework in Cloud. Proceedings of the 7th workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture conjunction with ISCA. 2013.
- [7] Dai, Dong and Chen, Yong, Domino: an incremental computing framework in cloud with eventual synchronization. Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. 2014
- [8] George, Lars, HBase: the definitive guide, 2011
- [9] Alakeel, Ali M, A guide to dynamic load balancing in distributed computer systems. International Journal of Computer Science and Information Security, 2010
- [10] Waraich, Sandeep Singh, Classification of dynamic load balancing strategies in a network of workstations. Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on, 2008
- [11] Barazandeh, Iman and Mortazavi, Seyed Saeedolah, Two hierarchical dynamic load Balancing algorithms in distributed systems. Computer and Electrical Engineering, 2009. ICCEE'09. Second International Conference on, 2009
- [12] Willebeek-LeMair, Marc H and Reeves, Anthony P. Strategies for dynamic load balancing on highly parallel computers. Parallel and Distributed Systems, IEEE Transactions on, 1993
- [13] Lin, Frank C. H. and Keller, Robert M. The gradient model load balancing method. Software Engineering, IEEE Transactions on, 1987
- [14] Eager, Derek L and Lazowska, Edward D and Zahorjan, John, A comparison of receiver-initiated and sender-initiated adaptive load sharing. Performance evaluation, 1986
- [15] Zomaya, Albert Y. and Teh, Yee-Hwei, Observations on using genetic algorithms for dynamic load-balancing. Parallel and Distributed Systems, IEEE Transactions on, 2001
- [16] Ousterhout, Kay and Wendell, Patrick and Zaharia, Matei and Stoica, Ion, Sparrow: distributed, low latency scheduling. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013
- [17] Gupta, Rajeev and Gopinath, Prabha, A hierarchical approach to load balancing in distributed systems. Distributed Memory Computing Conference, 1990., Proceedings of the Fifth, 1990
- [18] Chowdhury, Shyamal, The greedy load sharing algorithm. Journal of Parallel and Distributed Computing, 1990
- [19] Eager, Derek L. and Lazowska, Edward D. and Zahorjan, John, Adaptive load sharing in homogeneous distributed systems. Software Engineering, IEEE Transactions on, 1986
- [20] Tanenbaum, Andrew S and Van Renesse, Robbert, Distributed operating systems. ACM Computing Surveys (CSUR), 1985
- [21] Wang, Yung-Terng and Morris, Robert J. T, Load sharing in distributed systems. Computers, IEEE Transactions on, 1985
- [22] Ahmad, Ishfaq and Ghafoor, Arif, Semi-distributed load balancing for massively parallel multicomputer systems. Software Engineering, IEEE Transactions on, 1991
- [23] Ni, Lionel M. and Hwang, Kai, Optimal load balancing in a multiple processor system with many job classes. Software Engineering, IEEE Transactions on, 1985
- [24] Zhou, Songnian, A trace-driven simulation study of dynamic load balancing. Software Engineering, IEEE Transactions on, 1988
- [25] Eager, Derek L. and Lazowska, Edward D. and Zahorjan, John, Adaptive load sharing in homogeneous distributed systems. Software Engineering, IEEE Transactions on, 1986
- [26] Mitzenmacher, Michael, The power of two choices in randomized load balancing. Parallel and Distributed Systems, IEEE Transactions on. 2001



Yu Chen works in China Unionpay. He is the general manager of China Unionpay Technology Division. His research interests include various aspects of multicore and distributing systems. He has lead many country projects.



Mingming Sun works in China Unionpay. She is a doctor, graduated from University of Science and Technology of China. Her research interests include issues related to parallel algorithm, resource management of cloud computing, and task scheduling algorithm of MapReduce clusters. She has published research papers at international journals, conference proceedings.