Enhancing Sample-based Scheduler with Collaborate-state in Big Data Cluster

Chunliang Hao [✦✧] Celia Chen [★] Jie Shen [☆] Mingshu Li [✦] Barry Boehm [★]

[✦]Institute of Software, Chinese Academy of Science, Beijing, China
[★]Center for Systems and Software Engineering, University of Southern California, USA
[☆]Department of Computing, Imperial College London, UK
[✧]University of Chinese Academy of Science, China
Email: chunliang@nfs.iscas.ac.cn, qianqiac@usc.edu, js1907@imperial.ac.uk,
mingshu@iscas.ac.cn, boehm@usc.edu

Abstract

Sample-based scheduler design has become an emerging research topic for its high scalability and simple scheduling process in today's big data cluster. One major limitation of such design is its lack of global cluster knowledge, which leads to sub-optimal decisions. Some cutting edge schedulers solve this issue by deploying an extra centralized component in the cluster to capture the real-time cluster state and inform all schedulers. However, such solution is with high cost and low scalability. As an alternative, we introduce the Collaborated-Cluster State(CCS) technique in this paper. CCS is a low cost solution that merely harms the scalability of sample-based design, while achieving similar performance gain as ECC. Experiments with Google and Yahoo production trace both show that CCS under most scenarios can keep up with ECC's performance while reducing 87.7% (in Google trace) and 73.9% (in Yahoo trace) of communications.

1 Background and Introduction

Sample-based scheduling is currently one of the most promising branches of distributed scheduling. It is both fully distributed and low-latency. Currently, the cutting edge sample-based schedulers adopt the batch-probing sampling process proposed by Sparrow. The batch-probing sampling process contains following key steps: 1. In order to schedule a t-task-job, a scheduler randomly samples $2t$ nodes from the cluster (the ratio of sampling of 2 follows the power of two law[1], but could also be changed to other value). 2. The scheduler sends each selected node a probe request. 3. Each probe is then queued in the worker node to serve as task reservation. 4. When a task reservation is at the head of the queue and is ready to execute, the worker replies to the scheduler and gets a task to run. 5. Once the scheduler sends all t task to run, it cancels all remaining reservations for the job by notifying according workers. 6. Worker will notify the scheduler of the completion of the assigned task. More details could be found in the Sparrow paper[2].

DOI reference number: 10.18293/SEKE2017-136

One crucial problem of these sample-based schedulers is their lack of global knowledge of the cluster status. During each decision process, a scheduler can only communicate with a very small number (e.g. $2t$ in above example) of sampled worker nodes and makes decision based on information gathered from those nodes.This limits the sample-based scheduler and leads to sub-optimal task placement decision.

An example of sub-optimal decisions made by sample-based schedulers is shown in Figure 1. In this example, the scheduler has randomly sampled two busy worker nodes, hence the task under schedule will experience queue waiting no matter which worker is chosen. However, there are six available worker nodes elsewhere in the cluster, this queue waiting could have been avoided if the scheduler had that information. Such a decision is considered sub-optimal since there are better decisions that existed and should be found in the cluster. However, for most cases scheduler itself can't realize a sub-optimal decision is been made; for instance scheduler in both Figure 2 and Figure 1 will consider themselves in the same status.

Some cutting edge scheduler mitigate this problem by using a centralized software component to synchronize cluster status and push collected global cluster knowledge to each sample-based scheduler (refer to as EXCC).

This centralized component can either be a share-state master, which is specifically designed to synchronize task and resource status with all worker nodes, or an independent centralized scheduler, which is responsible to schedule and inform the sample-based schedulers at the same time. Either way, the centralized component has to synchronize with all worker nodes to capture the real-time status of the cluster and then inform all the sample-based schedulers accordingly.

Although this EXCC solution have proven to effectively reduce sub-optimal decision and improve scheduling precision, EXCC is very expensive to implement. Since the centralized component is required to synchronize with all workers in cluster in order to collect global cluster information, the component itself becomes a potential system bottleneck, which is the pri-

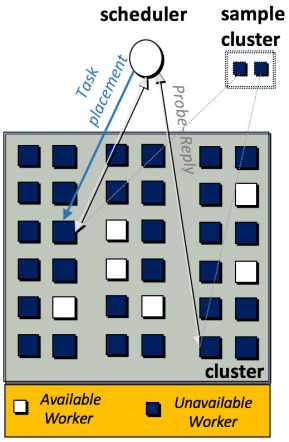

Figure 1: A sub-optimal task placement decision.

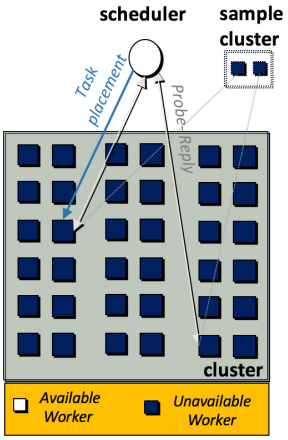

Figure 2: A good task placement decision.

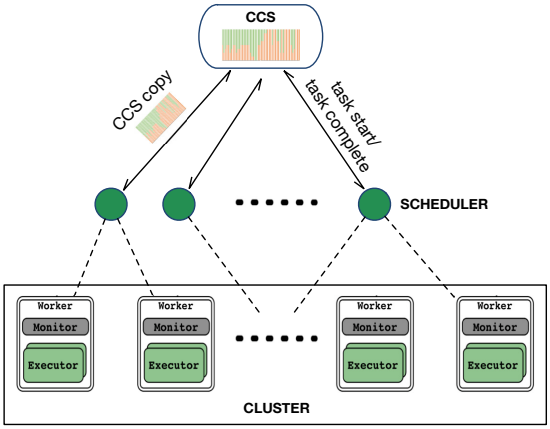

Figure 3: Overview of sample based scheduling with CCS

mary goal for sample-based schedulers to avoid. Moreover, a large amount of extra communications is needed to capture the real-time status of cluster.

In this paper, we propose collaborated-cluster-state (CCS), an alternative design that is low-cost and scalable, which allows the scheduler acquire global knowledge while keeping the simplicity of sample-based scheduling design.

2 Collaborated-Cluster-State

In this paper, we introduce a novel technique, collaborated-cluster-state (CCS), which provides sample-based schedulers with global cluster knowledge through scheduler collaboration. CCS keeps track of all occupied and available resource status by receiving updates from schedulers only. Each scheduler packs task start and complete information along with necessary task details together and send it to CCS periodically. CCS also periodically pushes its current accumulated knowledge to all schedulers. This pushed copy of CCS knowledge can provide each scheduler with advice on which workers should be probed and which should not.

In CCS, cluster resource information is stored in the form of an array of size N, in which N represents the number of worker nodes in the cluster. Each element in the array caches the resource status of the corresponding worker node in the cluster in a tuple $(\vec{ra}, \vec{ro})$, where $\vec{ra}$ denotes the available resource quantity in a worker node; $\vec{ro}$ denotes the occupied resource quantity in a worker node. For instance, $CCS[10] = (3, 4, 500, 5, 12, 200)$ means that the 10^{th} worker node in the cluster has 3 core, 4GB of DRAM and 500GB of Flash available and 5 core, 12GB of DRAM and 200GB of Flash occupied.

The overall design to maintain and use CCS knowledge in sample-based scheduling is shown in Figure 3.The CCS component needs to be initialized before each scheduler starts to function. Upon initialization, the CCS component reads in worker registration information from the cluster daemon to acquire the maximum resource capacity of each worker node and marks them as available.

After initialization, CCS component first waits for incoming messages from each scheduler. When a message arrives, CCS unpacks the message to recover the task start or complete information. Each piece of information is represented as one tuple. CCS then processes these tuples one by one. For each tuple, the component reads in the following information: whether the tuple represents task start or complete; the worker nodes of which each task runs on; the amount of resources each task claims. CCS then finds that worker node in its data and updates the amount of available and occupied resources accordingly. CCS also pushes a copy of the cluster state knowledge to all schedulers periodically. The interval of push, ω, is preset, where smaller ω value makes copies in each scheduler more precise but requires more communication while bigger ω value leads to the opposite situation. Each CCS copy also has an expire period β, to prevent the failure of CCS component.

Each scheduler receives a CCS copy and only keeps the latest version. At the beginning of each scheduling process, instead of choosing sample target at complete random, it checks with its own CCS copy first. Suppose $2t$ workers need to be selected for a t-task-job and each task claims resources $\vec{rc}$, the scheduler finds in its CCS copy for $2t$ worker that has $CCS[worker].\vec{ra} \geq \vec{rc}$. If more than $2t$ workers are found to be qualified, the scheduler chooses workers with most available resource by default. If less than $2t$ qualified workers are found, the scheduler chooses from the rest of workers randomly.

After sample target is chosen, the rest of decisioning process follows the common batch-probing process. The scheduler probes towards selected workers and places task reservations. When the scheduler is notified by any worker that one reservation is ready to execute, the scheduler places a task with that worker. It keeps doing so until all t tasks are placed. Once all tasks are placed, it cancels all the leftover reservations in the cluster. When the scheduler places task towards a worker, it caches a tuple $[1, workerID, rc]$ locally. When the sched-

Table 1: Number/distribution of job and task in trace

	Google(2011)	Yahoo(2011)
Number of Jobs	506.4k	24.2k
Number of Tasks	17889.7k	968.3k
% Jobs ed $\leq$ 1000s	89.0%	96.6%
% Jobs ed $\leq$ 100s	42.7%	36.4%
% Jobs ed $\leq$ 10s	0.0%	2.7%
% Tasks ed $\leq$ 1000s	69.7%	84.6%
% Tasks ed $\leq$ 100s	7.6%	54.0%
% Tasks ed $\leq$ 10s	0.0%	20.2%

uler is notified that a task is completed, the scheduler caches a tuple $[0, workerID, rc]$ locally. The first variable in tuple represents task start/complete, where 1 represents start and 0 represents finished. The second variable is the ID of the current worker, and the last variable is the amount of resources the task claims. Periodically (with an fixed interval γ) it packs all cached tuples together and sends them to CCS. Each tuple will be sent only once.

3 Experiments

3.1 Methodology

Workload: Google trace and Yahoo trace are used in the evaluation as a representation of today's production workloads in big data cluster. The Google trace used in this paper[3][4] is publicly available. Cleaning the trace by removing failed or invalid records resulted in more than 500k heterogeneous jobs. Task runtime also varies within each job. For Yahoo trace, we gathered centroid values for task duration and average number of task per job from the trace description[5][6], then used them as scale parameters in exponential distribution to generate the complete trace. The job/task distribute of Google trace and Yahoo trace are listed in Table 1.

Metrics: In this paper, we used three metrics to evaluate CCS against some baseline techniques. We first evaluated how many sub-optimal decisions a sample-based scheduler could avoid with CCS and then collated the job runtime results of CCS and of the baseline techniques in Google and Yahoo trace. 50^{th} and 90^{th} percentile job runtime of the workload were used in this experiment. Finally we compared the communication costs of CCS against the baseline schedulers.

Baseline Schedulers and Simulator: There are two baseline schedulers used in this study: Sparrow and EXCC. Sparrow was used to represent the cutting edge sample-based scheduler without any global cluster knowledge. In the experiment, we use the event-based Sparrow simulator from its own open-source project[2] and further augmented this event-based scheduler for the evaluation of CCS. It was used as the lower bound in performance and communication cost. EXCC represented the high-cost solution that uses a centralized component to obtain a more precise global cluster state for the sample-based schedulers. It was used to represent the upper bound in sub-optimal decision and lower bound in job run-time. We abstracted the EXCC used in Tarcil[7] scheduler for the evaluation of CCS.

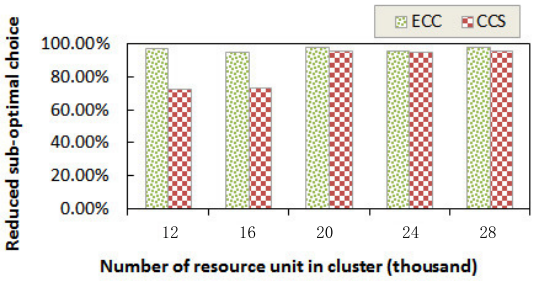

Figure 4: The percentage of reduced sub-optimal decision using CCS and EXCC, comparing to Sparrow, Google trace

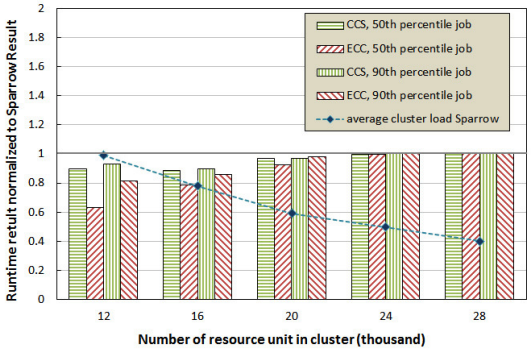

Figure 5: Google trace runtime results in five different cluster; evaluating Sparrow, CCS and EXCC. Both CCS and EXCC results are normalized to Sparrow.

Cluster setting: We changed the size of the cluster to achieve different cluster utilizations and simulate the industrial workloads. The number of schedulers deployed in the cluster was set to 10. For CCS, the parameter ω was set to 10 second and γ was set to 5 second. Probe ratio was fixed at 2 following the Power of Two Law[1].

3.2 Google Trace Results

The sub-optimal decision statistics from Google Trace are shown in Figure 4. Both CCS and EXCC effectively reduced most of the sub-optimal decisions in sample-based scheduling. Especially when the cluster was under medium and low loaded situation (20k,24k,28k slots cluster), almost all of the sub-optimal decisions were avoided. In high cluster, CCS made sub-optimal decisions mostly because of its imprecise global knowledge; EXCC because of conflicts.

The result of job runtime of Google trace is shown in Figure 5. We normalized the runtime results of CCS and EXCC to Sparrow for better comparison. In this case, smaller results indicated better performance improvement. As shown in the figure, the major improvement of both CCS and EXCC was in high and extreme loaded (16k and 12k slots) clusters. The difference between EXCC and CCS was small except in extreme loaded cluster, where EXCC outperformed CCS by about 20%.

The overall communication count in/out of CCS component

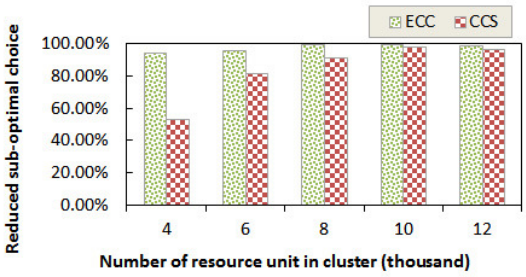

Figure 6: The percentage of reduced sub-optimal decision using CCS and EXCC, comparing to Sparrow, Yahoo trace

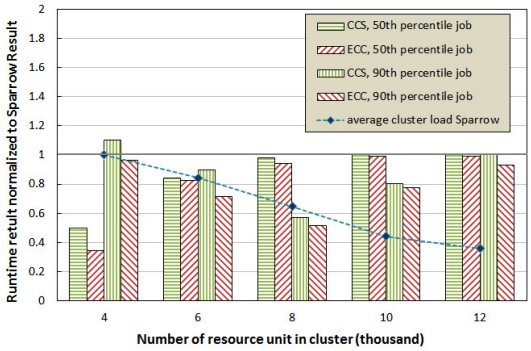

Figure 7: Yahoo trace runtime results in five different cluster; evaluating Sparrow, CCS and EXCC. Both CCS and EXCC results are normalized to Sparrow.

in above experiments was roughly 7.5 million in total, while the EXCC required about 60.6 million in total. When the throughput of workload and cluster size increases (as expected in future's big data cluster), the communication needed by EXCC increases accordingly while CCS stays the same.

3.3 Yahoo Trace Results

Yahoo trace results showed very similar trends as Google results. Compared to Sparrow, both CCS and EXCC reduced almost all sub-optimal decisions. CCS had a very similar performance as EXCC, as shown in figure 6.

Considering runtime results, the results showed that both CCS and EXCC reduced about 20% job runtime of Sparrow in high loaded cluster(6k slots) and cut 90th percentile runtime of Sparrow in half in medium loaded cluster(8k slots).

Similar to Google trace results, CCS needed much less communication than EXCC. The total amount of communication in/out of CCS is about 0.55 million and EXCC is about 2.11 million.

The results from both Google and Yahoo trace show that CCS achieved similar performance using far less (respectively 86.7% and 73.9% less) communication cost. The impact of latency (a 10 second pushing delay and a 5 second synchronization delay) in CCS has a limited impact on the final runtime result, as we expected.

In above experiments, while EXCC had to synchronize with thousands of work nodes, CCS only needed to synchronize with 10 worker nodes. Compared to EXCC, the packing of updates in CCS changed synchronization frequency from per-task frequency to a fixed rate, which in each experiment was from about 1 per second in EXCC to 0.2 per second in CCS.

4 Related Work

Among existing sample-based methods, Sparrow is one of the most popular and represented work with the batch-probing technique. In this paper, we have introduced how to implement CCS through augmenting Sparrow and also used Sparrow scheduler as the baseline in our experiment. Tarcil is also a distributed, sample-based scheduling approach. It uses extra centralized components to combine the benefit of share-state design and sample-based design. We have compared the performance of such EXCC method and our proposed CCS method.

Another direction of providing sample based scheduler with global knowledge is the hybrid design. Eagle[8] proposed a design that uses a EXCC as a scheduler for long jobs and also as a source of global information, which notifies sample based schedulers about the placement of long tasks in cluster.

5 Conclusions

Collaborated-cluster State (CCS) is a novel technique to improve the precision of sample-based schedulers by providing them with global cluster knowledge. Comparing to existing techniques that serve the same purpose, it requires much lower communication cost while preserving the scalability of sample-based design.

Acknowledgement

This work is financially supported by the Strategic Priority Research Program of the Chinese Academy of Science (No. XDA06010600), as part of the DataOS project.

References

[1] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE TPDS*, vol. 12, no. 10, pp. 1094–1104, 2001.

[2] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *SOSP*, 2013, pp. 69–84.

[3] J. Wilkes. More google cluster data. [Online]. Available: http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html

[4] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC*, 2012, pp. 7–13.

[5] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems," in *VLDB*, 2012, pp. 1802–1813.

[6] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*, 2011, pp. 390–399.

[7] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: reconciling scheduling speed and quality in large shared clusters," in *SoCC*, 2015, pp. 97–110.

[8] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *SoCC*, 2016.