Tiresias: low-overhead sample based scheduling with task hopping

Chunliang Hao[†,‡], Jie Shen[§], Heng Zhang[†,‡],Yanjun Wu[†], Mingshu Li[†]
[†]Institute of Software, Chinese Academy of Science, Beijing, China
[‡]University of Chinese Academy of Science, Beijing, China
[§]Department of Computing, Imperial College London, London, UK
Email: chunliang@nfs.iscas.ac.cn, js1907@imperial.ac.uk, {hengzhang, yanjun, mingshu}@iscas.ac.cn

***Abstract*—Sample based distributed scheduling methods have been shown to be promising lower overhead alternatives to their centralized counterparts. These methods can make fast decisions based on information gathered from just a small number of worker nodes instead of the whole cluster. Most recent works in the field tend to adopt a combination of probe actions and worker-end queues in their design. However, as individual worker nodes are becoming increasingly powerful thanks to the rapid hardware evolution, we argue that one-node sampling is now a viable choice. Specifically, we show that it is now possible to achieve even lower scheduling latency by latency by abolishing probes and worker-end queues altogether. With this insight, we introduce Tiresias, a low overhead distributed scheduler based on one-node sampling and a novel task hopping mechanism. Comparing to Sparrow's approach, experiment on Google trace shows Tiresias could reduce 20% and 60% of Sparrow's 50th percentile and 90th percentile job runtime, respectively. In addition, our experiment also shows Tiresias is especially effective in reducing the delay of small jobs in non-highly loaded clusters.**

I. Introduction

Centralized scheduling methods have been widely used in production clusters in the past few decades [**?**]. Serving as the single connection between jobs and workers, a centralized scheduler can make highly optimal scheduling decisions and is capable of enforcing complex global scheduling policies [**?**]. However, as the scale of both cluster and workload keeps expanding, centralized scheduler is increasingly more difficult to maintain and may become a potential performance bottleneck [**?**]. Furthermore, with the trend that high throughput, sub-second jobs are taking up larger fraction of production workload [**?**], scheduling overhead is becoming an important design consideration.

To address these problems, sample based scheduling methods have been proposed [**?**]. The core idea of such approaches is to make scheduling decisions based on the information gathered from only a small subset of nodes sampled from the cluster. In doing so, the decision-making process becomes much faster, resulting in low scheduling overhead, which is crucial to small interactive jobs. To be able to sample multiple nodes at a time, most existing sample based scheduling methods share the same design of using probes and worker-end queues [**?**]. However, this design choice comes with its own limitations [**?**], notably the sub-optimal job latency that we aim to address in this work.

Thanks to the rapid development in computational infrastructure, we argue that nowadays scheduler may sample only one node at a time, yet still achieve an acceptable level of optimality in scheduling decisions made under most circumstances. In this case, probe action and worker-end queue are no-longer mandatory features of the design. This, in turn, allows us to explore simpler and faster sample based methods. With this insight, we introduce Tiresias, a novel sample based scheduler no longer relying on probes and worker-end queues. Instead, Tiresias is based on direct task placement and a task hopping mechanism. As a result, we show the combination of these two techniques could decrease the runtime of all jobs and significantly reduce the delay of short jobs.

The main contributions of this work are as follows: 1. We show that under current cluster computing context one-node sampling is now becoming a viable option for sample based scheduler. Accordingly, we introduce direct task placement technique to further decrease scheduling overhead. 2. To compliment direct task placement, we further propose a task hopping mechanism to efficiently utilize system resource and in the same time to reduce the occurrences of sub-optimal scheduling decisions. 3. We implement Tiresias, a low overhead distributed scheduler based on direct task placement and task hopping. Based on our quantitative evaluation, we show Tiresias is capable of achieving lower scheduling overhead comparing to existing approaches.

II. Motivation

Although the probe-queue design is commonly adopted by most existing sample based schedulers, it is not without limitations. Notably, the use of worker-end queue may lead to various kinds of sub-optimal decisions, lowering the performance when handling interactive jobs [**?**] [**?**]. In addition, the probing procedure incurs a fixed scheduling overhead, which we suggest could be reduced.

Recent development in hardware capability has enabled some worker nodes to support 32 or more concurrent threads and more than 100GB of memory [**?**]. These powerful nodes are capable of holding many tasks in parallel, especially for small interactive ones. In this case, sampling only one node during decision making can still give a good chance of finding the required resource. For instance, when scheduling tasks in an 80%-utilized cluster with 32 slots per node, there is

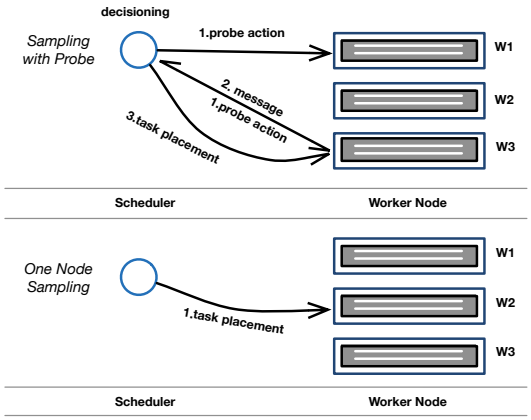


Fig. 1. Multi-node sampling + probe-queue design (top scheduler) vs. one-node sampling + direct task placement (bottom scheduler).

a 99.9208% chance that a randomly selected node will have at least one available slot.

Reducing the sample size to one node allows us to eliminate the probing procedure altogether and, in turn, to adopt a simpler scheduler design. Namely, instead of having to send out probe message and wait for the sampled node's reply, the scheduler can simply assigns the task directly to node since there is no longer a need to choose between multiple nodes.

A comparison between the probe-based design and the aforementioned *direct task placement* method is illustrated in Fig. 1. The scheduler illustrated on the top row performs two-node sampling, which requires three communication steps when scheduling a task: step 1) the scheduler sends probe messages to the randomly sampled nodes; step 2) the scheduler receive status information from at least one node; step3) based on the received information, the scheduler selects an optimal node to assign the task. The usage of probe in step 1, the information type and timing of response in step 2, and the decision-making logic in step 3 may vary in different scheduler designs. However, this three step procedure is shared among the approaches. In comparison, the direct task placement method, as illustrated on the bottom row in Fig.1, only requires a single step: the schedule simply assigns the task to a randomly selected node. Apart from the decrease in communication overhead, the method also requires very little computation within the scheduler itself, thus further reduces scheduling latency.

Even with powerful worker nodes, direct task placement may still fail (when the selected node has no available slot at all) from time to time, especially when the cluster is under extreme workload or poor load balancing. When this happens, putting the tasks into a worker-end queue is a straightforward but often sub-optimal solution since there is usually a high probability that another node in the cluster may have an available slot allowing the task to be executed immediately. In other words, when a fully occupied node receives a new

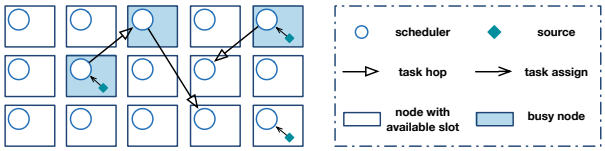


Fig. 2. The Overall architecture of Tiresisas, demonstrated on 15 node.

task, passing the task to another randomly selected node may allow the task to be executed faster, thus reduce task delay. Therefore, it would not only be possible, but also beneficial to replace the worker-end tasks queues with a novel *task hopping* mechanism.

Based on direct task placement and task hopping, we propose Tiresias, a simple and effective low overhead distributed scheduler.

III. Tiresias

Tiresias's architecture and its scheduling procedure are illustrated in Fig. 2. Functioning independently from each other, instances of the Tiresias's scheduler are deployed onto every node in the cluster. Since Tiresias does not distinguish between master node and worker node, it can tolerate malfunction in any single node.

The core decision process of the Tiresias scheduler is quick and simple: **1.** The scheduling process starts when task k arrives at node d;**2.** The scheduler tries to find an available slot on node d to execute task k locally;**3.** If node d is fully occupied, the scheduler passes the task to another randomly selected node in the cluster.

The transferring of task from one scheduler to another is referred to as 'task hopping'. In Tiresias, a task may hop through the entire cluster following a random walk pattern until it reaches the node allowing it to be executed. Specifically, when a task arrives at a fully occupied node, the scheduler will simply pass the task to another randomly selected node (as shown in Fig. 2). In order to account for potential failure of scheduler, node, and / or network connection, the scheduler will later receive an asynchronous acknowledgement from the hopping destination.

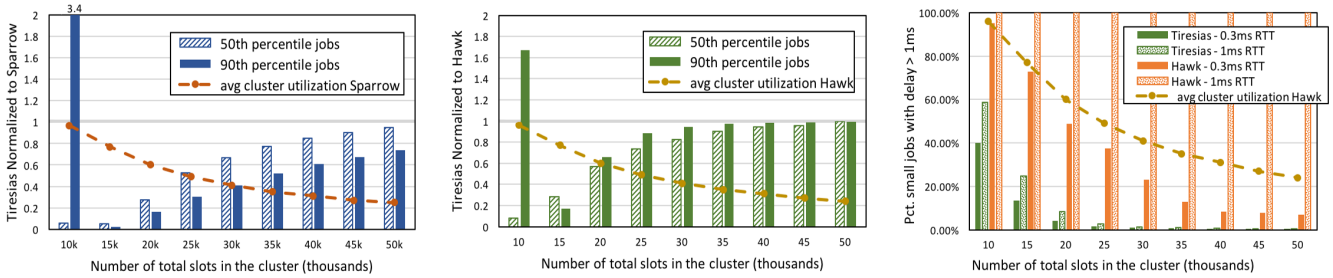
To prevent excessive hopping, Tiresias imposes a hop limit to all tasks. When a task reaches its hop limit, it will be marked as waiting for retry. A retry notification will then be sent to task's entry point, allowing the task to be either dropped or to be retried later.

Tiresias scheduler treats all incoming tasks equally, regardless of their source. Specifically, tasks transferred from other nodes are treated in the same way as those received from applications. This simplicity guarantees Tiresias's processing speed and throughput.

IV. Evaluations

A. Methodology

Workloads: In this work, we used the publicly available Google trace [**?**] [**?**] as the benchmark workload for evaluation.



(a) All jobs, runtime, Tiresias/Sparrow (b) All jobs, runtime, Tiresias/Hawk (c) Small jobs, delay, Tiresias/Hawk

Fig. 3. Google trace, simulation, comparing Tiresias to Sparrow and Hawk

Cleaning the trace by removing failed or invalid information resulted in a total of 506.4k valid jobs. Since many jobs do not have reducers, only mapper tasks were selected. Taking task parallelism into consideration, the jobs' duration were estimated by the duration of their longest task.

Simulation run: We compared the results of Tireias with Sparrow and Hawk. We used Hawk and Sparrow's own publicly available simulator to evaluate their performance. For Tiresias, we further augmented the event-based simulator. In simulation, each node was set to have 50 slots to represent the latest setting of production cluster (Sparrow and Hawk simulator analogues the performance of multiple slots containing independent queue by separated nodes). We used two settings of average RTT time for simulation, 1ms and 0.3ms. Local computation time such as decisioning time were not taken into account during the simulations.

Real cluster run: EC2 100-node cluster was used in the real cluster run test. We used a 5000 jobs subset of Goolge trace in this test. The subset was selected to match the original distribution of jobs according to estimated job duration. We then scaled down the duration of tasks by 1000x to make runtime proportional to Google trace. The number of task in each job was also scaled down to maintain the same ratio between the job's maximum task count and overall cluster slot count. In real cluster run, the job's mean inter-arrival time was changed to create different cluster load. We first found a biggest mean inter-arrival time that overload the 100 node in EC2 cluster, and then used it as the baseline. We gradually increased this mean inter-arrival time to create high, medium and low utilized situations.

B. Overall results on Google trace

What is the results for all jobs in Google trace?

Tiresias achieved the best 50th and 90th runtime when the cluster was not overloaded(i.e. 10k slots). As shown in Fig. 3(a), Tiresias surpassed Sparrow's 50th percentile runtime and 90th percentile runtime by 84% and 91% under high cluster utilization(i.e. 15k, 20k slots), 35% and 51% under medium cluster utilization(i.e. 25k - 35k slots), 10% and 33% under low cluster utilization(i.e. 40k - 50k slots), respectively. As shown in Fig. 3(b), Tiresias surpassed Hawk's 50th percentile runtime and 90th percentile runtime by 57% and 58% under high cluster utilization, 18% and 7% under medium cluster utilization, 2% and 0% under low cluster utilization, respectively.

What is the results for small interactive jobs?

As shown in Fig. 3(c), under 1ms RTT setting, Tiresias performed best in medium utilized(i.e. 30k, 35k slots) and low utilized(i.e. 40k - 50k slots) clusters. This is because majority of there tasks were executed after only one hop. In high utilized(i.e. 40k - 50k slots) clusters, tasks started to hop more than once, leading to longer delay. In comparison, Hawk results were all near 100%. Correspondingly, under 0.3ms RTT setting, in high utilized clusters Tiresias had less than 20% jobs with delay longer than 1ms, when Hawk had more than 40%. In clusters from 25k to 50k slots, Tiresias had less than 5% jobs with delay longer than 1ms, especially in 45k and 50k slots clusters when the number reduced to below 1%.

How many hops are required to place one task?

As shown in TABLE I, in both medium utilized(i.e. 25k - 35k slots) and low utilized clusters(i.e. 40k - 50k slots), the 90th percentile task hop count was always 1. The average hop count for task was within the range of 1.00 to 3.9. This means in each of these cluster most tasks were scheduled with very small delay. It also indicates in 30k-50k clusters the overall communication cost required by Tiresias was less than Sparrow and Hawk. In high utilized clusters(i.e. 15k - 20k), task hop count started to rise. The average hop count of 10-30 per task indicate a higher communication cost than Sparrow and Hawk. In overloaded cluster(i.e. 10k) Tireisas still functioned properly but with most task hopped more than once.

C. Real cluster run results

What is the EC2 cluster run results for all jobs?

As shown in Fig. 4, EC2 cluster results show similar trends as simulation results. Specifically, Tiresias performed the best when cluster was high utilized(i.e. 1.2x and 1.4x baseline average inter-arrival time), in which case Tiresias surpassed the 50th percentile and 90th percentile runtime of Sparrow by 18.5% and 61.3%, respectively. Under medium utilized situations(i.e. 1.6x and 1.8x baseline average inter-arrival time), Tiresias improved 3% and 32%, and under low utilized situations(i.e. 2x -2.4x baseline average inter-arrival time) 1.4% and 22.3%, respectively.

What is the EC2 cluster run results for small jobs?

TABLE I
GOOGLE TRACE, TIRESIAS, THE NUMBER OF HOPS FOR ONE TASK.

Total Cluster slots	10k	15k	20k	25k	30k	35k	40k	45k	50k
minimum task hop	1	1	1	1	1	1	1	1	1
10th percentile task hop	1	1	1	1	1	1	1	1	1
20th percentile task hop	2	1	1	1	1	1	1	1	1
30th percentile task hop	25	1	1	1	1	1	1	1	1
40th percentile task hop	82	1	1	1	1	1	1	1	1
50th percentile task hop	101	1	1	1	1	1	1	1	1
60th percentile task hop	101	3	1	1	1	1	1	1	1
70th percentile task hop	101	80	2	1	1	1	1	1	1
80th percentile task hop	102	101	2	1	1	1	1	1	1
90th percentile task hop	108	101	98	1	1	1	1	1	1
maximum task hop	501	301	129	111	109	106	106	105	105
Average Hops per Task	79.37	32.88	12.18	3.90	1.69	1.19	1.08	1.01	1.00
Average Cluster Utilization	95.45%	77.34%	60.05%	48.98%	40.85%	35.02%	30.65%	27.24%	24.52%

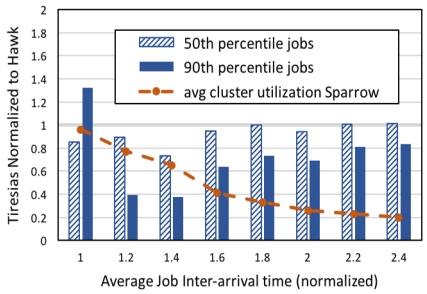


Fig. 4. EC2 cluster run, Tiresias normalized to Sparrow, duration of all jobs.

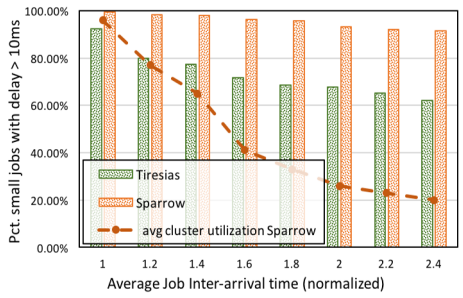


Fig. 5. EC2 cluster run, Tiresias and Sparrow, delay of small jobs.

In real cluster run, job runtime contains various source of cost that are not counted in simulation, including transport protocol overhead, local computing cost, network instability, and so on. As a result, job delay around 1ms becomes hard to observe. Hence in EC2 cluster run we only compare the proportion of small jobs(with estimated duration<100ms) that has delay larger than 10ms. Smaller results are preferred. As shown in Fig. 5, for these small jobs, Tiresias performed the best when the cluster is low utilized. In specific, when average job inter-arrival time is 2.4x baseline(cluster was 20% utilized), only 62.2% of small jobs in Tiresias had larger than 10ms delay, while Sparrow had 91.55%.

V. CONCLUSION

Tiresias uses direct task placement and task hopping for decision making. To the best of our knowledge, Tiresias is the first sample based scheduler without using probe messages and worker-end queues. Its simpler scheduling process significantly reduces job delay, especially in non-high-loaded cluster. In addition, our approach also improves job runtime thanks to the task hopping behaviour.

VI. 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. The work of Jie Shen has also been funded in part by the European Community Horizon 2020 [H2020/2014-2020] under grant agreement no. 645094 (SEWA). We thank all members in DataOS group for discussions and comments. We also thank Delgado Pamela and Florin Dinu for their support and advices.

REFERENCES

[1] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010, pp. 265–278.

[2] Apache hadoop capacity scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[3] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013, pp. 351–364.

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

[5] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015, pp. 499–510.

[6] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *EuroSys*, 2016, pp. 1–15.

[7] C. Hao, J. Shen, H. Zhang, X. Zhang, Y. Wu, and M. Li, "Sparkle: adaptive sample based scheduling for cluster computing," in *CloudDP*, 2015.

[8] Amazon elastic compute cloud. [Online]. Available: http://aws.amazon.com

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

[10] 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.