Cotask Scheduling in Cloud Computing

Yangming Zhao[1], Shouxi Luo[2], Yi Wang[3], Sheng Wang[1]
[1]University of Electronic Science and Technology of China
[2]Southwest Jiaotong University [3]Huawei Technologies Co., Ltd.

***Abstract*—Computing frameworks have been widely deployed to support global-scale services. A job typically has multiple sequential stages, where each stage is further divided into multiple parallel tasks. We call the set of all the tasks in a stage of a job a *cotask*. In this paper, we aim to minimize the average Cotask Completion Time (CCT) in cotask scheduling. To the best of our knowledge, there is no prior work on cotask scheduling for cloud computing. We propose the Cotask Scheduling Scheme (CSS), and take MapReduce as a representative of computing frameworks. CSS schedules cotasks following the Minimum Completion Time First (MCTF) policy, and we prove this problem is NP-hard. We formulate the model using the Integer Linear Programming (ILP), and solve it through an efficient heuristics based on ILP relaxation. Through real trace based simulations, we show that CSS is able to reduce the average CCT by up to 62.20% and 69.93% with traces from our testbed and from a large production cluster respectively.**

I. Introduction

Computing frameworks, such as MapReduce [1], Pregel [2] and Spark [3] have been widely deployed to support global-scale services such as web searching and social networking. Such a computing framework may run many jobs at any given time. A job typically has multiple sequential stages, where each stage is further divided into multiple parallel tasks [4]. The output of a stage often serves as the input of the following stage. The jobs in a data center are often scheduled centrally. For any task, the scheduler decides not only which compute node to assign this task to, but also when the node should execute the task.

The optimization goal of the scheduler is to reduce job completion time because shorter job completion time for data center requests leads to fast response time and better user experience. To reduce job completion time, it is critical to finish the tasks in the same stage, which do not have interdependency, all together. Note that the tasks in a stage cannot finish, or sometimes even start, until all the tasks in the previous stage have all completed. Thus, all tasks in a stage should be treated as a whole in job scheduling. We call the set of all the tasks in a stage of a job a *cotask*. The completion time of a cotask is from the starting time of the first flow in this cotask, to the ending time of the last task in this cotask.

This work is partially supported by The National Basic Research Program (China' s 973 Program) (2013CB329103), NSFC Fund (61271165, 61301153, 61671130,61671124), Program for Changjiang Scholars and Innovative Research Team in University (PCSIRT), the 111 Project(B14039), and Technology Program of Sichuan Province (2016GZ0138).

In this paper, we aim to minimize the average Cotask Completion Time (CCT) in cotask scheduling. To the best of our knowledge, there is no prior work on cotask scheduling for cloud computing. Some heuristic based schemes have been proposed to optimize MapReduce [5]–[9]. These schemes focus on optimizing some specific aspects, such as straggler mitigation, of computing frameworks. They do not consider minimizing cotask completion time. Some schemes perform the theoretical analysis of task scheduling [10, 11]. However, they focus on minimizing the average task completion time, which may not lead to average cotask completion time. Furthermore, as they assume that tasks have already been assigned to computing nodes, these schemes miss the significant opportunity and flexibility of assigning tasks to nodes in doing task scheduling.

We propose the Cotask Scheduling Scheme (CSS), and take MapReduce as a representative of computing frameworks. CSS schedules cotasks following the Minimum Completion Time First (MCTF) policy [12], and we prove this problem is NP-hard. We formulate the model using the Integer Linear Programming (ILP), and solve it through an efficient heuristics based on ILP relaxation. Through real trace based simulations, we show that CSS is able to reduce the average CCT by up to 62.20% and 69.93% with traces from our testbed and from a large production cluster respectively.

II. Background and Related Work

In this section, we first briefly introduce some characteristics of current workload (Section II-A), and then show a motivation example of our work (Section II-B). After that, we briefly discuss the related works of this paper in Section II-C.

A. Characteristics of Current Workload

1) Computation and Networking Phases: Real traces of MapReduce jobs on experiment testbed are reported in [13]. From those real traces, we can see that most of the time taken to execute a job is spent on the map phase and reduce phase, i.e. computation phases dominate the processing duration of a job. On the other hand, Hadoop traces from Facebook show that shuffle phase could also account for 33% running time of jobs [14]. Apparently, both computation and networking

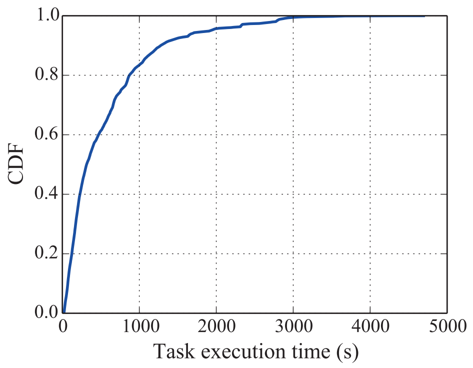


Fig. 1. CDF of the task execution time in a cotask.

account for a significant part of a job execution time; the scheduling should consider both of them together.

2) Heterogeneous Tasks: In cluster computing, a job is consisted of multiple tasks that may have quite different execution time. Fig. 1 shows the task execution time CDF of a job in a Google cluster [15]. We can see that some of the tasks require much longer time than most of the remaining tasks. These long tail tasks would block the execution of short jobs, if not scheduled correctly. Hereby, *appropriate task placement may benefit the average completion time of jobs.*

B. A Motivating Example

In this section, we present a motivation example to show how cotask based placement and scheduling benefits the job completion time for distributed computing. The example is shown in Fig. 2, where $T_{ij}^{(m)}$ denotes the task m that belongs to the j^{th} stage of job i; we also abuse $T_{ij}^{(m)}$ to denote the execution time of the corresponding task.

In this example, there are two jobs, each job has two computation phases, and there are two processors can be used to execute tasks. In addition, we also assume two jobs arrive into the cluster at the same time, and the task execution time satisfies $T_{21}^{(a)} < T_{11}^{(a)} = T_{11}^{(b)} < T_{21}^{(b)}$, and $T_{12}^{(b)} < T_{22}^{(a)} < T_{12}^{(a)} < T_{22}^{(b)}$. To minimize the average task completion time, we should schedule the tasks according to the scheme labelled as "Shortest Task First". To pursue the minimum average task completion time, when there is an idle host, the smallest task should be assigned to it. The tasks in the second phase of each job will arrive when their first phase complete at time t_2 and t_4. With this scheme, the task of two jobs will interleave with each other, and the two jobs will complete at time t_5 and t_7, respectively.

If we adopt the scheme labelled as "Shortest Cotask First", we should first assign the first phase of job 1 to the hosts. As the first phase of job 1 completes at time t_1, its second phase will arrive, whose completion time is even less that the first phase of job 2. Accordingly, we first execute the second phase of job 1. In this case, we avoid the unnecessary waiting of job 1 in "Shortest Task First" scheme. Another benefit brought by cotask based scheme is that we can first start the larger task in a cotask (as we first start $T_{21}^{(b)}$ rather than $T_{21}^{(a)}$) to minimize difference between completion time of each tasks in a cotask. It can also benefit the job completion time. Now the two

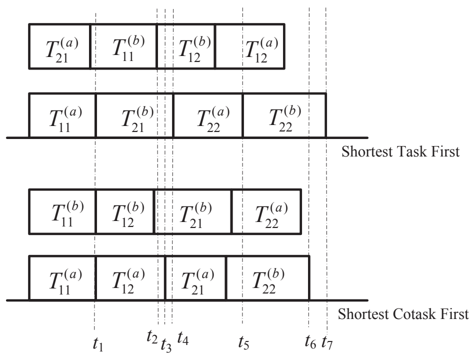


Fig. 2. Motivation example.

jobs will complete at time t_3 and t_6, respectively. Apparently, our cotask abstraction, and the minimize CCT objective, can significantly help reducing the job completion time.

C. Related Work

Our work is to minimize the average cotask completion time (CCT) by optimizing the task placement and execution order. There are many works on this field. Some of them focus on dealing with the so called *straggler* tasks to reduce job tail latency [5]–[7]. Some of them focus on data locality and fairness in a multi-tenant scenario [16, 17], while our paper focuses on a single tenant multi-job scenario. ShuffleWatcher and DynMR [8, 9] perform network aware task placement and scheduling, via heuristic insights, to reduce network congestion. As a comparison, our approach is more a theoretical understanding of the relationships among computation/networking phases. Kwiken takes an end-to-end view of latency improvements [18] by intelligently allocating resources among phases in a higher level. Our approach is a lower level per-task scheduling to approximate the minimum CCT. The most related works to our paper is presented by Kodialam et.al. [10, 11]. As mentioned before, they are fundamentally flawed in both task assignment assumptions and optimization objectives.

III. CSS Overview

A. System Overview

CSS adopts a queue based scheduling scheme. If there are M hosts available for the tenant to execute its tasks, we maintain M queues on the master server, one for each host. When a cotask enters the network, CSS places each task in this cotask into one of these queues. If there is a host becomes idle, the first task in its corresponding queue will be popped out and start to execute on this idle host. It is worth noting that the task placement and order in the queues can be adjusted when more cotasks enter into the network. Accordingly, CSS is a preemptive system. On the other hand, only when there is an idle host, a task in the associated queue can be popped out to execute. Therefore, all the tasks that have already been launched can be maintained till they are finished.

CSS is a centralized system and all the algorithms should be deployed on the master server. How CSS works on the master server is illustrated in Fig. 3. In this figure, there are six computation hosts in the network (H_1–H_6). Correspondingly,

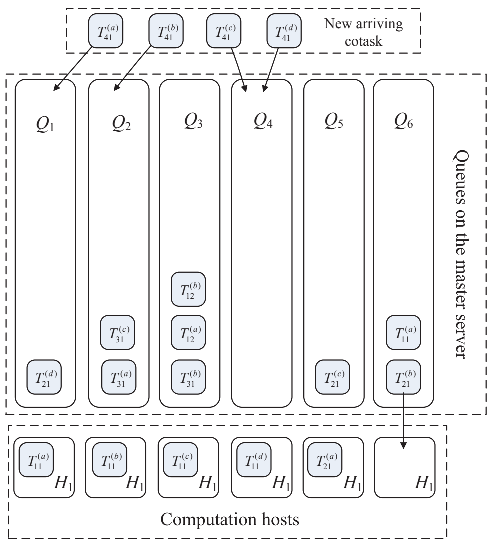


Fig. 3. System overview of CSS.

we maintains six queues (Q_1–Q_6) on the master server. If there is an idle host (H_6 in Fig. 3), the first task in the corresponding queue ($T_{21}^{(b)}$ in Q_6) should be popped out and launched on this idle host.

B. CSS *Framework*

There are two modules in CSS. One is the Static Task Scheduling and Placement (STSP) module, while the other one is the Dynamic Task Placement Adjusting (DTPA) module. STSP is triggered when a new cotask arrives, and DTPA works when a task is waiting for some data due to some tasks in its previous stage have not completed. The pseudo code of these two modules is shown in Algorithm 1.

Given all the cotasks to be scheduled and the remaining time to finish the current running cotasks on each host W, STSP first schedules the cotasks that are waiting for a long time (for the starvation-free purpose), and then schedule the cotasks based on MCTF policy (Line 5-15) to optimize the average CCT. After that, STSP places all the tasks of the selected cotask to appropriate hosts (Line 17). The most important step in STSP is to calculate the minimum CCT (Line 6).

When STSP optimizes the CCT, it assumes that no task would be blocked when it is fetching the required data. DTPA is designed as a complement to run on each host H in case such blocking occurs. When the waiting time is so long that the last task in the current executing cotask, C, cannot start before all other tasks in C complete (Line 26), DTPA will reposition this task to reduce C's completion time.

IV. Algorithm Details

There are three technical details needing further discussions in CSS: how to calculate the minimum completion time of each cotask, how to schedule the tasks on each host, and how to dynamically change the task placement for further optimization. We discuss these aspects in this section. For space limiation, we omit the proofs of the lemma, corollary and theorem in this section. However, they can be easily derived with the same technology in Section 1.4.1 of [19].

Algorithm 1: CSS Algorithm Framework

```
1:  **procedure** STSP(CoTasks $S$, Map<HostId, Time> $W$)
2:  Sort all the cotasks in $S$ non-increasingly in terms of their waiting time
3:  **while** $S \neq \Phi$ **do**
4:      $T_{min} \leftarrow \infty$, $C_{min} \leftarrow \Phi$
5:      **for** $C \in S$ **do**
6:          $T_C$=minCCT($C$, $W$);
7:          **if** $C.waitTime() > \delta$ **then**
8:              /*waitTime() returns the total waiting time of all the tasks of cotask $C$, and $\delta$ is a predefined starvation threshold*/
9:              $T_{min} \leftarrow T_C$, $C_{min} \leftarrow C$
10:             break;
11:         **end if**
12:         **if** $T_C < T_{min}$ **then**
13:             $T_{min} \leftarrow T_C$, $C_{min} \leftarrow C$
14:         **end if**
15:     **end for**
16:     $S \leftarrow S \setminus C_{min}$
17:     Place $C_{min}$ to appropriate hosts and update $W$
18: **end while**
19: **end procedure**

20: **procedure** DTPA(Host $H$)
21: $t_{wait}^{(0)} \leftarrow 0$ //total waiting time on $H$
22: **while** true **do**
23:     $C \leftarrow H.currentCotask()$, $task \leftarrow C.lastTask(H)$ /* $C$ is the executing cotask on $H$, $task$ is the last task in $C$ that will be executed on $H$*/
24:     $t_{wait} \leftarrow currentTask.timeToWaitData() + t_{wait}^{(0)}$
25:     $t_{exec} \leftarrow task.timeConsumption(H)$
26:     **if** $t_{wait} + task.ESTCT() - C.ESTCT() > t_{exec}$ **then**
27:         //ESTCT denotes ESTimated Completion Time
28:         $TL.removeBack()$
29:         replaceTask(task)
30:         $t_{wait} \leftarrow t_{wait} - task.ESTCT()$
31:     **end if**
32:     **if** $currentTask.startProcessing()$ **then**
33:         $t_{wait}^{(0)} \leftarrow t_{wait}$
34:     **end if**
35: **end while**
36: **end procedure**
```

A. *Minimize Single Cotask Completion Time*

Let w_k denote the remaining time to complete all the tasks on host k that should be completed before the cotask that is being optimized, t_{ij}^k denote the time to complete the j^{th} task of cotask i on host k, and introduce binary integer variables x_{ij}^k to indicate whether the j^{th} task of cotask i is placed on host k. The problem to minimize the CCT of cotask i can be formulated as following CCT-ILP model:

$$\text{minimize} \quad t_i \tag{1}$$

Subject to:

$$w_k + \sum_j t_{ij}^k x_{ij}^k \leq t_i, \quad \forall k \tag{1a}$$

$$\sum_k x_{ij}^k = 1, \quad \forall j \tag{1b}$$

$$x_{ij}^k \in \{0,1\}, \quad \forall i,j,k \tag{1c}$$

The objective is to minimize the CCT of cotask i, denoted by t_i. Constraint ($1a$) says that on any host k, the time to complete all the tasks of cotask i should be no later that the CCT. Constraint ($1b$) indicates that any task j must and can only be placed on one host. CCT-ILP cannot be solved in a timely manner for large scale systems, thus we need an approximation algorithm to estimate the minimum CCT.

To this end, we find the minimal t_i that ensures there are feasible solutions satisfying all following constraints (referred as CCT-LP):

Algorithm 2: Task Placement

```
1:  procedure taskPlacement(CoTask $C$, Solution $x$)
2:  Construct a bigraph $BG$ according to $x$ as in Corollary 1
3:  Remove all the task nodes with only one nodal degree
    and place these tasks to the connecting hosts
4:  for all connected components $P \in BG$ do
5:      if $\|E(P)\| = \|N(P)\|$ then
6:          Find the cycle in $P$
7:          Assign the tasks along the cycle
8:      end if
9:      Set arbitrary task node as the root
10:     for all task node $v \in P$ do
11:         Place task $v$ to its arbitrary child
12:     end for
13: end for
14: end procedure
```

$$w_k + \sum_{j \in T_k(t_i - w_k)} t_{ij}^k x_{ij}^k \leq t_i, \quad \forall k \tag{1aL}$$

$$\sum_{k \in H_j(t_i - w_k)} x_{ij}^k = 1, \quad \forall j \tag{1bL}$$

$$x_{ij}^k \geq 0, \quad \forall i,j,k \tag{1cL}$$

where $T_k(t)$ is the set of tasks that can be completed on host k in time t, while $H_j(t)$ is the set of hosts that can complete task j in time t. When t_i is a variable to be minimized, CCT-LP is a nonlinear programming that is difficult to solve. However, if t_i is fixed, it becomes a linear programming model and there are efficient algorithms to check its feasibility. Based on this fact, we design an algorithm based on binary search to get t_i.

In the solution of CCT-LP, a task may be split onto multiple hosts. Accordingly, we need to round the solution of CCT-LP to derive an integer task placement solution.

Lemma 1: For any cotask i with T tasks, if there are M hosts available for it, then at most $(M+T)$ variables will be non-zero in the optimal solution of CCT-LP.

Corollary 1: We construct a bigraph $G(x) = \{U, V, E\}$ according to the solution of CCT-LP, x, where $U = \{h_1, h_2, \ldots, h_M\}$ is the set of nodes denoting hosts, called host nodes, while $V = \{t_1, t_2, \ldots, t_T\}$ is the set of nodes denoting tasks, called task nodes. There is an edge between t_j and h_k, iff $x_{ij}^k > 0$. In this case, any connected component, P, in $G(x)$ can be modified to a pseudo tree without increasing CCT.

Based on Corollary 1, we can design Algorithm 2 to place each task to a host, where $E(P)$ and $N(P)$ are the set of edges and nodes in graph P, respectively. In this Algorithm, Line 3 is to deal the tasks that have only one placement option according to x. After that, each task node in P has at least two node degrees. If the number of edges is equal to the number of nodes, there must be a cycle in P. In this case, we first find this cycle with DFS, and assign the tasks along this cycle to ensure every host gets only one task (Lines 6–7). Then, every leaf nodes in the remaining tree should be a host node and it has one (and only one) parent that is a task node. Even if we assign a task to its arbitrary children, any host can get at most one task in P. The performance of Algorithm 2 can be guaranteed through the following theorem:

Theorem 1: For any cotask i, Algorithm 2 can get a task placement scheme such that the CCT of cotask i is at most $2t_i - \min_{k \in H_i} w_k$, where t_i is the CCT lower bound derived from CCT-LP, and H_i is the set of host available to cotask i.

B. Task Aggregation

In last subsection, we derive a task placement that can complete a cotask within 2x of the optimal placement. To further optimize the system performance, we need to gather tasks together (place them on as fewer hosts as possible). However, this is clearly a NP-complete problem since it is a generalized bin-packing problem. Different from the standard bin-packing, in our problem, the object size is different in different bins and the size of each bin is not identical. Accordingly, we design a heuristic to do the task aggregation.

This heuristic is shown in Algorithm 3. The main idea of this algorithm is to sequentially check if a host used by a cotask can be removed without increasing the CCT. To get a better performance, this algorithm does this checking from the host with the least tasks in cotask C (Line 2). Only if all the tasks in C on a host can be moved to other hosts used by C without increasing the cotask C's CCT, such replacement will be concreted (Lines 5–11).

C. Scheduling Tasks of a Cotask on One Host

In previous subsections, we introduce algorithms to determine how to place tasks to hosts and schedule cotasks in the cluster. However, there remains an additional issue — when there are multiple tasks of a cotask placed on it, how to schedule these tasks. Different from scheduling cotasks, we should schedule tasks on each host following the maximum completion time first policy. The small tasks have more chances to be repositioned, so as to further optimize the CCT. In addition, this policy can delay the start of the next

Algorithm 3: Task Aggregation

```
1:  procedure taskAggregation(CoTask $C$, Map<Host, List<Task>> $placement$, List<Host> $usedHost$)
2:  Sort usedHost non-decreasingly in terms of the task number of $C$ placed on it
3:  for $H \in usedHost$ do
4:      $tempPlacement \leftarrow placement$
5:      for  $task \in Placement.get(H)$ do
6:          if task cannot be moved to other host without increasing $C.ESTCT()$ then
7:              $placement \leftarrow tempPlacement$
8:              break
9:          end if
10:         Move $task$ to other host
11:     end for
12: end for
13: end procedure
```

stage computation, hence leave more optimization space to the overall cotask scheduling.

D. *Dynamical Task Placement Adjusting in DTPA*

In Algorithm 1, the "if" expression in Line 26 means that on host H, the last task belonging to the currently executing cotask C cannot even start before all the other tasks in C complete. In this case, we can try to move this task to another host and reduce the CCT of C. In CSS, we treat this task as a new arriving cotask and call procedure STSP.

V. Performance Evaluation

In this section, we evaluate the performance of CSS with some real traces both from a Hadoop system in our testbed (Section V-A), and from Google cluster [15] (Section V-B). In addition, we evaluate how estimation error of task execution time impacts the performance of CSS in Section V-C.

A. *Real-trace Driven Simulation*

In this section, we first collect some traces from our testbed with 20 hosts where Hadoop 0.20.2 is deployed. To collect these real traces, we set the slow-start parameter to be 70% and run 3×25 GB Terasort jobs on the testbed, and record all the information we need into log files, such as the execution time of each mapper/reducer, the volume of each flow, the source/mapper and destination/reducer of each flow.

With these traces as input, we evaluate the system performance under CSS. The simulation results are shown in Fig. 4. From this figure, the key result is that CSS can not only reduce the completion time of both the map phase and reduce phase of all the jobs, but also reduce the unncessary waiting of each job. Therefore, CSS can efficiently reduce the completion time of each job.

Fig. 4(a) shows the map phase completion time of each job before and after CSS is deployed. In this figure, we can see that the map phases of all the three jobs are shortened by 25.13% to 30.22%, and the average CCT is reduced by 27.00%. As to the reduce phase completion time of each job shown in Fig. 4(b), it is reduced by 44.90% to 78.03% and by 62.20% on average, respectively. The reason is that CSS can appropriately place the task on different hosts and cut the long tail of each cotask.

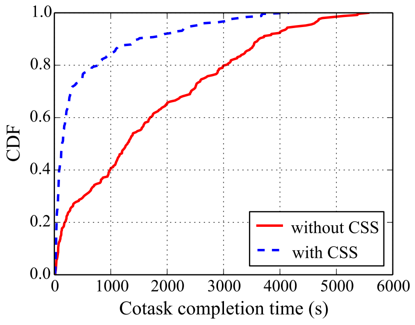


Fig. 5. Performance evaluate the of CSS by using traces from google cluster.

Fig. 4(c) shows how long (on average) each reducer should wait for its required data before and after the CSS is deployed. Before the CSS is deployed, almost one third of the job execution time is spent on waiting the required data of each reducer (the overall JCT is shown in Fig. 4(d)). After CSS is deployed, all the tasks in the same cotask will be executed together, and then the required data can be prepared for fetching in a relatively short time. Therefore, the average waiting time is reduced by 32.28% to 59.29%.

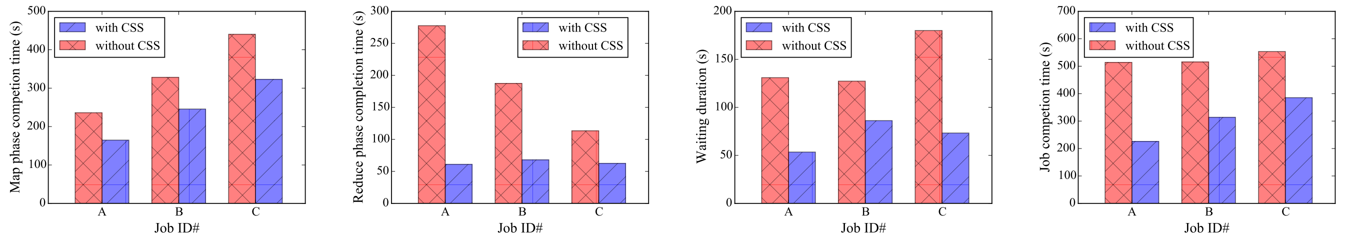
Due to the reduction of reducer waiting time, the JCT is also reduced. This result is shown in Fig. 4(d). From this figure, we can see that the JCT is reduced by 30.40% to 56.05% with CSS. It should be noted that the JCT saved by CSS is larger that the time saved from reducing the reducer's waiting time. This is due to the fact that in addition to reduce the waiting time, CSS also reduce the duration of each computation stage by scheduling all the tasks of a cotask together.

B. *Performance in Production Clusters*

To study the performance of CSS in production clusters, we use the Google cluster data [15] as input. Since google only published the task information without communication information, we can only study how CSS affects the average CCT by treating all the tasks in a job as a cotask. Therefore, the CCT is equivalent to JCT in this simulation.

Although there are 9545 hosts in google's cluster, each job uses only tens of hosts. Accordingly, we only pick out 248 jobs (which are divided into 1005 tasks) executed on 100 hosts to test CSS for time efficiency purpose.

Fig. 5 shows the CDF of CCT without and with our proposed CSS. From this figure, we can see that with CSS, the CDF curve is moving left, which means the average CCT is reduced. In fact, the average CCT is reduced from 1647.47s to 495.43s. About 69.93% of the average CCT is reduced. Furthermore, more than 82% of all cotasks can be finished in 1000s with CSS, while only 40% of all cotasks can be finished in 1000s without CSS. This indicates that the QoS of computing cluster should be greatly improved by CSS.



(a) Map phase completion time (b) Reduce phase completion time (c) Idle duration (d) Job completion time

Fig. 4. Performance evaluate the of CSS using traces from testbed.

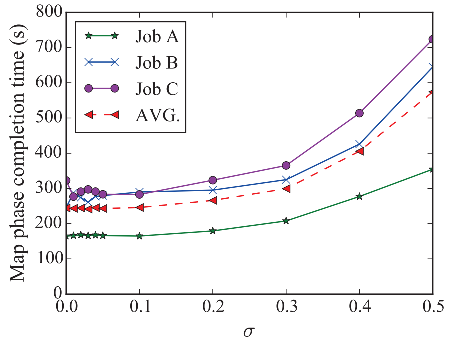


Fig. 6. Impact of the estimated errors of task execution time.

C. Impact of Execution Time Estimation

As CSS schedules the cotasks according to the estimation of task execution time, we study how such estimation error impacts the system performance in this section. To this end, we form the *imitation* task execution time by adding some zero-mean normal "noise" to the real execution time of each map task used in Section V-A. After that, we use these imitation task execution time as the input to run CSS and calculate the map phase completion time based on the original trace.

The simulation result is shown in Fig. 6. In this figure, the x-label σ means that the noise we add to the mapper completion time follows the normal distribution with σ times of the real execution time as its variance. Each point in this figure is obtained by averaging the results of 20 experiments. From this simulation, we can see that the CCT of each task becomes larger with the increasing of estimation error. However, even if the error variance is 0.3 times of the real execution time, it will result in slight CCT increments. Accordingly, CSS is insensitive to the estimation of task execution time.

VI. Conclusion

In this paper, we make three key contributions. First, we propose the concept of cotask to better capture the relationships among tasks and the problem of minimizing the average cotask completion time in task scheduling. Second, we formulate the average cotask completion time minimization problem using Integer Linear Programming, and solve it through efficient heuristics based on ILP relaxation. Third, we implemented our cotask scheduling scheme and evaluated it using real trace driven simulation. Through real trace based simulations, we show that CSS is able to reduce the average CCT by up to 62.20% and 69.93% with traces from our testbed and from a large production cluster respectively. For reducing job completion time, CSS reduce the average JCT by at least 30.40% to each job on the testbed, while reduce the average JCT by 69.93% according to the data from the large production cluster.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *USENIX OSDI*, 2004.
[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD*, 2010.
[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the USENIX HotCloud*, 2010.
[4] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the ACM NotNets*, 2012.
[5] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI*, 2008.
[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *NSDI*, 2013.
[7] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, 2010.
[8] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proceedings of the USENIX ATC 2014*, pp. 1–12.
[9] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling," in *Proceedings of the EuroSys*, 2014.
[10] H. Chang, M. Kodialam, R. Kompella, T. Lakshman, M. Lee, and S. Mukherjee, "Scheduling in mapreduce-like systems for fast completion time," in *IEEE INFOCOM*, 2011.
[11] F. Chen, M. Kodialam, and T. Lakshman, "Joint scheduling of processing and shuffle phases in mapreduce systems," in *IEEE INFOCOM*, 2012.
[12] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the ACM SIGCOMM 2014*.
[13] V. Abhishek, C. Ludmila, and C. Roy H., "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the EuroSys*, 2010.
[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra." in *Proceedings of the ACM SIGCOMM*, 2011.
[15] "Google cluster data," https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1, [Online; accessed 28-July-2014].
[16] 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 *Proceedings of the EuroSys*, 2010.
[17] "The hadoop fair scheduler," https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html, [Online; accessed 28-July-2014].
[18] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," vol. 43, no. 4, pp. 219–230, 2013.
[19] D. S. Hochbaum, Ed., *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., 1997.