Automatic Task Slots Assignment in Hadoop MapReduce

Kun Wang, Juwei Shi, Ben Tan, Bo Yang
Peking University, IBM Research - China

Contribution

--demonstrate the drawback of fixed task slots in Hadoop MapReduce by experiments.
  --The experimental results indicate that the optimal task slots of various workloads are different

--design and implement the automatic task slots assignment mechanism.
  --Experimental results show that the implementation can dynamically adjust the task slots capacity to the optimal setting in run time

Introduction

--Hadoop uses the task slots parameter to set the maximum number of parallel Map and Reduce tasks can run on a Task Tracker (slave) node

--The cluster's task slots capacity is the sum of task slots of each Task Tracker node

--controlling the task slots directly based on the captured resource usages is not efficient.
  --There is a delay between the time the task slot is assigned by Job Tracker and the time the task enters the computing or I/O intensive stage.
  --The straightforward control may lead to resource contention and system overload

--The main idea is to periodically capture the resource utilization such as CPU consumption from each Task Tracker node.
  --Job Tracker dynamically assigns task slots according to the captured information within a sliding window

--The goal of the automatic assignment is improving system utilization and avoiding resource contention

Preliminaries

--Task slots determine the resource utilization on each node

--Assume that the number of cores on a Task Tracker node is Nc

  --The recommended range is [Nc /2, Nc × 2]

Preliminaries--Impact of Task Slots

--examine the case that the overall CPU utilization of the cluster is low

--conduct the experiment on the 2- node POWER cluster
  --each node has 12 cores and 48 threads
  --cluster has 96 threads

--the workload is the Maximal Clique Finding
  --use the BBS data as the input data set
  --the social network data is an undirected graph with no self loops, consisting of 201, 319 nodes and 1, 602, 025 links

--set the number of both Map and Reduce tasks to 96 (1 task slots per hardware thread)

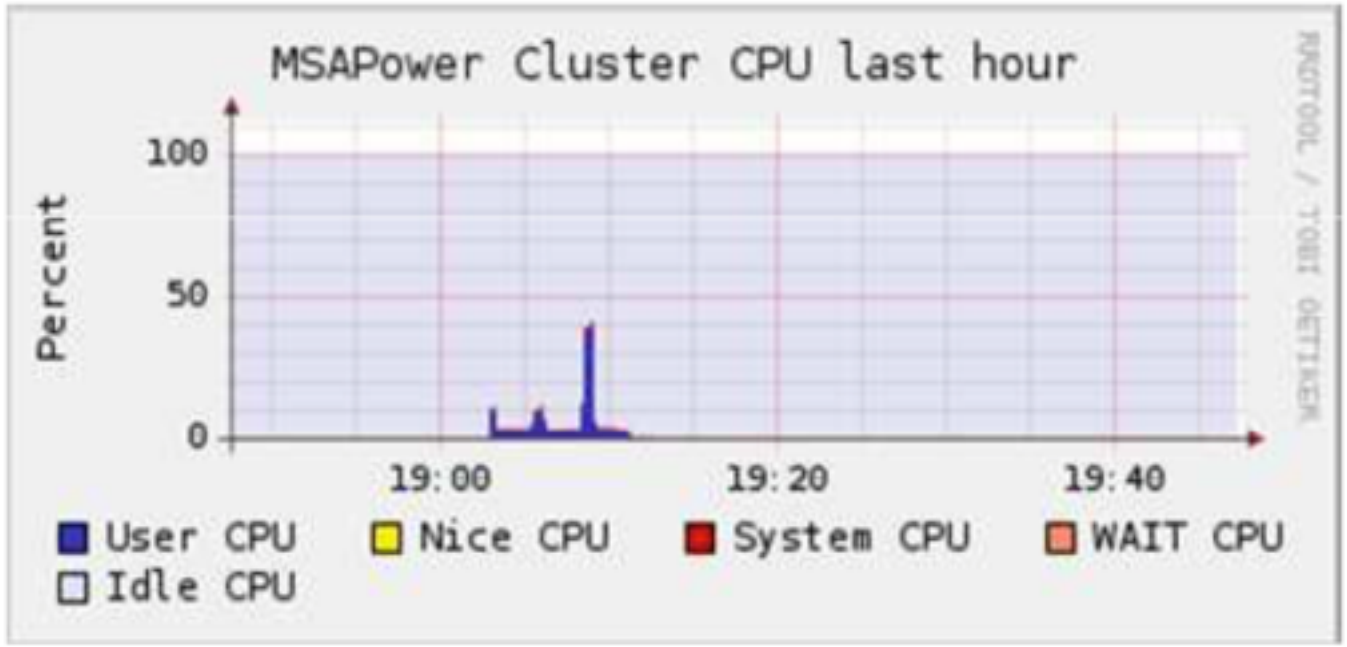

Figure 1: Cluster-wide CPU Utilization of Maximal Clique Finding

--It indicates that the CPU utilization is very low (less than 10%).

--The reason is that the CPU consumption of each task is very low.

--We obtain the optimal setting when the task slots capacity is 236, which is larger than two times of the number of hardware threads(2 * 96) in the cluster

Preliminaries--Impact of Task Slots

--examine the case that the overall CPU utilization of the cluster is relative high

--conduct the experiment on the 2- node POWER cluster
  --each node has 12 cores and 48 threads
  --cluster has 96 threads

--the workload is Terasort
  --input data size 30GB

--set the number of both Map and Reduce tasks to 96 (1 task slots per hardware thread)

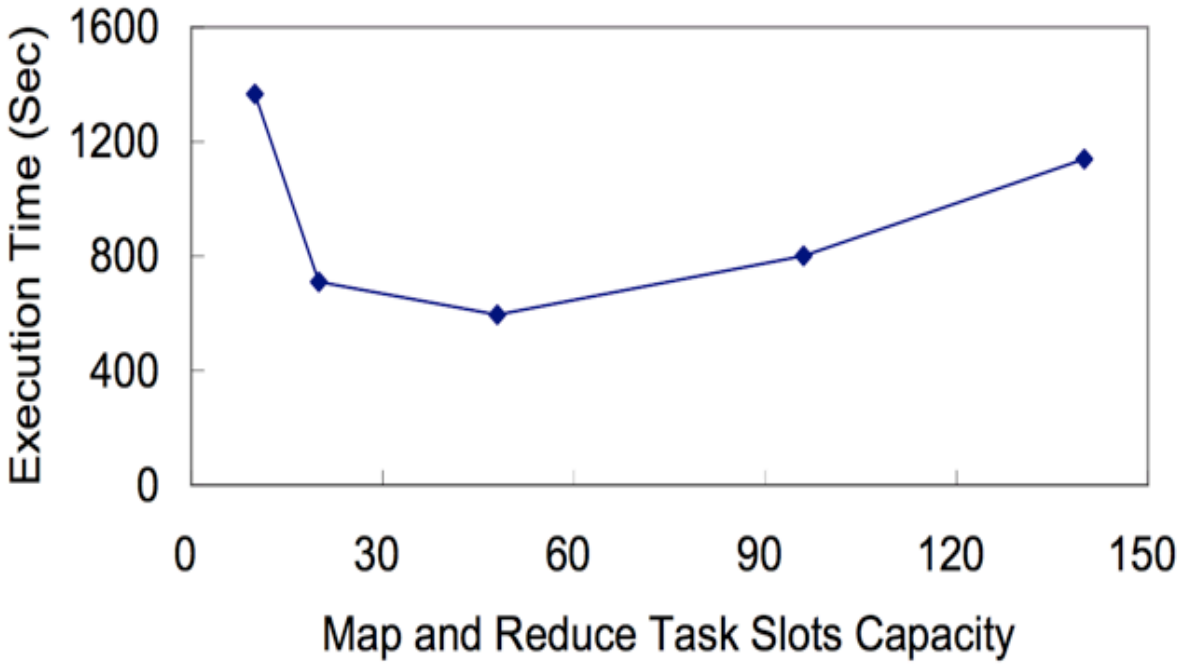

Figure 2: The Impact of Task Slots on Terasort

--shows the impact of Map and Reduce task slots on the execution time of Terasort

--the optimal execution time when task slots capacity is 48, which equals to a half of number of hardware threads

Preliminaries--Impact of Task Slots

--examine the impact of task slots on x86 platform

--conduct the experiment on the 5- node x86 cluster
  --each node has 4 cores
  --cluster has 20 threads

--the workload is World Count
  --data size is 25GB

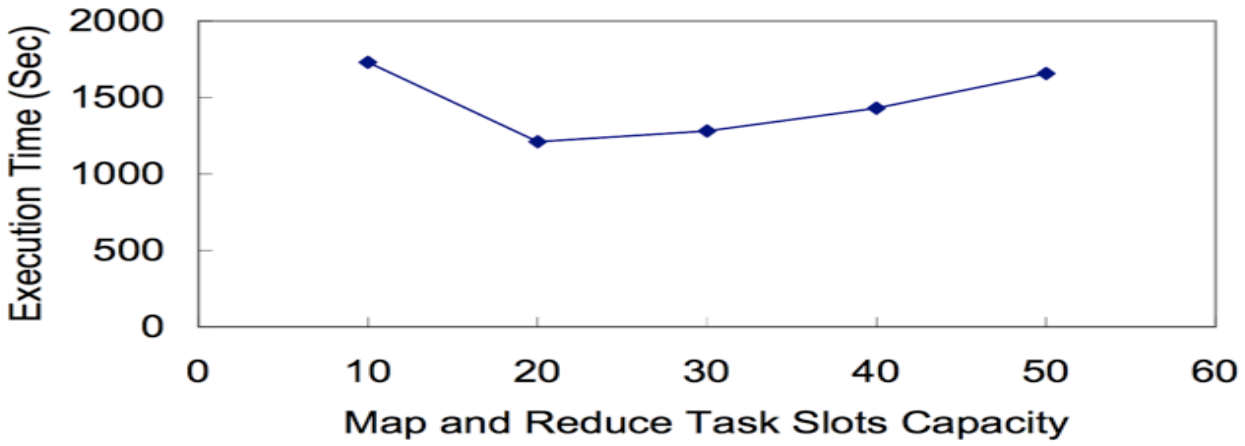

Figure 3: The Impact of Task Slots on Word Count

--obtain the optimal setting when the task slots capacity is 20, which equals to the number of cores in the cluster

Table 1: Optimal Task Slots for Various Workloads

	Map Task Slots	Reduce Task Slots
Maximal Clique	$> 2 \times cores$	$> 2 \times cores$
Terasort	$cores/2$	$cores/2$
Word Count	$cores$	$cores$

-- It is obvious that manually configure optimal setting for each workload on each Task Tracker node is not feasible.

-- have to manually traverse all the settings to obtain the optimal one for each workload

Dynamic Task Slots Assignment

-- capture user time, system time, and iowait time as the resource utilization states
  -- user time and system time represent the time spent on executing user code and system kernel code, respectively
  -- If we have more task slots than the machine's capacity, system time will increase due to resource contention overhead among tasks.
  -- iowait time is the percentage of time the CPU is idle and there is at least one I/O in progress.
    --High percentage of iowait time often indicates that the machine is bounded by I/O operations

Algorithm 1 State based Task Slots Assignment

Require: Host h, RunningTasks n, AssignedTasks A

Get the CPU info I for h

Update $\{\langle n, h\rangle, I\}$ in State Set of this Job

$S \leftarrow getStateSet(\langle n, h\rangle)$

$S' \leftarrow getStateSet(\langle n+1, h\rangle)$

for each $s \in S$ **do**

  if $s > T_{up}$ **then**

    return $A = \varnothing$

for each $s \in S'$ **do**

  if $s \neq$ `NULL` and $s > T_{up}$ **then**

    return $A = \varnothing$

if the pending Map task set $M_P \neq \varnothing$ **then**

  add one task $m \in M_P$ to A

if the pending Reduce task set $R_P \neq \varnothing$ **then**

  add one task $r \in R_P$ to A

--The main idea is to keep the resource usage state of the current running job based on the number of concurrent running tasks

--I contains user time, system time and iowait time

--The update is the average of the recent t seconds

--update the state record corresponding to host h and currently running tasks n on that slave host

--The job scheduler will not assign new task slots if the historical resource usage state set indicates that the resource usage after the assignment will extend the specified threshold

--if the speed of task assignment is faster than that of task completion, the task slots on the Task Tracker node will increase

Implementation

--Automatic Task Slots Assignment (ATSA)

-- ATSA client is responsible for collecting and sending CPU information of Task Tracker nodes

--ATSA server receive CPU information and parse it into hash map format, then can be queried by ATSA scheduler

-- When ATSA scheduler receives the heartbeat from a Task Tracker, it queries the CPU information from ATSA server using the Task Tracker's host name. Then ATSA scheduler assigns tasks based on the algorithm

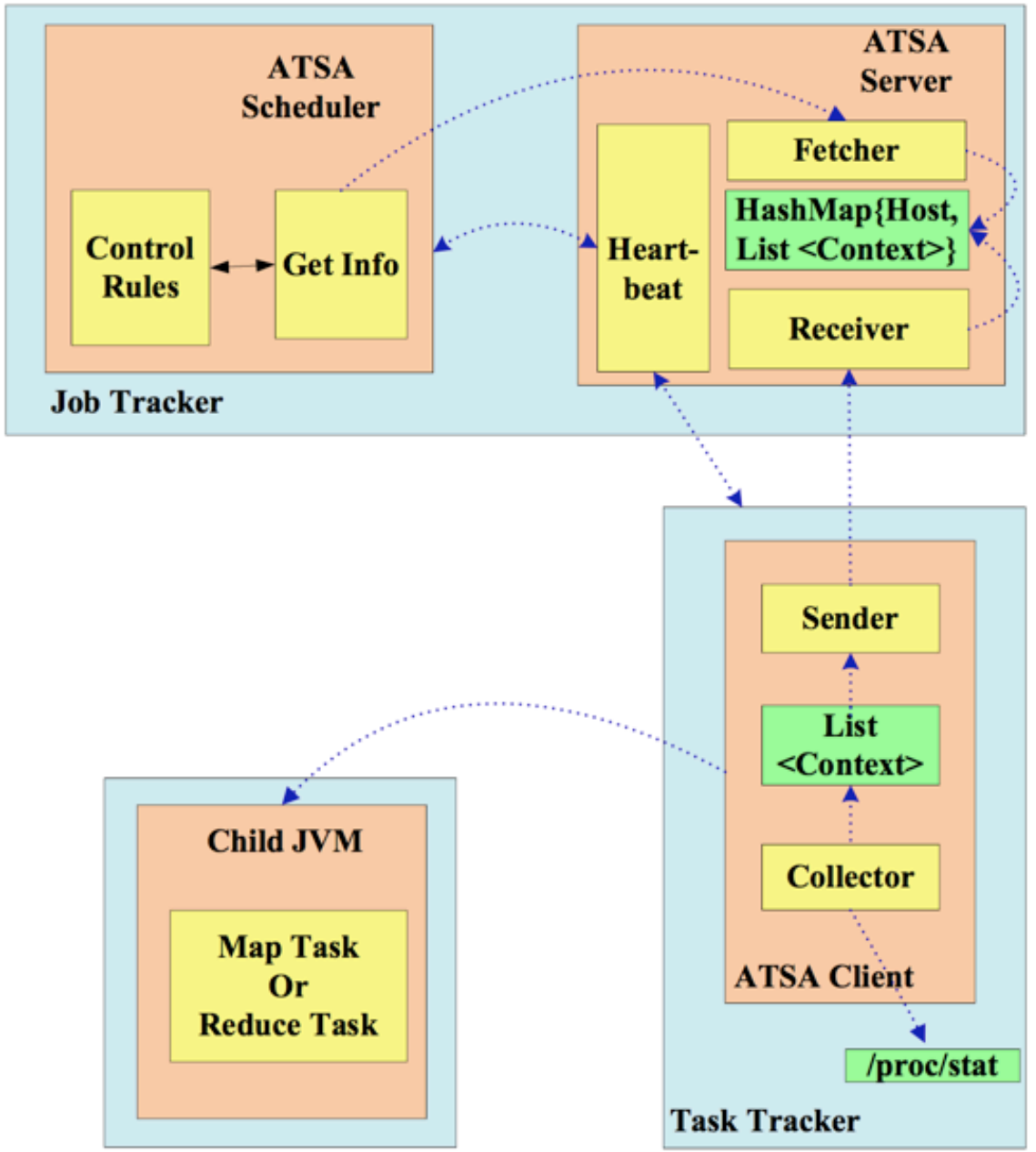

Figure 4: Overview Architecture of the ATSA Implementation

Evaluation

--evaluation ATSA performance based on x86 platform

--conduct the experiment on the 5- node x86 cluster
  --each node has 4 cores
  --cluster has 20 threads

--CPU intensive workload is World Count

--I/O intensive workload is Stream Sort

--data size is 25GB

-- 100 Map tasks

--MC represents Map task slots capacity

--RC represents Reduce task slots capacity

--M represents the number of map tasks

--R represents the number of Reduce tasks

--ATSA outperforms the optimal setting of Hadoop 0.21.0 in terms of execution time

Table 3: Comparison of Word Count Performance on x86 Cluster

	MC	RC	M	R	Execution Time
v0.21.0	10	10	100	10	28min, 50sec
v0.21.0	20	20	100	20	**20min, 11sec**
v0.21.0	30	30	100	30	21min, 21sec
v0.21.0	40	40	100	40	23min, 50sec
v0.21.0	50	50	100	50	27min, 37sec
ATSA	Auto	Auto	100	50	**19min, 21sec**

--data size is 13GB

-- 100 Map tasks

-- indicates that ATSA nearly achieves the performance of Hadoop 0.21.0 with optimal setting

Table 4: Comparison of Stream Sort Performance on x86 Cluster

	MC	RC	M	R	Execution Time
v0.21.0	10	10	100	10	30min, 17sec
v0.21.0	20	20	100	20	19min, 43sec
v0.21.0	30	30	100	30	**11min, 48sec**
v0.21.0	40	40	100	40	15min, 22sec
ATSA	Auto	Auto	100	40	**13min, 40sec**