Improving Resource Utilization by Timely Fine-Grained Scheduling

Tatiana Jin, Zhenkun Cai, Boyang Li, Chenguang Zheng, Guanxian Jiang, James Cheng

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Core Problem

Cluster Resource Utilization

• Scheduling Efficiency
• Utilization Efficiency
Cluster Resource Utilization

- hadoop
- YARN
- kubernetes
- MESOS
- Borg
- Sparrow
- Apollo
- Mercury
Scheduling Efficiency and Utilization Efficiency

**Scheduling Efficiency (SE)**
- Capacity
  - Allocated

**Utilization Efficiency (UE)**
- Capacity
  - Actually Utilized
  - Allocated
Application Scenario

- Workload: 70% OLAP, 20% machine learning and 10% graph analytics

- Performance Objective
  1. Maximize job throughput (minimize makespan)
  2. Minimize average job completion time (JCT) (time from submission to completion)
Dynamic Resource Utilization Pattern

Figure 1. Resource utilization of different workloads (best viewed in color)
Central Idea

Ursa: achieving high SE and UE by fine-grained, dynamic, load-balanced resource negotiation
Design Objectives

- Obj-1. Accurate resource request
- Obj-2. Timely provision and release of resource
- Obj-3. Load-balanced task assignment
- Obj-4. Low-latency resource scheduling
Using Monotask to Handle Dynamic Patterns

- Monotask* is a unit of work that uses only a single type of resource (e.g. CPU, network bandwidth, disk I/O) apart from memory
- Introduced for job performance reasoning
- A unit of execution with steady and predictable resource utilization

System: Ursa

A scheduling and execution framework
template<typename ValueType>

class Dataset {

auto ReduceByKey(Combiner combiner, int partitions) {
    auto msg = dag.CreateData(this->partitions);
    auto shuffled = dag.CreateData(partitions);
    auto result = dag.CreateData(partitions);
    auto ser = dag.CreateOp(CPU) // create CPU Op
        .Read(this).Create(msg)
        .SetUDF(/*apply combiner locally and serialize*/);
    auto shuffle = dag.CreateOp(Net).Read(msg).Create(shuffled);
    auto deser = dag.CreateOp(CPU)
        .Read(shuffled).Create(result)
        .SetUDF(/*deserialize and apply combiner*/);
    this->creator.To(ser, ASYNC);
    ser.To(shuffle, SYNC);
    shuffle.To(deser, ASYNC);
    return result;
}

// ...
OpGraph dag;
Op creator;
int partitions;
}
High-Level APIs

• SQL (connected to Hive)
• Spark-like dataset transformations
• Pregel-like vertex-centric interface
System Overview

Scheduler

Resource Monitoring

Worker

Resource Status Report

Job Admission & Task Placement

CPU, Network, Disk

Monotask Queues

CPU, Network, Disk

Monotask Queues
System Overview

Scheduler
- Job Admission & Task Placement
- Resource Monitoring

Resource Status Report

Workers
- CPU, Network, Disk
- Monotask Queues

Job Manager
- DAG Manager
- Resource Demand Estimator
- Metadata Store
- Monotask Resource Request

Task Resource Usage
Task placement

• Resource usage estimation
  • The CPU, network and disk I/O usage is estimated on a monotask basis
    • The execution layer is designed to guarantee stable resource utilization by each type of monotasks during their execution
  • The memory usage is estimated on a task basis
    • The memory usage during the execution of a task is relatively stable

In contrast to simply using coarse-grained (historical) peak resource demands, monotask-based resource estimation allows per-resource needs to be captured dynamically at runtime
Task placement

• Stage-aware load-balanced task placement
  • A unified measure for multi-dimensional resource consumption
  • Total resource consumption in contrast to the peak demands of tasks
  • Stage-aware task placement to avoid stragglers due to scheduling delay
Task placement

- Stage-aware load-balanced task placement
  - Approximate Processing Time ($APT_r$)
    - $\text{APT}_r = \frac{(\text{Total input data size of assigned type-} r \text{ monotasks})}{\text{(Processing rate)}}$
  - $\text{APT}_r$ tells when resource-$r$ on a worker will become idle
  - Per-resource processing rates on each worker are periodically updated to the scheduler

- Expected Processing Time (EPT)
  - EPT is an indicator of whether a worker is over-loaded or under-loaded
  - Set to slightly larger than the scheduling interval
Task placement

From APT and EPT, we can compute

- Difference between EPT and APT for resource $r$ at worker $w$ as
  \[ D_r(w) = \max(0, \frac{EPT - APT_r(w)}{EPT}) \]

- The increase in the load of worker $w$ in using resource $r$ if task $t$ is placed in $w$ as $Inc_r(t, w)$

- Task placement score as a dot product
  \[ F(t, w) = \sum_{r \in \{CPU, network, disk, mem\}} D_r(w) \times Inc_r(t, w) \]
Task Placement

• Stage-awareness
  • Each schedule decision is a plan with tasks in the same stage instead of with a single task
  • Ranking plans by stage-average scores
  • A large bonus is given to a plan if the plan assigns all tasks in stage S, so that such plans are always considered before other plans
Other Scheduling Details

• Supporting scheduling policies
  • Earliest Job First (EJF) and Smallest Remaining Job First (SRJF)
  • Job ordering at the scheduler and monotask ordering at distributed queues

• Concurrency control
  • Avoid resource contention among running monotasks
  • Maintain high utilization of resource
Settings

- Workloads
  - **OLAP**: TPC-H and TPC-DS
  - **Mixed**: 70% OLAP, 20% machine learning and 10% graph analytics (ratio by total CPU usage)

- A cluster of 20 machines connected by 10 Gbps Ethernet
  - Resembles a small cluster requested by a quota group
Limitations of using coarse-grained containers

<table>
<thead>
<tr>
<th></th>
<th>makespan</th>
<th>avgJCT</th>
<th>UE_cpu</th>
<th>SE_cpu</th>
<th>UE_mem</th>
<th>SE_mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJF</td>
<td>2803</td>
<td>600.00</td>
<td>99.64</td>
<td>92.47</td>
<td>78.83</td>
<td>39.80</td>
</tr>
<tr>
<td>SRJF</td>
<td>2859</td>
<td>489.96</td>
<td>99.65</td>
<td>89.73</td>
<td>78.02</td>
<td>48.85</td>
</tr>
<tr>
<td>YARN+Spark</td>
<td>3849</td>
<td>1407.40</td>
<td>69.35</td>
<td>93.32</td>
<td>34.69</td>
<td>44.13</td>
</tr>
<tr>
<td>YARN+Tez</td>
<td>9228</td>
<td>4287.00</td>
<td>58.97</td>
<td>98.19</td>
<td>28.81</td>
<td>70.71</td>
</tr>
</tbody>
</table>

Performance on TPC-H

<table>
<thead>
<tr>
<th></th>
<th>makespan</th>
<th>avgJCT</th>
<th>UE_cpu</th>
<th>SE_cpu</th>
<th>UE_mem</th>
<th>SE_mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJF</td>
<td>1613</td>
<td>453.20</td>
<td>99.57</td>
<td>88.31</td>
<td>81.64</td>
<td>25.01</td>
</tr>
<tr>
<td>SRJF</td>
<td>1630</td>
<td>242.27</td>
<td>99.75</td>
<td>86.99</td>
<td>85.83</td>
<td>32.93</td>
</tr>
<tr>
<td>YARN+Spark</td>
<td>2927</td>
<td>894.36</td>
<td>48.56</td>
<td>90.48</td>
<td>19.39</td>
<td>37.65</td>
</tr>
</tbody>
</table>
Limitations of using coarse-grained containers

TPC-H

TPC-DS
## Compare with Alternative Approaches

### Performance on Mixed

<table>
<thead>
<tr>
<th></th>
<th>makespan</th>
<th>avgJCT</th>
<th>UE&lt;sub&gt;cpu&lt;/sub&gt;</th>
<th>SE&lt;sub&gt;cpu&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ursa-EJF</td>
<td>464.00</td>
<td>208.21</td>
<td>99.57</td>
<td>86.60</td>
</tr>
<tr>
<td>Ursa-SRJF</td>
<td>473.50</td>
<td>170.64</td>
<td>98.89</td>
<td>86.08</td>
</tr>
<tr>
<td>YARN+Ursa</td>
<td>842.92</td>
<td>443.80</td>
<td>44.15</td>
<td>89.97</td>
</tr>
<tr>
<td>YARN+Spark</td>
<td>1072.66</td>
<td>435.00</td>
<td>67.92</td>
<td>83.84</td>
</tr>
<tr>
<td>Capacity</td>
<td>511.00</td>
<td>226.16</td>
<td>99.77</td>
<td>78.66</td>
</tr>
<tr>
<td>Tetris</td>
<td>562.33</td>
<td>254.52</td>
<td>98.62</td>
<td>70.02</td>
</tr>
<tr>
<td>Tetris2</td>
<td>506.00</td>
<td>240.83</td>
<td>99.71</td>
<td>79.75</td>
</tr>
</tbody>
</table>

Using monotasks alone

Using other scheduling algorithms

Over-subscription of CPU

<table>
<thead>
<tr>
<th>Subscription ratio</th>
<th>makespan (YARN+Ursa)</th>
<th>avgJCT (YARN+Ursa)</th>
<th>makespan (YARN+Spark)</th>
<th>avgJCT (YARN+Spark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>842.92</td>
<td>443.80</td>
<td>1072.66</td>
<td>435.00</td>
</tr>
<tr>
<td>2</td>
<td>637.96</td>
<td>345.99</td>
<td>872.67</td>
<td>341.77</td>
</tr>
<tr>
<td>4</td>
<td>596.66</td>
<td>325.32</td>
<td>892.83</td>
<td>365.30</td>
</tr>
</tbody>
</table>
Conclusions

Ursa:

• A framework for both resource scheduling and job execution
• Handles jobs with frequent fluctuations in resource usage
• Captures dynamic resource needs at runtime and enables fine-grained, timely scheduling
• Achieves high resource utilization, which is translated into significantly improved makespan and average JCT
Thank You

Contact: Tatiana Jin (tjin@cse.cuhk.edu.hk)