D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications

Xunyun Liu and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia

December, 2017
Outline

1 Background
   - Stream Processing
   - Apache Storm
   - Issues with the Existing Scheduler

2 Solution Overview
   - Our Approach
   - Framework Overview

3 Dynamic Resource-Efficient Scheduling
   - Problem Formulation
   - Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
Outline

1 Background
   • Stream Processing
   • Apache Storm
   • Issues with the Existing Scheduler

2 Solution Overview
   • Our Approach
   • Framework Overview

3 Dynamic Resource-Efficient Scheduling
   • Problem Formulation
   • Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
Stream Processing

Stream Data

- Arrive continuously in real time, possible infinite
- Various data sources & various data structures
- Transient value & short data lifespan
- Asynchronous & unpredictable

Table: Example usages of stream data versus batch data

<table>
<thead>
<tr>
<th></th>
<th>Batch Data</th>
<th>Stream Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Analysis</td>
<td>What happened an hour ago?</td>
<td>What is happening in the system?</td>
</tr>
<tr>
<td>Billing Analysis</td>
<td>How much did a user spend in the last billing period?</td>
<td>Notify a user that the billing limit is approaching</td>
</tr>
<tr>
<td>Fraud Prevention</td>
<td>Audit / forensic evidence</td>
<td>Intervene to stop the ongoing fraud</td>
</tr>
</tbody>
</table>

Volume
(Terabytes, Petabytes)

Velocity
(Real-time, Low latency)

Batch Data
(Supported by Hadoop, DBMS)

Stream Data
(Supported by Data Stream Management System)

Variety
(Social Networks, logs, Sensor data)
Stream processing is a technique that allows for the collection, integration, analysis, visualization, and system integration of stream data in real time, powering on-the-fly analytic that leads to immediately actionable insights.

- Process-once-arrival
  - Queries over recent data on rolling window
  - Generally independent computations
  - Incremental update of results
- Queries run continuously unless being explicitly terminated
- Suitable for latency-sensitive scenarios
Stream Processing

- Logic level
  - Inter-connected operators
  - Data streams flow through these operators to undergo different types of computation

- Middleware level
  - Data Stream Management System (DSMS)
  - Apache Storm, Samza ...

- Infrastructure level
  - A set of distributed hosts in cloud or cluster environment
  - Organised in Master/Slave model

![Distributed Stream Processing System](image)
The structural view and logical view of a Storm Cluster

(a) Structural view
(b) Logical view
Task Scheduling in Apache Storm

- **Static schedulers**
  - **Default scheduler**
    - Assign executors as evenly as possible between all the workers
    - Round-robin procedure

- **Dynamic scheduler**
  - Throughput-oriented schedulers [2, 3]
  - Latency-oriented schedulers [4, 5, 6]
  - Communication-reduction schedulers [7, 8, 9]

- Peng’s Resource Aware Scheduler (RAS) [1]
Issues with Existing Schedulers

- **Static assignment**
  - Not able to tackle runtime changes
  - Require users’ intervention to specify resource requirements

- **Resource agnostic**
  - Mismatch of task resource demands and node resource availability
  - Leading to over/under utilisation

- **Load balancing design**
  - Endeavour to distribute the workload as evenly as possible
  - Unable to perform resource consolidation when the input is small
  - Collocating multiple applications results in resource contention
1 Background
   - Stream Processing
   - Apache Storm
   - Issues with the Existing Scheduler

2 Solution Overview
   - Our Approach
   - Framework Overview

3 Dynamic Resource-Efficient Scheduling
   - Problem Formulation
   - Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
Our Approach

Dynamic Resource-efficient Scheduling

A user-transparent mechanism in DSMS to schedule streaming applications as compact as possible, matching the resource demands of streaming tasks to the capacity of distributed nodes, so that fewer resources are consumed to achieve the same performance target.

Core contributions:

- Profile streaming tasks at runtime to get accurate resource demands
- Model and solve the scheduling problem as a bin-packing variant to enable resource consolidation
- Reschedule the application automatically with a MAPE loop
D-Storm Framework

The extended D-Storm architecture

Nimbus
- Topology Adapter
- Custom Scheduler
  - D-Storm Scheduler

Worker Node
- Supervisor
  - Worker Process
    - Task Wrapper
      - Task
      - Task
      - Task

Supervisor
- Worker Process
  - Task Wrapper
    - Task
  - Task Wrapper
    - Task
  - Task Wrapper
    - Task

Zookeeper
- System Analyser
  - Scheduling Solver
    - Metric Flow
    - Control Flow
    - Data Stream Flow

Task Wrapper:
- A middle tier between task and executor abstractions
- Obtain task CPU usages
- Log communication traffics

System Analyser:
- A boundary checker on metrics
- Determine whether the current system state is normal
- Two abnormal states:
  - Unsatisfactory performance
  - Consolidation required
The extended D-Storm architecture

Task Wrapper:
- A middle tier between task and executor abstractions
- Obtain task CPU usages
- Log communication traffics
The extended D-Storm architecture

Task Wrapper:
- A middle tier between task and executor abstractions
- Obtain task CPU usages
- Log communication traffics

System Analyser:
- A boundary checker on metrics
- Determine whether the current system state is normal
- Two abnormal states:
  - Unsatisfactory performance
  - Consolidation required

- Task Wrapper: A middle tier between task and executor abstractions
- Obtain task CPU usages
- Log communication traffics

- System Analyser: A boundary checker on metrics
- Determine whether the current system state is normal
- Two abnormal states:
  - Unsatisfactory performance
  - Consolidation required

- Task Wrapper: A middle tier between task and executor abstractions
- Obtain task CPU usages
- Log communication traffics

- System Analyser: A boundary checker on metrics
- Determine whether the current system state is normal
- Two abnormal states:
  - Unsatisfactory performance
  - Consolidation required
D-Storm Framework

The extended D-Storm architecture

Scheduling Solver:
- Invoked by the System Analyser reporting an abnormal system state
- Perform scheduling calculation, devising a new plan

- Nimbus
- Worker Node
- Supervisor
- Worker Process
  - Task Wrapper
  - Task
- Custom Scheduler
- D-Storm Scheduler
- Topology Adapter
- Task Wrapper
  - Task
- Task Wrapper
  - Task
- D-Storm Scheduler
- Data Stream Flow
- Control Flow
- Metric Flow
- Task Wrapper
  - Task
- Worker Node
- Supervisor
- Worker Process
  - Task Wrapper
  - Task
- Worker Process
  - Task Wrapper
  - Task
- System Analyser
- Zookeeper

Scheduling Solver:
Invoked by the System Analyser reporting an abnormal system state
Perform scheduling calculation, devising a new plan

Xunyun Liu et al. (CLOUDS Lab)
Scheduling Solver:
- Invoked by the System Analyser reporting an abnormal system state
- Perform scheduling calculation, devising a new plan

D-Storm Scheduler:
- Implement the IScheduler interface
- Put the new plan into effect
The extended D-Storm architecture

Nimbus
- Topology Adapter

Custom Scheduler
- D-Storm Scheduler

Worker Node
- Supervisor
- Worker Process
  - Task Wrapper
    - Task

Supervisor
- Task Wrapper
  - Task

Task Wrapper
- Task

Metric Flow
- Control Flow
- Data Stream Flow

Scheduling Solver:
- Invoked by the System Analyser reporting an abnormal system state
- Perform scheduling calculation, devising a new plan

D-Storm Scheduler:
- Implement the IScheduler interface
- Put the new plan into effect

Topology Adapter:
- Mask changes made for profiling

Scheduling Solver:
- Invoked by the System Analyser reporting an abnormal system state
- Perform scheduling calculation, devising a new plan

D-Storm Scheduler:
- Implement the IScheduler interface
- Put the new plan into effect

Topology Adapter:
- Mask changes made for profiling
Outline

1 Background
   • Stream Processing
   • Apache Storm
   • Issues with the Existing Scheduler

2 Solution Overview
   • Our Approach
   • Framework Overview

3 Dynamic Resource-Efficient Scheduling
   • Problem Formulation
   • Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
Problem Formulation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>The number of tasks to be assigned</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>Task $i$, $i = 1, \ldots, n$</td>
</tr>
<tr>
<td>$m$</td>
<td>The number of available worker nodes in the cluster</td>
</tr>
<tr>
<td>$\nu_i$</td>
<td>Worker node $i$, $i = 1, \ldots, \nu$</td>
</tr>
<tr>
<td>$W^t_{\nu_i}$</td>
<td>CPU capacity of $\nu_i$, measured in a point-based system</td>
</tr>
<tr>
<td>$W^m_{\nu_i}$</td>
<td>Memory capacity of $\nu_i$, measured in Mega Bytes (MB)</td>
</tr>
<tr>
<td>$\omega^c_{\tau_i}$</td>
<td>Total CPU requirement of $\tau_i$ in points</td>
</tr>
<tr>
<td>$\omega^m_{\tau_i}$</td>
<td>Total memory requirement of $\tau_i$ in Mega Bytes (MB)</td>
</tr>
<tr>
<td>$\rho^c_{\tau_i}$</td>
<td>Unit CPU requirement for $\tau_i$ to process a single tuple</td>
</tr>
<tr>
<td>$\rho^m_{\tau_i}$</td>
<td>Unit memory requirement for $\tau_i$ to process a single tuple</td>
</tr>
<tr>
<td>$\xi_{\tau_i, \tau_j}$</td>
<td>The size of data stream transmitting from $\tau_i$ to $\tau_j$</td>
</tr>
<tr>
<td>$\Theta_{\tau_i}$</td>
<td>The set of upstream tasks for $\tau_i$</td>
</tr>
<tr>
<td>$\Phi_{\tau_i}$</td>
<td>The set of downstream tasks for $\tau_i$</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>The volume of inter-node traffic within the cluster</td>
</tr>
<tr>
<td>$V_{used}$</td>
<td>The set of used worker nodes in the cluster</td>
</tr>
<tr>
<td>$m_{used}$</td>
<td>The number of used worker nodes in the cluster</td>
</tr>
</tbody>
</table>

minimise $\kappa(\xi, \mathbf{x}) = \sum_{i,j \in \{1, \ldots, n\}} \xi_{\tau_i, \tau_j}(1 - \sum_{k \in \{1, \ldots, m\}} x_{i,k} \cdot x_{j,k})$

subject to $\sum_{k=1}^{m} x_{i,k} = 1, \quad i = 1, \ldots, n,$

$\sum_{i=1}^{n} \omega^c_{\tau_i} x_{i,k} \leq W^c_{\nu_k} \quad k = 1, \ldots, m,$ (1)

$\sum_{i=1}^{n} \omega^m_{\tau_i} x_{i,k} \leq W^m_{\nu_k} \quad k = 1, \ldots, m,$

where $\mathbf{x}$ is the control variable that stores the task placement in a binary form: $x_{i,k} = 1$ if and only if task $\tau_i$ is assigned to machine $\nu_k$. 

Xunyun Liu et al. (CLOUDS Lab)
Algorithm 1: The multidimensional FFD heuristic scheduling algorithm

Input: A task set \( \vec{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\} \) to be assigned

Output: A machine set \( \vec{\nu} = \{\nu_1, \nu_2, \ldots, \nu_{m_{\text{used}}}\} \) with each machine hosting a disjoint subset of \( \vec{\tau} \), where \( m_{\text{used}} \) is the number of used machines

1. Sort available nodes in descending order by their resource availability as defined in Eq. (4)
2. \( m_{\text{used}} \leftarrow 0 \)
3. \textbf{while} there are tasks remaining in \( \vec{\tau} \) to be placed \textbf{do}
4.  Start a new machine \( \nu_m \) from the sorted list;
5.  \textbf{if} there are no available nodes \textbf{then}
6.     return Failure
7.  Increase \( m_{\text{used}} \) by 1
8.  \textbf{while} there are tasks that fit into machine \( \nu_m \) \textbf{do}
9.      \textbf{foreach} \( \tau \in \vec{\tau} \) \textbf{do}
10.         Calculate \( \varrho(\tau_i, \nu_m) \) according to Eq. (5)
11.         Sort all viable tasks based on their priority
12.          Place the task with the highest \( \varrho(\tau_i, \nu_m) \) into machine \( \nu_m \)
13.          Remove the task from \( \vec{\tau} \)
14.          Update the remaining capacity of machine \( \nu_m \)
15. return \( \vec{\nu} \)

Algorithm Design:
- Based on greedy heuristic
- Large machines are utilised first
- Task priority updated dynamically
- Seek to increase the sum of internal communications at the current node

Complexity Analysis:
- Line 1 requires at most quasilinear time \( O(m \log(m)) \)
- Line 8 to Line 14 will be repeated for at most \( n \) times
- Line 10 consumes linear time of \( n \)
- The worst case complexity: \( O(m \log(m) + n^2 \log(n)) \)
Scheduling Algorithm

Algorithm 2: The multidimensional FFD heuristic scheduling algorithm

Input: A task set \( \bar{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\} \) to be assigned

Output: A machine set \( \bar{\nu} = \{\nu_1, \nu_2, \ldots, \nu_{m_{\text{used}}}\} \) with each machine hosting a disjoint subset of \( \bar{\tau} \), where \( m_{\text{used}} \) is the number of used machines

1. Sort available nodes in descending order by their resource availability as defined in Eq. (4)
2. \( m_{\text{used}} \leftarrow 0 \)
3. while there are tasks remaining in \( \bar{\tau} \) to be placed do
   4. Start a new machine \( \nu_m \) from the sorted list;
   5. if there are no available nodes then
      return Failure
   6. Increase \( m_{\text{used}} \) by 1
   7. while there are tasks that fit into machine \( \nu_m \) do
      8. foreach \( \tau \in \bar{\tau} \) do
         9. Calculate \( \varrho(\tau_i, \nu_m) \) according to Eq. (5)
         10. Sort all viable tasks based on their priority
         11. Place the task with the highest \( \varrho(\tau_i, \nu_m) \) into machine \( \nu_m \)
         12. Remove the task from \( \bar{\tau} \)
         13. Update the remaining capacity of machine \( \nu_m \)
   14. return \( \bar{\nu} \)

Algorithm Design:

- Based on greedy heuristic
- Large machines are utilised first
- Task priority updated dynamically
- Seek to increase the sum of internal communications at the current node

Complexity Analysis:

- Line 1 requires at most quasilinear time \( O(m\log(m)) \)
- Line 8 to Line 14 will be repeated for at most \( n \) times
- Line 10 consumes linear time of \( n \)
- The worst case complexity: \( O(m\log(m) + n^2\log(n)) \)
Outline

1 Background
   • Stream Processing
   • Apache Storm
   • Issues with the Existing Scheduler

2 Solution Overview
   • Our Approach
   • Framework Overview

3 Dynamic Resource-Efficient Scheduling
   • Problem Formulation
   • Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
Experiment Setup

- **Nectar IaaS Cloud**
  - 16 worker nodes: 2 VCPUs, 6GB RAM and 30GB disk
  - 1 Nimbus, 1 Zookeeper, 1 Kestrel node

- **Extended on Apache Storm v1.0.2, with Oracle JDK 8, update 121**

- **Two test applications**
  - Synthetic linear application
  - Twitter Sentiment Analysis topology

- **Profiling environment**
**Experiment Setup**

- **Synthetic linear application**
  - 3 synthetic bolts concatenated in serial
  - Produce different patterns of resource consumption, such as CPU bound and I/O bound computations.

- **Twitter Sentiment Analysis topology**
  - 11 operators constituting a tree-like topology
  - Sentimental score calculated using AFFINN, a list of words associated with pre-defined sentiment values.

**Table: Evaluated configurations and their values**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_s$ (synthetic topology)</td>
<td>10, 20, 30, 40</td>
</tr>
<tr>
<td>$S_s$ (synthetic topology)</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>$T_s$ (synthetic topology)</td>
<td>4, 8, 12, 16</td>
</tr>
<tr>
<td>Number of tasks (realistic application)</td>
<td>11, 21, 31, 41</td>
</tr>
</tbody>
</table>
Applicability Evaluation

Research Question

Whether D-Storm is applicable to different types of streaming applications and capable of reducing the total amount of inter-node communication?

(a) Varying \( C_s \) (Synthetic App)
(b) Varying \( S_s \) (Synthetic App)
(c) Varying \( T_s \) (Synthetic App)
(d) Varying number of tasks (Real App)

Figure: The relative change of the inter-node communication, with the baseline produced by the Default Storm scheduler
Research Question

How is D-Storm performing when the input load decreases and the cluster is under-utilized?

Figure: Cost efficiency analysis of D-Storm scheduler as the input load decreases. Fig. 2b is calculated based on the price of m1.medium instances in the AWS Sydney region.
Research Question
How long does it take for D-Storm to schedule relatively large streaming applications?

Table: The time consumed in creating schedules by different strategies (unit: milliseconds)

<table>
<thead>
<tr>
<th>Schedulers</th>
<th>Test Cases</th>
<th>Parallelism Intensive Synthetic App</th>
<th>Twitter Sentiment Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_s=4$</td>
<td>$T_s=8$</td>
<td>$T_s=12$</td>
</tr>
<tr>
<td>D-Storm</td>
<td>16.4</td>
<td>16.2</td>
<td>16.8</td>
</tr>
<tr>
<td>Static Scheduler</td>
<td>5.2</td>
<td>5.7</td>
<td>5.5</td>
</tr>
<tr>
<td>Default Scheduler</td>
<td>0.72</td>
<td>0.77</td>
<td>0.75</td>
</tr>
</tbody>
</table>
Outline

1 Background
   - Stream Processing
   - Apache Storm
   - Issues with the Existing Scheduler

2 Solution Overview
   - Our Approach
   - Framework Overview

3 Dynamic Resource-Efficient Scheduling
   - Problem Formulation
   - Scheduling Algorithm

4 Evaluation

5 Related Work

6 Conclusions and Future Work
In comparison with existing schedulers, we conclude that our work is dynamic resource-aware, communication-aware, self-adaptive, user-transparent and cost-efficient.

Table: Related work comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Resource-aware</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Communication-aware</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Self-adaptive</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>User-transparent</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Cost-efficient</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
Outline

1. Background
   - Stream Processing
   - Apache Storm
   - Issues with the Existing Scheduler

2. Solution Overview
   - Our Approach
   - Framework Overview

3. Dynamic Resource-Efficient Scheduling
   - Problem Formulation
   - Scheduling Algorithm

4. Evaluation

5. Related Work

6. Conclusions and Future Work
Conclusions

- We proposed a resource-efficient algorithm for scheduling streaming application with bin-packing formulation.
- We implemented D-Storm, a prototype scheduler for algorithm validation.
- The compact scheduling strategy leads to the reduction of resource usages as well as the minimisation of inter-node communication.
- We adjust the scheduling plan to the runtime changes while remaining sheer transparent to the upper-level application logic.

Outlook

- Use meta-heuristics to find a better solution
- Take the network characteristics into account during the scheduling process, placing intensive task communications on links with higher bandwidth.
Any questions?


