Parallelizing Stateful Operators in a Distributed Stream Processing System: How, Should You and How Much?
The Problem

- In several real-world distributed stream processing applications a small subset of the constituent operators are often the performance bottleneck for the entire application.

- The performance of the bottleneck operators is limited by the capacity of the compute node or the core on which the operator is hosted even when some other resources are idle.

- The compilers, even if they are aware of the resources, throughput requirements and the underlying architecture, are not equipped to exploit this information.
The Problem – Scenario 1

- Given at compile time:
  - A simple data stream processing application with few bottleneck operators
  - A large cluster of compute nodes
  - Need for high throughput

- The compiler is unable to exploit the large number of available compute nodes
The Problem – Scenario 2

Given at compile time:
- A simple data stream processing application with few bottleneck operators
- A small set of multicore processors
- Need for high throughput

The compiler is unable to exploit the multiple cores
Overall Approach

- Given a stream processing operator and its parallel implementation. A compiler would be responsible for:

  1. Identifying operators that should be parallelized
  2. Determining the degree of parallelism
  3. Generating the code based on above determinations and the architecture of the underlying hardware.

```java
com.ibm.streams.RocketScience
com.ibm.streams.parallel.RocketScience
```

Diagram:

- Split
- Process
- Merge
- Shared state
Generated parallelized operator for a cluster of commodity compute nodes
Code generation for a multicore

- Generated parallelized operator for a multicore
Challenge

• The level of parallelism:
  – More process operators, better performance?
  – No!
  • Contention when accessing shared variables
  • Cost of accessing the shared variables
  • Computations per tuple in the process operator
  • Shared variable accesses per tuple in the process operator
  • Limitation of incoming stream rate
  – How to decide a proper parallelism level?
Parallelization Model

• Access to a single shared variable is synchronized

• A specific shared variable, Lock, is used to guarantee synchronization when multiple shared variables are involved.
  – Only one operator can get the lock.

• Parallel operator Logic:
  – Pre-process (cost : t₀)
  – Acquire the lock (optional)
  – Critical process (cost : t₁)
  – Release the lock (optional)
  – Post-process (cost : t₂)

• Shared variable Access type:
  – Non-synchronized read/write (read/write outside the lock)
  – Synchronized read/write (read/write within the lock)
Parallelization Model (cont.)

• We use \(<N, n_r, n_w, s_r, s_w>\) to denote a specific configuration of a parallel operator
  – \(N\): number of parallel operators
  – \(n_r\): number of non-synchronized reads
  – \(n_w\): number of non-synchronized writes
  – \(s_r\): number of synchronized reads
  – \(s_w\): number of synchronized writes
Model Analysis

• Single Operator
  – \( <1, 0, 0, 0, 0> \): 1 operator, local memory access
    • \( TH = \frac{1}{\text{general processing cost}} = \frac{1}{t_0 + t_1 + t_2} \)

  – \( <1, n_r, n_w, s_r, s_w> \): 1 operator, hybrid reads/writes
    • Note, in single node case, non-synchronized access = synchronized access
    • \( TH = \frac{1}{t_0 + t_1 + t_2 + (n_r + s_r) \times \text{read_cost} + (n_w + s_w) \times \text{write_cost}} \)

  – The throughput is also bounded by the incoming stream rate, e.g. \( TH \leq R_{\text{in}} \)
Model Analysis (cont.)

• Multiple Operators
  – \(<N, 0, 0, 0, 0>\): N parallel operators, no shared variable access
    • \(TH = \min(N/(t_0+t_1+t_2), R_{in})\)
  – \(<N, n_r, n_w, 0, 0>\): N parallel operators, \(n_r\) non-synchronized reads and \(n_w\) non-synchronized writes
    • \(TH = \min(N/(t_0+t_1+t_2+n_r*\text{read}\_\text{cost}+n_w*\text{write}\_\text{cost}), R_{in})\)
  – \(<N, n_r, n_w, s_w, s_r>\): N parallel operators, \(n_r/n_w\) non-synchronized reads/writes and \(s_w/s_r\), synchronized reads/writes
    • Operators compete for the lock, therefore contention happens.
Model Analysis (cont.)

Without lock contention

With lock contention
• Cost of $<N, n_r, n_w, s_r, s_w>$:
  – Cost = $t_0 + t_1 + t_2 + (n_r + s_r) \times \text{read\_cost} + (s_r + s_w) \times \text{write\_cost} + \text{wait\_time}$

  \[ \text{wait queue} \rightarrow \begin{array}{c}
  \text{OP3} \\
  \text{OP5} \\
  \text{OP2} \\
  \text{OP1}
  \end{array} \]

  \[ \text{Current lock holder} \rightarrow \begin{array}{c}
  \text{OP4}
  \end{array} \]
  \[ \text{Wait time of OP2} = 3 \times t_1 \]

  – Maximal possible throughput: $1/t_1$
Verification of Parallelization Model: Setup

• Pre-processing, critical processing and post-processing are set to 20ms  
  – E.g. $t_0=t_1=t_2=20\text{ms}$

• Input rate of approximately 80 tuples/second

• Metric : Throughput (number of processed tuples per second)

• Read cost = 0.153ms, Write cost = 0.185ms (determined experimentally)
• Multi-operator non-synchronized access

\[ T_{<3,50,0,0,0>} = \min\left(\frac{3 \times 1000}{60 + 50 \times 0.153}, 80\right) \approx 44 \]
Verification of Parallelization Model: Experiment (cont.)

- Multi-operator hybrid access

\[
\eta_{\leq 5,5,5,5} > \frac{60 + 10 \times 0.153 + 10 \times 0.185}{20 + 5 \times 0.153 + 5 \times 0.183} = 3
\]
Study of a Real Application

• Moving KNN computation
  – Traffic monitoring, Mobile games, P2P sharing
Implementation of Moving KNN in Spade

• Building a Grid-Index by exploiting shared variables

<table>
<thead>
<tr>
<th>G11</th>
<th>G12</th>
<th>G13</th>
</tr>
</thead>
<tbody>
<tr>
<td>G21</td>
<td>G22</td>
<td>G23</td>
</tr>
<tr>
<td>G31</td>
<td>G32</td>
<td>G33</td>
</tr>
</tbody>
</table>

Shared variable list
- G11
- G12
- G13
- ...
- G33
Approximate KNN vs KNN

APP-KNN: lock (G41, G24), low risk of contention
KNN: lock(G41, G13, G14, G23, G24, G33, G34), high risk of contention
KNN Application: Experimental Setup

• 8 hosts for running the application
• We used 100 grid regions and 1,000 – 10,000 objects per grid region
• Objects are uniformly distributed in the space and each object has a random speed
• For each incoming tuple, we need to update the object’s location and return its current KNN neighbors (K=100)
• Metric : Throughput (number of processed tuples per second)
Experiment Result

- Effect of number of operators
• Number of objects affects the computation cost of KNN
• We change the data per grid (on average) from 1,000 to 10,000
Our aim with this work is to provide advanced developers/toolkit creators a framework for creating a parallel version of a subset of operators.

Operators that follow this parallelization framework would be known to the compiler and thus the compiler can transparently replace an operator with the parallel version, if the throughput requirement dictate the same.

Our theoretical framework allows compiler to reason about parallelism – Should the operator be parallelized and if yes, what should be the degree of parallelism.

The split-process-merge is just one parallelization possibility, others that do not necessarily utilize shared state or follow another topology are also possible.

Runtime parallelism and automatic generation of parallelized version of operator are some other topics of interest to us.