RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing

Cagri Balkesen, Nihal Dindar, Matthias Wetter, Nesime Tatbul
Motivation – Processing an Event Stream

- Continuous data stream of emotional states: ..., <90>, <89>, <79>, <12>, <51>, ...
- My cerebral flow meter produces 5’000’000 events/second.
- I have no hardware available to store such a huge amount of data.
- I like to recognize a pattern within this event stream that expresses a wish.
Motivation – The Wish Pattern
I have a program that is able to process such events and detect this pattern.

Unfortunately, that program is too slow

- It can only handle 500’000 events/second.

The program is single-threaded.

I have a multi-core processor available.
Outline

- Background
  - Sequential pattern matching
- Parallel pattern matching
  - State-based parallelism
  - Run-based parallelism
- Performance Comparison
- Conclusion
BACKGROUND:
SEQUENTIAL PATTERN MATCHING
Motivation – Real World Applications

- Financial market data analysis
- Operational business intelligence
- Security applications
- Click stream analysis
- Electronic health systems
A Pattern Matching Query

- PATTERN (A B+ C* D E)

- DEFINE

  - A AS (A.emotion > 40 AND A.emotion < 60)
  - B AS (B.emotion > 90)
  - C AS (C.emotion < PREV(C.emotion))
  - D AS (D.emotion < 10)
  - E AS (E.emotion > A.emotion)
Compile a Query into a Finite State Machine (FSM)

- **PATTERN (A B+ C* D E)**
- **DEFINE**
  - A AS (A.emotion > 40 AND A.emotion < 60)
  - B AS (B.emotion > 90)
  - C AS (C.emotion < PREV(C.emotion))
  - D AS (D.emotion < 10)
  - E AS (E.emotion > A.emotion)
Sequential Pattern Matching with an Example

Query:

pA: A.emotion < 60 AND A.emotion > 40
pB: B.emotion > 90
pC: C.emotion < PREV(C.emotion)
D: D.emotion < 10
pE: E.emotion > A.emotion

Processing:

for each event $e$
create a new run
update runs

<table>
<thead>
<tr>
<th>event</th>
<th>run ($r_{id}$)</th>
<th>&lt;curr_state; PM&gt;</th>
<th>prd</th>
<th>&lt;curr_state'; PM'&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1(55)</td>
<td>r1</td>
<td>&lt; s0; () &gt;</td>
<td>pA</td>
<td>&lt; A; (a1) &gt;</td>
</tr>
<tr>
<td>e2(99)</td>
<td>r2</td>
<td>&lt; s0; () &gt;</td>
<td>pA</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>r1</td>
<td>&lt; A; (a1) &gt;</td>
<td>pB</td>
<td>&lt; B; (a1, b2) &gt;</td>
</tr>
<tr>
<td>e3(100)</td>
<td>r3</td>
<td>&lt; s0; () &gt;</td>
<td>pA</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>r1</td>
<td>&lt; B; (a1, b2) &gt;</td>
<td>pB</td>
<td>&lt; B; (a1, b2, b3)</td>
</tr>
</tbody>
</table>
Sequential Pattern Matching

- evaluate($\text{pred}_A$, PM$_1$, e$_1$)
- evaluate($\text{pred}_B$, PM$_1$, e$_2$)
- evaluate($\text{pred}_A$, PM$_2$, e$_2$)
- evaluate($\text{pred}_B$, PM$_1$, e$_3$)
- evaluate($\text{pred}_C$, PM$_1$, e$_3$)
- evaluate($\text{pred}_D$, PM$_1$, e$_3$)
- evaluate($\text{pred}_A$, PM$_3$, e$_3$)
- evaluate($\text{pred}_B$, PM$_1$, e$_4$)
- evaluate($\text{pred}_C$, PM$_1$, e$_4$)
- evaluate($\text{pred}_D$, PM$_1$, e$_4$)
- evaluate($\text{pred}_A$, PM$_4$, e$_4$)
- evaluate($\text{pred}_C$, PM$_1$, e$_5$)
- evaluate($\text{pred}_D$, PM$_1$, e$_5$)
- evaluate($\text{pred}_B$, PM$_4$, e$_5$)
- evaluate($\text{pred}_A$, PM$_5$, e$_5$)
- evaluate($\text{pred}_C$, PM$_1$, e$_6$)
- evaluate($\text{pred}_D$, PM$_1$, e$_6$)
- evaluate($\text{pred}_E$, PM$_1$, e$_6$)
Parallel Pattern Matching

- Assumption: Event evaluation is a bottleneck
- Idea: Parallelize, but how?
  - by input events
    - Such as PARTITION BY
  - by state predicates
    - Similar to pipelined approach
  - by runs
    - The new approach we propose
STATE-BASED PARALLELIZATION
State-based Parallelization

- Idea: Each processing unit is responsible to evaluate an FSM state

- Communication Overhead
- Replication of input stream
- Query Dependence
- Load Imbalance
State-based Parallelization

- evaluate($\text{pred}_A$, $\text{PM}_1$, $e_1$)
- evaluate($\text{pred}_B$, $\text{PM}_1$, $e_2$)
- evaluate($\text{pred}_A$, $\text{PM}_2$, $e_2$)
- evaluate($\text{pred}_B$, $\text{PM}_1$, $e_3$)
- evaluate($\text{pred}_C$, $\text{PM}_1$, $e_3$)
- evaluate($\text{pred}_D$, $\text{PM}_1$, $e_3$)
- evaluate($\text{pred}_A$, $\text{PM}_3$, $e_3$)
- evaluate($\text{pred}_B$, $\text{PM}_1$, $e_4$)
- evaluate($\text{pred}_C$, $\text{PM}_1$, $e_4$)
- evaluate($\text{pred}_D$, $\text{PM}_4$, $e_4$)
- evaluate($\text{pred}_A$, $\text{PM}_4$, $e_4$)
- evaluate($\text{pred}_C$, $\text{PM}_1$, $e_5$)
- evaluate($\text{pred}_D$, $\text{PM}_1$, $e_5$)
- evaluate($\text{pred}_B$, $\text{PM}_4$, $e_5$)
- evaluate($\text{pred}_A$, $\text{PM}_5$, $e_5$)
- evaluate($\text{pred}_C$, $\text{PM}_1$, $e_6$)
- evaluate($\text{pred}_D$, $\text{PM}_1$, $e_6$)
- evaluate($\text{pred}_E$, $\text{PM}_1$, $e_6$)
RUN-BASED PARALLELIZATION
Run-Based Parallelization Approach

- Data Stream is divided into batches of events

- Each batch is sent to exactly one processing unit.

- A processing unit detects all matches that start with events in that batch.

- How to deal with matches that overlap batches?
Run-Based Parallelization Approach

- Data Stream is divided into batches that overlap

```
e_{14} e_{13} e_{12} e_{11} e_{10} e_{11} e_{10} e_{9} e_{8} e_{7} e_{8} e_{7} e_{6} e_{5} e_{4} e_{5} e_{4} e_{3} e_{2} e_{1}
```

Batch (N) Shared Part (S)

- Each batch is sent to exactly one processing unit.

- Each processing unit detects all matches that start in the first N-S events in a batch.

- \( S = \text{MAXIMAL\_MATCH\_LENGTH} - 1 \)

- \( S \leq \frac{1}{2} \cdot N \)
Run-Based Parallelization Approach Overview

... e5, e4, e3, e2, e1

... e8, e7, e6, e5, e4

... e11, e10, e9, e8, e7

... e1, e7, e8, e7

No Replication of Input Stream

No Communication Overhead

Query Independent

Load Imbalance, but Fair Scheduling
Run-Based Parallelization

- evaluate(pred$_A$, PM$_1$, e$_1$)
- evaluate(pred$_B$, PM$_1$, e$_2$)
- evaluate(pred$_A$, PM$_2$, e$_2$)
- evaluate(pred$_B$, PM$_1$, e$_3$)
- evaluate(pred$_C$, PM$_1$, e$_3$)
- evaluate(pred$_D$, PM$_1$, e$_3$)
- evaluate(pred$_A$, PM$_3$, e$_3$)
- evaluate(pred$_B$, PM$_1$, e$_4$)
- evaluate(pred$_C$, PM$_1$, e$_4$)
- evaluate(pred$_D$, PM$_1$, e$_4$)
- evaluate(pred$_A$, PM$_4$, e$_4$)
- evaluate(pred$_A$, PM$_4$, e$_4$)
- evaluate(pred$_C$, PM$_1$, e$_5$)
- evaluate(pred$_D$, PM$_1$, e$_5$)
- evaluate(pred$_B$, PM$_4$, e$_5$)
- evaluate(pred$_A$, PM$_5$, e$_5$)
- evaluate(pred$_C$, PM$_1$, e$_6$)
- evaluate(pred$_D$, PM$_1$, e$_6$)
- evaluate(pred$_E$, PM$_1$, e$_6$)
PERFORMANCE COMPARISON
Experimental Setup

- **Datasets and Machine**
  - Synthetic data and NYSE data in a client/server setup
  - AMD Opteron CPU, 2.3 Ghz with 4-sockets/48-cores (Gigabit Ethernet)

- **Queries**
  - Fixed-length queries such as (PATTERN ABC)
  - Variable-length queries such as (PATTERN AB*C)

- **Implementations**
  - SEQ: sequential processing
  - SB: state-based approach
  - RB+: run-based approach with batching

- **Evaluation criteria:** Throughput

- **Speedup = throughput of parallelization approach / sequential processing**
State-based Parallelization

- Gap between achieved and ideal speedup increase?
  - Load imbalance

➤ State-based approach helps, but it does not scale!
State-based vs. Run-based (balanced Queries)

Run-based approach outperforms state-based approach even in case of balanced queries
State-based vs. Run-based (imbalanced Query)

Run-based approach scales well
Conclusions

- State-based parallelization can increase performance of pattern matching
- But speedup and scalability are limited due to:
  - Communication overhead
  - Query dependency
  - Replication of input stream
  - Load imbalance (bottleneck states)
- Run-based approach eliminates these problems
  - Improves performance linearly with almost an optimal speedup, up to the point where other factors become a bottleneck

Questions