Scalable Complex Event Processing: A Hybrid Parallelization Approach

Maor Yankovitch
Faculty of Computer Science
Technion, Israel Institute of Technology
Haifa, Israel
yankovitch@cs.technion.ac.il

Ilya Kolchinsky
Faculty of Computer Science
Technion, Israel Institute of Technology
Haifa, Israel
ikolchin@cs.technion.ac.il

Assaf Schuster
Faculty of Computer Science
Technion, Israel Institute of Technology
Haifa, Israel
assaf@cs.technion.ac.il

***Abstract*—The ability to promptly and efficiently detect arbitrarily complex patterns in massive real-time data streams is a crucial requirement for a wide range of modern applications. The ever-growing scale of these applications and the sophistication of the patterns involved makes it imperative to employ advanced solutions that can optimize pattern detection. One of the most prominent and well-established ways to achieve the above goal is to apply complex event processing (CEP) in a parallel manner, using a multi-core and/or a distributed environment. However, the inherent tightly coupled nature of CEP severely limits the scalability of the parallelization methods currently available.**

We introduce a novel parallelization mechanism for efficient complex event processing over data streams. This mechanism is based on a hybrid two-tier model combining multiple layers of parallelism. It thus allows for high scalability and fine-grained load balancing, while significantly reducing synchronization overhead. An extensive experimental evaluation on multiple real-life datasets shows that our approach consistently outperforms state-of-the-art CEP parallelization methods by a factor of up to two orders of magnitude.

I. Introduction

Complex event processing, (CEP) is a leading technology for robust and high-performance real-time detection of arbitrarily complex patterns in massive data streams ([9], [10], [12]). It is widely employed in many areas where extremely large amounts of streaming data are continuously generated and need to be promptly and efficiently analyzed on-the-fly. Online finance [11], credit card fraud detection [26], sensor networks [15], healthcare industry [6], and IoT applications [33] are among the many examples.

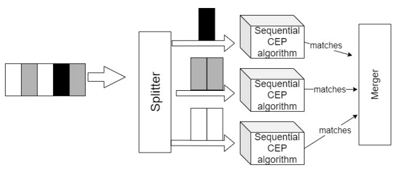
CEP engines treat the data items that make up the input streams as *primitive events* arriving from event sources. As new primitive events are observed, they are assembled into higher-level *complex events* that match the user-defined *patterns*. Detecting complex events generally consists of collecting primitive events and incrementally combining them into *partial pattern matches* using some type of *detection model*. As more events are added to a partial match, a full pattern match is eventually formed and reported. The most widely used detection models are non-deterministic finite automata ([2], [19], [31], [32]) and evaluation trees [23]. In these models, the loose order of constructing and extending partial matches is represented by the topology of an automaton or a tree, with the automaton's states or the tree's nodes denoting the full match subsets.

As discussed by multiple authors ([3], [19], [23]), the processing time, latency, and resource consumption of the CEP execution grows exponentially with the size of the pattern being detected. The main factor contributing to this growth is the need to explicitly examine a large fraction of event subsets of any size (up to pattern's size) to determine whether they comprise valid pattern matches. As the patterns are often characterized by their extreme length, complexity, and nesting level, this limitation presents a crucial performance bottleneck. The situation is exacerbated by the tight real-time constraints imposed on these systems, as well as by a common requirement to simultaneously process multiple patterns and streams. Therefore, advanced algorithmic solutions and sophisticated optimization techniques are essential for achieving an acceptable level of service quality.

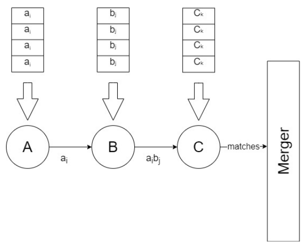
Parallelizing CEP evaluation flows is one of the most popular avenues for improving the performance of event processing applications. Various methods for allocating the workload of a CEP system to multiple execution units and managing their parallel execution have been proposed, addressing multi-core and fully distributed scenarios. These solutions can be roughly divided into two separate categories: data-parallel and state-parallel methods.

Data-parallel approaches ([5], [16], [21]) operate by splitting the input data stream into different partitions according to some predefined criteria and routing each partition to a dedicated CEP unit; this unit may be a thread, a process, or a separate machine. Each unit then executes the same sequential pattern matching algorithm. The pattern matches detected by each CEP unit are then merged and jointly returned to the end users. Fig. 1(a) presents an example of a parallel CEP system architecture implemented according to this paradigm. While this scheme has proven highly powerful and efficient in many cases, its inherent limitation lies in the difficulty of designing a good partitioning scheme. In particular, since any subset of data items can potentially represent a pattern match, at least a fraction of the sub-streams must be duplicated and delivered to multiple units to avoid missing results and to guarantee detection correctness.

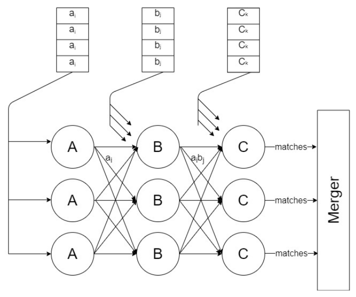
The second category of CEP parallelization methods is



(a) Data parallelism - Input is split into partitions and handled separately by each execution unit



(b) State parallelism - Each state is a separate execution unit that receives events of a specific type



(c) Hybrid parallelism - Two-layer approach combining both state-parallelism and data-parallelism

Fig. 1: **Parallelism classes**

known as *state-parallel* ([5], [8], [30]). This approach assigns a dedicated execution unit to each building block of the pattern detection model, which may be an NFA state or an evaluation tree node. Consequently, each unit is exclusively responsible for some functional part of the sequential pattern matching algorithm. Each unit receives a part of the input stream, which is usually events of a specific type. An example is shown in Fig. 1(b). This parallelization scheme avoids the data stream duplication problem that plagues data-parallel methods. However, it imposes a strict limit on the application scalability since the maximal number of execution units is a linear value bounded by the number of states or nodes.

In this paper, we propose and implement a new, third paradigm for parallelizing CEP applications, which we refer to as a *hybrid-parallel approach*. In a hybrid-parallel system, the execution units are organized in two layers, and the workload distribution proceeds in two stages. First, a state-parallel procedure allocates a set of execution units to each state according to its expected load. This procedure comprises the outer parallelism layer. The inner layer is manifested as a state executes a data-parallel routine that divides the input of its state between the individual units. This process is repeated continuously during the system run. In this way, the system can dynamically adapt to the ever-changing data arrival rates, system properties, and resource availability. Fig. 1(c) illustrates this parallelization scheme.

By providing two distinct layers of parallelism, our approach combines the strengths of data-parallel and state-parallel solutions, while overcoming their limitations. Unlike that of a pure state-parallel system, the degree of parallelism in a hybrid-parallel system is unbounded. On the other hand, no duplicate data transmission is required, since the *outer state-parallelization layer* mimics the evaluation flow of the state-parallel approach. It does not require duplicate input since the detection algorithm is sequential in its essence with only the states themselves running in parallel. The *inner data-parallelization layer* is designed to leverage the shared memory of a multi-core architecture to avoid the need for an explicit partitioning scheme. Units of the same state can directly access data stored in other units instead of having to duplicate such data. The hybrid-parallel approach also provides a significant degree of flexibility, since the outer parallelization layer can be deployed in a fully distributed share-nothing environment. An additional advantage of our proposed paradigm is a two-tier load balancing scheme that allocates different numbers of execution units to states, depending on their load. It can also dynamically adjust the volume of the input data processed by each unit.

The contributions of this paper can thus be summarized as follows:

- A novel paradigm for scalably and efficiently distributing CEP workloads between multiple execution units using a hybrid-parallel approach.
- A detailed NFA-based implementation of the above approach and athorough analysis of its performance.
- Practical extensions to the basic hybrid-parallel mechanism to cover *dynamic thread allocation* and *state fusion*. These extensions allow us to further improve the performance and resource utilization of the pattern detection process.
- An extensive experimental evaluation of our method, demonstrating its superiority over state-of-the-art CEP parallelization approaches.

The remainder of this paper is organized as follows. Section II provides background on CEP and introduces the notations used throughout the paper. In Section III we introduce the hybrid-parallel approach and present an NFA-based parallel CEP system using this approach. Section IV provides a detailed theoretical analysis of our solution. Section V describes important implementation details and properties of the system. Section VI describes two significant extensions of our basic method, to further improve its efficiency and scalability. We report the results of our extensive experimental study in Section VII. Section VIII discusses the related work and Section IX concludes the paper with our planned future work.

II. Background and Terminology

The functionality of a CEP system revolves around the basic notion of an *event*. An event is an indication that an action of interest happened at a specific point in time. CEP systems receive events as their main input, usually from a streaming source. Each event is typically associated with a single *event type*, which defines a set of *attributes* that an event contains.

Users interact with a CEP system by providing *patterns* that specify the combinations of events to be detected, that we also call *complex events*. A pattern typically conveys the following information: the *structure* of the complex event (e.g., whether the desired combination is a sequence of a predefined number of primitive events, a conjunction or disjunction thereof, or a more complex expression); the optional Boolean *conditions* that must be satisfied for the complex event to be considered a pattern match; and the *time window* W that defines the maximal difference between the occurrence time of the events in a pattern match, that is, the time frame within which the complex event is to be detected.

As an example, consider the following:

Example 1. A warehouse receives orders for specific items and delivers them to customers. These items have RFID tags that are scanned when certain actions occur, such as removing an item from storage, loading it onto a forklift, and registering it as ready for delivery. We are interested in detecting sets of items that were ordered recently (in the last hour) and are ready to be delivered in the next shipment.

Using the terminology defined above, we will represent each action on an item as an event with the type determined by the action type. For example, type S can represent taking the item from the storage, type O for ordering the item, type R for registering the item for delivery, and so on. The attributes of each such event could include the item ID, the employee performing the action, and the customer that ordered the item. A CEP pattern for monitoring recent orders ready to be shipped could then be formulated as "detect a sequence of three events, o of type O (an item was ordered), s of type S (an item was fetched from the storage), and r of type R (an item was registered and is ready for delivery), such that the item ID of o, s, and r is the same and the time window W is equal to one hour". Every triplet of the events satisfying this formulation would contitute a complex event matching the above pattern.

Pattern matches are constructed incrementally by combining incoming events with already formed partial matches. For the pattern defined above, each newly arrived event of type O triggers the creation of a partial match of size 1 containing this event. Consequently, a new event of type S is matched with all partial matches containing previous O events, and the pairs sharing the same costumer ID form new partial matches of size 2. In a similar manner, each event of type R is compared against all partial matches containing the above pairs, and the triplets satisfying the conditions are united into full pattern matches which are then reported to the system users.

CEP systems commonly utilize a *detection model* for creating, extending, and managing partial matches during pattern detection. One model extensively used due to its simplicity and expressibility is the non-deterministic finite automata, also known as NFA ([2], [19], [31], [32]). An automaton allows to easily express the connections between the various subsets of a full pattern match according to the underlying structure of the pattern. Fig. 2 provides an example of an automaton constructed for the pattern from Example 1. Each state represents a particular step during the pattern matching process, with every traversed edge causing a new partial match creation. The transition from the initial state is performed upon an arrival of an event of type O, creating a new partial match. Likewise, an outgoing transition of state S will be traversed when an event of type S arrives satisfying the mutual condition with some previously obtained event of type O. The traversal and the creation of a new partial match with an active state R will take place for each valid (O, S) pair. Finally, the outgoing transition of state R will be traversed by each (O, S, R) triplet representing a full pattern match.

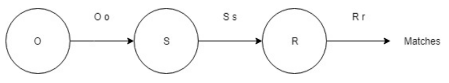


Fig. 2: NFA plan of P1

It can be observed that the computational cost of complex event detection as described above is inherently exponential in the size and the complexity of the pattern. For example, assume that the CEP system from Fig. 2 creates 100 partial matches $o_{i_1}, o_{i_2}, ..., o_{i_{100}}$ from 100 events past events of type O. An arriving event s_j will be evaluated to see if it can be used to extend those stored partial matches. Assuming that the conditions hold for every evaluation(or that there are none), 100 partial matches will be created - $o_{i_1}s_j, o_{i_2}s_j, ..., o_{i_{100}}s_j$. Similarly, 100 new events of type S can create 10,000 partial matches, each of which will have to be evaluated upon each arrival of an event of type R. This problem becomes even more severe for longer patterns.

To overcome the above problem, real-life CEP engines employ advanced strategies for optimizing system performance. Parallelizing the event detection process, an increasingly popular choice for a CEP optimization method, is the focus of this paper and is discussed in detail below.

III. Hybrid Parallelization Approach

In this section, we present our hybrid parallelization approach. We discuss the internal design of the system and its two-layered parallelism approach. Our system comprises execution units called agents. The outer layer splits the stream of events up between a group of agents such that each agent receives an incoming stream that contains one type of primitive events that appears in the pattern being detected. Each agent receives two types of input: these incoming events and partial matches that were found by other agents. The flow of these partial matches from agent to agent is determined by the detection model used.

An agent receives an event stream from the system's sources and a partial match stream from other agents as inputs. The agent combines these two inputs to extend the partial matches received from the agents in the outer layer. This combination of inputs is done by workers. Each agent comprises workers that examine the incoming events and partial matches and check whether new partial matches can be made based on its event stream and partial matches stream received from its neighboring agent. This internal execution done by the agent's workers is the inner layer of the hybrid-parallel approach. This layer can be viewed as an independent parallel tier, similar to the data-parallel approach. The layer's parallelism is manifested in the agent's internal design. It comprises several threads that run concurrently and merges of the two inputs.

Our hybrid-parallel approach combines the strengths of both state-parallel and data-parallel methods. It provides an unbounded degree of parallelism, since any number of cores can be allocated to an agent. Furthermore, events need not be replicated. An individual event is only handled by a single agent and processed by a single core of that agent. The event is only passed on to another agent once it has been combined with other events to form a partial match. Our method provides the flexibility to choose among distributed and multi-core architectures. For the outer layer, passing partial matches is the only communication between cores, thus agents could be easily deployed on both shared memory and share-nothing architectures. The inner layer is more suited to shared memory scenarios, as cores have to constantly pass along information needed for the pattern detection process. However, this inner layer can also be deployed on a distributed system if needed. While the system's design supports any detection model, we present our solution using NFA in the following subsections.

A. Agent Architecture

Events are received from outside the system and forwarded for handling as input to our system.[All basic events coming into the system are passed through a splitter. The splitter produces multiple streams, where each stream contains only events of a specific type. This stream is forwarded as input to an agent specializing in this type of event. All basic events coming into the system are passed through a splitter. The splitter produces multiple streams, where each stream contains only events of a specific type. Each type of stream is then forwarded as input to an agent that specializes in this type of event.

As explained, each agent receives two input streams. One input stream contains events of a specific type, and is received from the system. The second input stream contains partial matches; these are received from different agents that constructed the partial matches. The agent's output stream will contain the partial matches it managed to extend based on its input streams. This stream of extended partial matches is sent on to the next agent. The job of the next agent is to further extend the matches by combining them with the events it receives from the system.

Detecting matches requires that the agent stores the partial matches it received as input. This allows the partial matches to be examined to determine whether they can be combined with the incoming events to extend the match. Consequently, each agent maintains two complementary lists: a match buffer (MB) that stores the partial matches and an input buffer (IB) that stores the incoming events. When an item arrives from any of the two types of input streams, it is stored in its corresponding buffer. Then, the agent's workers evaluate each item with every item stored in the complementary buffer to discover whether it can create a more complete partial match. In short, an incoming event is checked against every partial match currently stored in the match buffer.

Execution threads are allocated among the different agents in the system. This allocation is decided by applying a cost model calculation that factors in the load expected for each agent. We describe it in detail in Section IV-A. An agent divides its threads into two groups: event workers and match workers. Both kinds of workers process the input streams, but each differs based on the input it handles. Event workers first receive events and put them in the input buffer (IB). Then, they check whether any of the partial matches stored in the complementary match buffer (MB) can be used to extend the match. Similarly, match workers receive partial matches and put them in the match buffer (MB). Then, they check whether any of the events stored in the input buffer (IB) can be used to extend the match.

Executing threads in this manner concurrently requires carefully managed access to shared data structures. Workers storing items in the buffers raise the problem of simultaneous access from multiple threads. A worker stores items in its group's buffer, and also has to iterate over its complementary buffer during the pattern detection process. Thus, a worker has to synchronize buffer access with other workers in its group and with workers in the complementary group. A simple solution would use coarse-grained synchronization, such as acquiring a lock for every buffer modification; however, this would cause a major overhead and degrade the system's throughput. Instead, our system has a sub-buffer for every worker; thus, different workers can iterate over different sub-buffers simultaneously. While synchronization is still required, locking now involves only two workers instead of every worker in an agent's group. Each worker maintains its buffer by adding items received from the input stream and by removing items as necessary. Detailed explanation on the removal method is discussed later in this section.

All workers of the same group in a specific agent, whether event workers or match workers, access the same input stream. The items are divided on a "first come, first served" basis. A worker either waits until an item arrives in the stream or accepts it immediately if there are already items that came in. While this method does not enforce load balancing on the size of the sub-buffers, it does not degrade the system's performance. A sub-buffer's size is not relevant since the buffer size does not affect insertion or removal time. The complementary workers have to iterate over all the stored items without differentiating where they are stored. From now on, our use of the terms IB and MB refers to the union of all sub-buffers of an agent's event workers and match workers, respectively.

```
1 while event stream has events:
2     e ← get_event_from_stream()
3     insert_to_own_input_buffer(e)
4     foreach match_worker in
        complementary_worker_list:
5         partial_matches_sub_buffer
            ←get_worker_sub_buffer(match_worker)
6         foreach m in partial_matches_sub_buffer:
7             if check_predicates(m, e):
8                 send_to_succeeding_agents(append(m,
                    e))
9     remove_old_events_from_buffer(partial_matches)
```
Algorithm 1: Event worker algorithm

Algorithm 1 showing the event worker algorithm contains a main loop that runs as long as events continue arriving as input. In lines 2-3 an event is received from the stream and is added to the worker's IB. Adding elements to the buffer requires that it be locked to prevent concurrent access by other workers who need to access the buffer, as seen in line 5. The worker then starts iterations over the match workers; in each iteration it gets the worker's sub-buffer. The sub-buffer can be accessed directly in a multi-core shared-memory architecture, but the event worker has to acquire a lock while going through the sub-buffer. In lines 6-7, the event is evaluated with every partial match from the sub-buffer, by checking the relevant predicates in the newly arrived event. If the condition check is passed, the event is appended to the partial match and is sent to the succeeding agents, as in line 8. Finally, some events are removed from the worker's IB in line 9. Events that should be removed are determined by the partial matches currently in the MB.

The match worker algorithm is similar to the event worker algorithm; the differences stem from handling partial matches instead of events. Instead of receiving events, the match worker gets partial matches as input. It then evaluates the partial match with the events from the input buffer to check for a more complete partial match. Finally, removing partial matches is determined based on the events in the IB, as explained below.

Because both worker groups perform similar actions, the question of creating duplicate partial match arises. Duplicates are avoided as there is no single combination of an event and a partial match that is evaluated twice. An arriving item can be evaluated only with items that are already in the complementary buffer. This means that a specific item is used in the check predicate function (line 7) only in two scenarios: (i) Immediately after arriving from the input stream, thus checked only with complementary items that arrived earlier. (ii) When stored in a buffer and a newly arrived complementary item triggers the check. The same

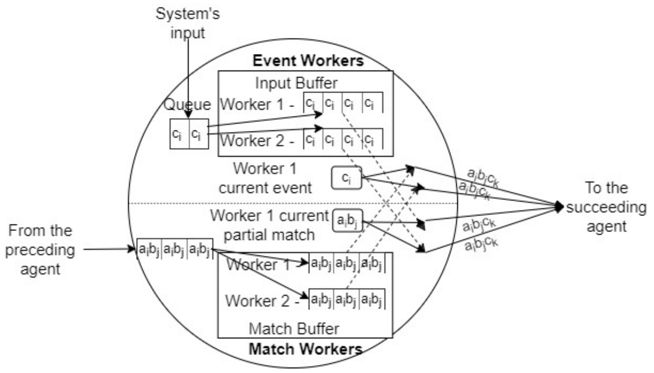


Fig. 3: Type C agent for detecting SEQ(A,B,C,D)

complementary item cannot exist in both scenarios and thus we do not check the same combination twice.

Items are removed from a buffer to ensure correctness of the algorithm, and prevent unnecessary evaluations and wasted memory space. Removal is done when an item can no longer form partial matches. An item can be discarded if it has arrived early enough such that a combination of new arriving items does not satisfy the pattern's window requirement, denoted as W.

Removing an event requires finding the latest timestamp of all the partial matches in the MB. The timestamp of a partial match is defined as the earliest timestamp among the events it comprises. Say t_{latest} is the latest timestamp of all the partial matches in the MB. Events are removed if they have a timestamp t for which $t + W < t_{latest}$ is true. Correctness is maintained because any arriving partial match is guaranteed to have at least one event with a timestamp that is at least as late as t_{latest}; otherwise it would have been created earlier and thus received earlier at the match worker. We cannot combine newly arriving partial matches with the removed events without violating time window requirements, so they can be safely removed. To find t_{latest} we need to iterate over the agent's MB and check all partial matches in it. This iteration requires another synchronization action with each match worker. To avoid those synchronization actions, we find t_{latest} while iterating the MB during the match detection algorithm.

Removing partial matches from the MB is done in a similar way, and is based on the events currently in the IB. All partial matches whose timestamp is earlier than the current time by at least W are removed. Each match worker derives the current time from the timestamp of the latest event in the IB. The partial matches can be safely removed if the events arriving from that point on will have a later timestamp than previously arrived events. Thus they could not be used to create matches with the partial matches that were removed.

Fig. 3 depicts an overview of an agent's components. This agent is part of a system for detecting the pattern's structure SEQ(A,B,C,D). The agent for event type C receives an event stream of type C events and a stream of partial matches of the form a_ib_j. Depending on whether it is an event or a partial match, an arriving item is added to a concurrent queue that supports simultaneous access using locks. Each of the two event workers accesses the event queue, takes an event, and adds it to its own sub-buffer. This event is evaluated with partial matches of the form $a_i b_j$ from the agent's match buffer, which is composed of two sub-buffers from the two match workers. Newly created partial matches of the form $a_i b_j c_k$ are then sent to the succeeding agent. The two match workers perform similar work. They access the partial match queue for partial matches, add them to a sub-buffer, and compare them with the events from the two input sub-buffers.

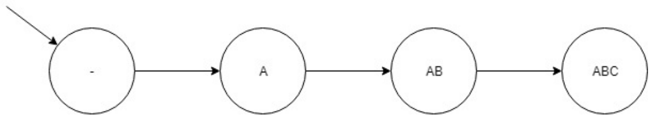
Some operators of the pattern's structure influence the input and output streams of the agent. Specifically, we will discuss how the Kleene closure operator affects agents. An agent altered by a Kleene closure sends partial matches as output to other agents and to itself. This is easily done by duplicating the stream and adding the partial match to the agent's queue of partial matches. At the technical level, the input queues support multiple threads, so there is no change in the data structure. Additional changes to the algorithm are also unnecessary, since the operators affect only the input and output of the agents.

B. System Design

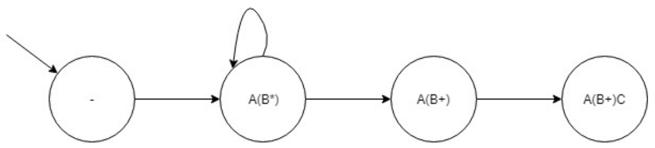
This section describes the overall structure of the system and inter-agent communication. First, an evaluation mechanism is created based on the structure of the pattern and the topology of the detection model. For our example, an NFA determines the topology of the algorithm and hence the input and output for the agents. A state in the NFA corresponds to an agent and represents a subset of a pattern's structure, since partial matches of a specific subset are created at a specific state. The initial state represents the empty subset. Edges between states of the NFA are built according to the operators of the pattern's structure. Partial matches flow from one agent to other agents that are connected to it by edges.

As an example of building an NFA based on a structure's operators, the sequence operator creates an NFA where each state has exactly one outgoing edge, leading to a different state or the system's output. Kleene closure on a specific category of the sequence alters the corresponding state by adding an output edge that leads to itself (self-loop). Our system also supports all common CEP operators such as conjunction, disjunction, and negation; these operators influence edge building, but are outside the scope of this discussion for brevity. Our system also supports other alterations of the evaluation mechanism such as pattern reordering [19]. These alternations can be provided with a pre-made detection model so the partial match is built in a different order. We define an agent as preceding another agent if their corresponding states immediately precede each other in the NFA. Similarly, succeeding agents immediate follow each other in the NFA.

Fig. 4 presents an example of the NFA for our example's structure. For the pattern's structure of SEQ(A,B,C,D) the resulting NFA is shown in Fig. 4a. Its states are ordered by the sequence imposed. The effect of the Kleene closure on the resulting NFA is presented in Fig. 4b. In the structure



(a) NFA created for detecting SEQ(A,B,C,D)



(b) NFA created for detecting SEQ(A,B+,C,D)

Fig. 4: NFA created for different structures

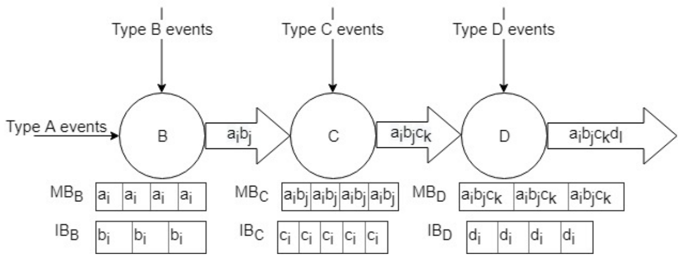


Fig. 5: System for detecting SEQ(A,B,C,D)

SEQ(A,B+,C,D), type B events can exist more than once in the match so the state representing agent B has an added self-loop.

Incoming events are split according to their type and are sent to specific agents. An agent receives all the events that correspond to its event type, and only those events. Partial matches are received from agents that correspond to preceding states. As explained in the previous subsection, an agent combines the input streams it receives and creates more complex partial matches as output. These partial matches are sent to succeeding agents as the output stream of that agent. To illustrate, we look at Fig. 5, which shows the actual parallel system used for detecting the sequence in Fig. 4a. Taking agent C as an example, it receives events of type C and partial matches containing events of types A and B. The agent uses the pattern matching algorithm and sends newly created partial matches of the form $a_i b_j c_k$ to the agent that correspond to state D.

The initial state does not have preceding states, thus it cannot receive partial matches and will not create any output. This state sends its events as partial matches of size 1, directly to their succeeding states without any evaluation. As an optimization, our system removes the initial state and forwards incoming events of the relevant type to the partial match stream of the succeeding states. The optimization can be observed in Fig. 5. Type A events arrive directly to the partial match stream of the type B agent and are stored in the match buffer. The buffers' contents for every agent are also presented in the figure.

IV. Theoretical Analysis

This section provides a theoretical analysis of our system. First, we discuss the optimal number of threads that should be allocated to each agent and how these threads are divided internally. The allocation is based on the expected load of each agent, which is in turn affected by the input parameters of the system. These parameters include the arrival rate of incoming events, the window size, and the selectivity of the pattern's condition, which is a fraction denoting the number of successful evaluations divided by the total number of evaluations. The second theoretical analysis is a detailed breakdown of the system's main metrics: throughput,latency, and memory usage.

A. Thread Allocation Cost Model

Our system's design relies on the fact that the agents are all running in parallel. Each agent i is statically allocated a specific number of threads T_i. This number is determined by the total number of cores available in the system T and by the load expected for each agent. In a specific agent i, the total number of threads T_i is further divided between event workers and match workers. T_{IB_i}, T_{MB_i} represent the number of threads in each group, respectively. When allocating threads to agents, our goal is to efficiently balance the load among them. Our goal is to have the same load on each thread so there will be no threads sitting idle while others are causing a bottleneck. Throughout this discussion, we assume the threads are homogeneous and perform calculations at the same speed.

The first allocation we discuss is the inner-agent allocation. Given a specific agent i and a number of threads T_i we now explain how we allocate the threads among the agent's groups. We start calculating the optimal allocation by comparing the rate of computational and synchronization actions each group of event workers and match workers performs. First, we define the rate of items arriving at that agent. Since the arrival rate can vary due to data bursts and congestion on the agents, we define the average arrival rate of incoming events e_i and the average rate of incoming partial matches m_i. These rates, together with the window size W, determine the size required for an agent's buffers. When the system has been running for at least W time, we can say that items are expected to be discarded from the buffer at the same rate at which they enter it. Thus, an item exists in the buffer for a time frame of roughly W. Therefore $|IB_i| = e_iW$ and $|MB| = m_iW$ are the expected input buffer and match buffer sizes, respectively.

The expected buffer sizes assume that the window size is very large compared to the latency of the incoming partial matches. Partial matches arrive with some latency as they pass through the preceding agents compared to the events, which arrive directly. When a partial match arrives at the IB, the buffer most likely already holds events with a timestamp that is later than that of the incoming partial match. The delta of these two timestamps is denoted d. Thus, an average partial match stays in the match buffer for $W - d$ time and an event stays in the input buffer for $W + d$ time. Having $W >> d$ is a realistic assumption as our system is mainly intended to work with patterns that have large time windows because they require more calculations. These patterns that require the largest number of calculations are those that benefit the most from parallel execution.

These calculations involve the computation performed to match the incoming events with the incoming partial matches, and are the computational actions we consider for our cost model. A worker executes evaluations when it receives an incoming item and tries to combine it with items in the agent's complementary buffer. All event workers in agent i cumulatively perform evaluations at a rate of $l_{event_i} = e_i \cdot |MB_i|$, since every incoming event is evaluated with every partial match in the input buffer. Similarly, match workers cumulatively perform evaluations at a rate of $l_{match_i} = m_i \cdot |IB_i|$. When simplifying the expressions, we observe that $l_{event_i} = l_{match_i} = e_i m_i W$, which indicates that the two groups perform an equal amount of evaluations.

All evaluations done by the same agent do not require the same number of computational actions, due to the difference in payloads of the participating incoming events and the partial matches. Therefore, in our discussion we use the average number of computations done for each evaluation. While evaluations vary in their complexity, both groups evaluate the same predicates and are thus considered to have the same average complexity. With this observation, we infer that the number of computational actions performed in both worker groups of the same agent is the same.

The second factor that affects a worker group's load is synchronization. Often in CEP systems, the pattern's condition is very simple and easy to compute and requires a small amount of computational actions. Thus, while there are relatively few synchronization actions compared to computational actions, it is important to include those synchronization actions in our cost model.

Synchronization between workers must to be done in two cases. The first case occurs when the worker is iterating over a complementary sub-buffer and locking it. T_{MB_i} and T_{IB_i} are the number of synchronization actions required for event workers and match workers, respectively. The second occurs when a worker sends a partial match to its succeeding agents using the concurrent queue between agents. In short, there is synchronization among the workers in an agent, and synchronization when partial matches are sent among agents. Since we cannot use these values before allocating the threads, we make the assumption that the threads are split evenly between the groups and so $T_{IB_i} = T_{MB_i} = T_i/2$. The second form of synchronization is when a worker sends a partial match to its succeeding agents using the concurrent queue between agents. The rate of partial matches sent depends on the selectivity of the pattern's condition. All evaluations in a specific agent have the same selectivity, regardless of the worker that performed them. Since the rate of evaluations and the condition's selectivity is the same, the rate of outgoing partial matches and thus the rate of synchronization actions is the same. Consequently, the expected load on both groups is the same and as a general rule, threads should be split evenly between the groups.

However, in specific scenarios e_i or m_i can be so low that there would be many more threads in that group than what is needed to handle those few events or partial matches. Thus, using half the threads allocated to that agent would be wasteful and would lead to idle threads. Taking input workers as an example, we say that a worker can complete p evaluations in the interval between two event arrivals. Therefore, the number of threads used for event workers must not exceed m_iW/p. If that limit is below half the threads allocated for the agent, the excess threads will be used as match workers. This observation also holds for the match workers with a limit of e_iW/p.

Thread allocation among agents requires knowledge of the load on each agent. As discussed above, for a state i an agent has a rate of $2e_im_iW$ evaluations. When comparing the load on all agents, we have to consider that different agents have different evaluations that vary in their complexity. We define c_i as the average cost of one evaluation in agent i, measured in computational actions. As mentioned above, evaluations by the same agent can also vary in complexity. However, during the system execution this cost is amortized over many instances of computations and we can use the average in our cost model. c_i can be obtained by checking the predicates on a set of events and partial matches, and measuring the computational actions performed. This set can be collected from a previous run or by simulating some events and partial matches. Consequently, the number of computational actions in agent i is

$comp_i = c_i(l_{event_i} + l_{match_i}) = 2e_im_ic_iW$.

Workers of agent i cumulatively performs $(e_i + m_i)T_i/2$ synchronization actions when they lock complementary sub-buffers. This value requires the actual thread allocation to calculate the precise load. As we cannot calculate the load before the allocation, we assume an even allocation among agents such that $T_i = T/n$. We define b_i as the cost of locking a sub-buffer while iterating it. The synchronization load also includes sending partial matches using the concurrent queue at a rate of m_{i+1} at a load cost of q_i. Therefore, the total synchronization load of an agent is
$sync_i = b_i(e_i + m_i)T_i/2 + q_im_{i+1}$.
Thus, the total load on an agent is
$load_i = comp_i + sync_i$.
Thread allocation is performed using this formula (n is the number of agents in the system):
$T_i = \frac{load_i}{\sum\limits_{j=1}^{n} load_j}$

While most symbols in this formula are input parameters and therefore known, m_i has to be expressed by those parameters. The rate of incoming partial matches depends on the NFA's structure. Therefore, we limit our calculation to specific structures due to space constraints. First, we discuss NFAs where each state has exactly one direct preceding state, except for the initial state. These are NFAs that are built when the pattern's structure is a sequence without an additional operator, for example the, NFA presented in Fig. 4a. The agents are numbered by the order of their corresponding states in the chain. The initial state does not correspond to an agent because our optimization forwards its incoming events directly to the agent that would succeed it. Thus, Agent 2 is actually the "first" agent, with no preceding agent.

m_i is calculated as follows

$$m_i = \begin{cases} e_1, & i = 2 \\ |MB_{i-1}| * e_{i-1}s_{i-1} = m_{i-1}e_{i-1}s_{i-1}W & i > 2 \end{cases}$$

Agent 2 receives its partial matches stream directly from the system. These will partial matches of size 1, which contains only events of the first category and their arrival rate is defined as e_1. For all other agents, partial matches are received as a product of evaluations done by the preceding agent on events and partial matches. Every incoming event has the potential for creating $|MB_{i-1}|$ partial matches. However, as not every comparison produces a partial match, the number of partial matches created is lower than the number of partial matches in the match buffer. This depends on the product of the selectivity of all the conditions required to pass for the creation of a new partial match. We denote this selectivity as s_i and assume it stays constant. Thus, only $|MB_{i-1}|s_{i-1}$ partial matches are actually created for every incoming event.

An arriving partial match can also be used to create partial matches when it is compared against events in the input buffer. However, to simplify the calculation, we assume that only arriving events can create new partial matches. As mentioned in Section III, every combination of event and partial match is evaluated exactly once. Since the selectivity is also the same regardless of the worker performing the evaluation, our assumption is valid. In both cases, the rate of partial matches created is the same.

After simplifying the formula, m_i can also be represented by a non-recursive formula:
$m_i = e_1W^{i-2}\prod\limits_{j=2}^{i-1}(e_js_j)$.
This value is used to calculate T_i using only the input parameters.

Kleene closure is the second pattern type we discuss. Its NFA differs from the sequential pattern by having states with self-loops. Agents corresponding to states affected by Kleene closure, such as Agent B in Fig. 4b have a different m_i than the one calculated above. This is because any partial matches created by an agent with Kleene closure are also forwarded back to that same agent. The rate of partial matches arriving at an agent i with Kleene closure can be viewed as two different factors. m_i^{prev} is the rate of partial matches arriving directly from the preceding state and is calculated in the exact same way as m_i is calculated in a sequence pattern. $m_i^{KC_j}$ is the rate of partial matches arriving from the self-loop that has j events of the type of agent i, such that
$m_i = m_i^{prev} + \sum_{j=1}^{\infty} m_i^{KC_j}$.
We observe that, initially, partial matches are constructed from the input arriving from the preceding state at a rate of
$m_i^{KC_1} = m_i^{prev} * |IB_i| * s_i$.
Every partial match is evaluated with every event in the input buffer with a selectivity of s_i. Then, these partial matches arrive on the self-loop and create partial matches at a rate of

$m_i^{KC_2} = m_i^{KC_1} * |IB_i| * s_i$,
which are again forwarded to agent i. And so it goes on indefinitely; therefore
$m_i^{KC_j} = m_i^{prev}\left(e_i^k s_i^k W^k\right)$.

Consequently, the rate of partial matches for a Kleene closure agent is

$$m_i = m_i^{prev}\left(1 + \sum_{k=1}^{\infty}\left(e_i^k s_i^k W^k\right)\right).$$

If all preceding agents are not affected by Kleene closure, then the non-recursive form is

$$m_i = e_1 W^{i-2} \prod_{j=2}^{i-1}(e_j s_j)\left(1 + \sum_{k=1}^{\infty}\left(e_i^k s_i^k W^k\right)\right).$$

We can use it in the load formula and then calculate T_i, just as we did for the sequence pattern.

B. Complexity Analysis

To complete our presentation of the pattern detection system, we analyze its complexity regarding the computational and synchronization actions, needed along with memory consumption. The rate of computational actions for individual agents is calculated in the previous subsection, and the total amount in the entire system is $\sum_{j=1}^{n} comp_i$. As expected, a larger time window, faster arrival rate of events, and longer pattern sequence all contribute to more calculations and therefore increase this rate.

The rate of synchronization actions was also calculated in the previous subsection, where we assumed an even allocation of threads among agents. Since we now have the actual allocation, we can properly calculate the rate. Thus, instead of using $T_i = T/n$, we use the actual values $T_{IB_i}, T_{MB_i}$and the rate of synchronization actions is $b_i(e_i{+}m_i)T_i/2{+}q_i m_{i+1}$. This rate is most affected by the time window, because a larger window leads to more evaluations and therefore more synchronization actions. Also, as expected, more threads in the system requires more synchronization.

Memory consumption is defined as the size of all the event buffers and match buffers in the system. We define v_i as the average event size that is handled by agent i. Therefore, the size of agent's i input buffer is $|IB_i| * v_i = e_i v_i W$. A partial match from the match buffer of agent i is composed of single events, and each preceding agent added a single event. Thus its size is $\sum_{j=1}^{i-1} v_j$ and the total match buffer's size for agent i is $\sum_{j=1}^{i-1} v_j * |MB_i|$. Combining both buffer sizes and summing over all agents, we show that the total memory consumption is $\sum_{i=1}^{n}\left(e_i v_i W + \sum_{j=1}^{i-1}(v_j m_j W)\right)$.

V. Implementation Details

There are a few implementation details that contribute to the optimization of the system but also impose some restrictions on using it. Knowledge of these details can help explain some design choices we have taken.

Scoping parameters [19] is a critical optimization implemented in our system. We observe that an arriving partial match does not necessarily have to be checked with every event in the input buffer. Due to temporal constraints, an event that does not have its timestamp within the interval imposed by the partial match cannot be used to extend the match. While this optimization was originally intended to be used with pattern reordering, it is useful for our algorithm as well. When a partial match arrives, there may be events with a later timestamp that have already arrived. Only those events should be evaluated with the arriving timestamp and not the entire IB.

The input sub-buffers are sorted based on the incoming events' timestamps, since each event worker stores events in the order in which they are received. This makes pruning the events outside the useful timestamp scope a constant-time action to eliminate the need for a full scan of the input buffer. This is not the case when dealing with the match buffer, as it is not sorted. However, we can still check the timestamps before checking the actual predicate. This is helpful in cases of conditions that have more significant computation times.

Concerning the parallel process, we observe that while our system does not have an upper bound on its parallelism degree, it warrants a minimal number of threads to operate. Each agent requires at least an event worker thread and a match worker thread. The total number of agents is the same as the number of categories in the pattern, subtracted by one due to our optimization. Thus the minimal amount of threads should be twice that number. For example, the system would need at least six threads to process the patterns in Fig. 5. To maximize throughput, the system should be used with an appropriate processor that supports parallel execution of that many threads. Unfortunately, because an agent can have a relatively low amount of computations to perform but still requires two threads to operate, this can lead to idle threads and a waste of possible activity. One solution would be to use fusion [17], a known parallel computing technique in which two or more parallel units are merged into a single one. We discuss how fusion can be implemented by merging agents in Section VI-B

Restricting a thread to function only as an event worker or only as match worker can cause some threads to be idle for a long period of the execution time. While the theoretic cost model shows that the thread allocation inside an agent splits the threads evenly between event workers and match workers, in practice this does not always lead to the best throughput. The actual event rate e_i can change over time and so can the match arrival rate m_i. The execution time of every thread also varies and depends on the system's load, memory caching, and the complexity of the specific condition being computed.

Even minor variances in the throughput of a group of threads can cause that group to perform significantly better or worse than another group. For instance, if at a certain point in the execution, the event workers of a specific agent handle the incoming events at a slightly faster rate, the IB for that agent will also grow faster. However, a larger IB means that every arriving partial match is going to be evaluated with more events. This will take thus taking more time and slow the rate at which partial matches are handled, which also leads to slower growth of the MB. As the MB is now smaller than it would have been without the variance of computation, the event workers can handle events even faster due to fewer evaluations needed. Ultimately, we will see a "rolling effect" in which the IB grows even faster as the event workers need to perform fewer evaluations. Meanwhile, the match workers have to perform most of the evaluation and the MB grows even slower. The event workers will finish their task much faster and thus become idle while the match workers continue to run. The same phenomenon can occur if the match workers start to perform slightly better and the MB is the one that grows faster instead of the IB.

As a practical solution to our experiments, we added support for threads to operate on both groups. In this "state-dynamic" allocation, each thread maintains two workers - an event worker and a match worker with one of them being the primary worker and the other the secondary worker. The threads are still split evenly between the groups such that half the threads have an event worker as their primary worker, while the rest have a match worker as their primary worker. Handling an item is now split to an input phase and an evaluating phase. In the input phase, the thread receives an element from one of the possible streams. Each thread has a primary stream and a secondary stream depending on its primary worker. A thread with a primary event worker will have an event stream as its primary stream. The thread then tries to receive an element from its primary stream, if it cannot do so because there is no such an element available, it checks the secondary stream. Based on the element received, the thread starts the evaluating phase in which the corresponding worker is used to handle the element.

Supporting this dynamic allocation requires twice the amount of sub-buffers, as there are twice the amount of workers. When the worker is handling an item, it has to get the complete complementary buffer (IB or MB) from the other group of workers, which now includes all the threads of the agent. This, in turn, leads to more synchronization actions. Although the extra synchronization affects the performance negatively, this change to support dynamic allocation ensures that there will not be idle threads in the agent – which outweighs the negative impact. We further improve this allocation by having a thread handle elements of different agents, as detailed in Section VI-A.

VI. Extensions

A. Dynamic Thread Allocation

Agents can become idle for various periods of time during the system's execution. While the cost model aims to minimize this problem, it can still happen due to imperfect application of the continuous cost model on a discrete number of threads. As explained in Section V, changes in the input parameters also affect the cost model. When an agent is idle, its threads are no longer contributing to the throughput of the system and using them for another agent's computations would improve the system's performance. As a further optimization to the state-dynamic allocation explained in Section V, we propose a fully dynamic thread allocation.

Similar to the state-dynamic allocation, each thread maintains both an event worker and a match worker, but it does so for every agent. Each thread is also accessible to all input streams of the system. A thread is associated with an agent according to the cost model thread allocation, denoted as the primary agent. That thread is further assigned a primary worker and a secondary worker of that specific agent, similar to the procedure in the state-dynamic allocation. The thread tries to receive an input from its primary and secondary streams of its assigned agent. If there is no element in both streams, the agent is considered idle and that thread can perform work for other agents. It randomly tries inputs associated with other agents until it receives an element and handles it using the corresponding worker. This input is saved and will be tried first instead of randomly choosing an input.

This optimization ensures that threads will contribute to the system even if their assigned agents are idle. However, it can also impact the performance negatively. Every thread now holds a worker of a specific group for every agent. Moreover, a worker has to get the sub-buffer from every thread in the system instead of just the thread of its agent. This adds numerous synchronization actions, which could outweigh the benefits from the additional computing power provided by now non-idle threads. Thus, the fully dynamic allocation is better suited to environments with a low amount of threads, so the additional synchronization cost will not be significant. Also, such environments have a higher chance of thread allocation that differs significantly from the theoretic cost model allocation.

B. Fusion

As explained in Section V, the system's design requires two threads for every agent. If an agent is expected to have a significantly smaller load than other agents in the system, the cost model will allocate less than two threads for it. However, due to the minimum thread limit, these agents will actually be allocated two threads. These threads must be taken from other agents that could have actually used them. For patterns that have many low-load (and thus wasteful) agents, losing throughput is a critical issue.

We introduce fusion [17] to our hybrid algorithm as a viable solution. Two consecutive agents, A and its successor B, can be fused together to create a single fused agent that handles both functions. Similar to a non-fused agent, the fused agent has two input streams. Its event stream is the combination of the two event streams of the original agents, while its partial match input stream is the same partial match input as the original agent A. The fused agent uses the same output as agent original B. The fused agent maintains two IBs and two MBs, which contain the same elements as those of the original agents. On arrival of some event a of agent A, it is evaluated with partial matches received from the preceding agent. Newly created partial matches are added to the second MB, denoted MB$_B$. Some event b of agent B that arrives is compared with the partial matches in MB_B. Partial matches created at that stage are forwarded to the succeeding agent. A fused agent requires just two threads to operate and thus minimizes the total system's requirement without performing unnecessary additional computations. This optimization ensures that the actual allocation is more similar to the allocation calculated in the cost model. Fusion is especially suited for patterns resulting in many agents with such small loads that they are allocated less than two threads. Furthermore, it is also highly beneficial in scenarios with a low number of available cores, where the additional threads gained from fusion is more significant.

VII. Experimental Evaluation

In this section, we present the results of the experimental evaluation of our hybrid parallelism method. Our main objective in this empirical study was to assess the overall system performance achieved by our approach using different metrics and to measure its improvement over a single-core sequential CEP algorithm. Furthermore, we evaluated the performance of the state-of-the-art parallel CEP system and compared it with our own. Our second objective was to study how the extensions of our system (as presented in Section VI) affect the overall performance and analyze those results.

A. Experimental Setup

We implemented the parallel CEP system described in Section III, which we refer to as the "hybrid approach", together with the cost model presented in Section IV-A. We also added the extensions described in Section VI, with the option to enable or disable them as needed. We also implemented a sequential CEP engine and a parallel state-of-the-art CEP engine, RIP [5] to measure our system's performance on a relative scale. We pre-calculated the rates of the event arrival frequencies and predicate selectivities, and fed them into the thread allocation mechanism. Unless stated otherwise, we used the hybrid approach with a thread allocation determined by the cost model and with none of the extensions from Section VI.

We used two independent real-world datasets in the experiments. The first was taken from the NASDAQ stock market historical records [1]. Each record represents a single update to the price of a stock. The data we used spans a one month period covering over 2100 stock identifiers with prices updated periodically. Our input stream contained 6,239,997 primitive events, each comprising a stock identifier, a timestamp, and a current price. To support the detection of some patterns described below, the event was augmented with an additional 20 values. Each value is a price taken from the last 20 prices of that specific stock, ordered in chronological order. We considered an update of each stock identifier as an event belonging to a separate type. For example, updates to Google, Apple, and Twitter stocks represent 3 different types of events. The second dataset contains measurements from sensors placed in apartments to recognize human activities and was taken from the research of D.Cook [7]. Each sensor measurement comprises a timestamp, the activity performed by a resident of the apartment, and an additional 33 attributes describing the measurement itself. For example, these might represent the type of the sensor, the value it measured, the time since this sensor was last activated and so forth. The dataset contains 13,956,534 measurements, with each considered a separate event. We consider each activity to be a separate type of the event.

The patterns we used for experimenting on both datasets are similar in their structure. Each pattern represented a sequence of event types, and a Kleene closure operator applied to one type in the relevant experiments. The condition used for these patterns is a predicate between two consecutive event types. For the first dataset, wanted to define patterns that could monitor desirable changes in stock prices. Therefore, the condition used was either a change on the price from the last price as compared to the change of the other event, or a requirement on the correlation between two stocks by their 20 previous prices. For the second dataset's patterns, the conditions tested the different changes in activity levels as measured by the sensors, which is an event attribute. The rationale of these patterns was to help predict the next activity the person would perform. Unless stated otherwise, we ran the experiments with sequences of six event types and a changing time window.

The metrics we used to measure performance were (1) Throughput, defined as the number of events processed per second. (2) Latency, defined as the average of all time intervals between the moment a match is created and the latest event in that match. (3) Memory consumption, defined as the maximal amount of bytes used by the system during its execution on a specific pattern. All experiments were run on a machine with 24-cores, 2.20 GHz CPU, and 16.0 GB RAM; the parallel systems were given 24 cores unless stated otherwise. All models and algorithms were implemented in Java.

B. Experimental Results

In our first experiment, we evaluated the throughput gain of our hybrid-parallel system as compared to a sequential approach. We tested various values of time windows, available cores, and pattern length. For this experiment, we used both sequence patterns and Kleene closure patterns when testing the throughput as a function of the time window or the number of cores. When testing the effect of different pattern length on the throughput we used only sequence patterns. The Kleene closure operator was not used in this specific test, since it is similar in its function to extending the length of the pattern.

To calculate the throughput, we measured the total execution time for processing a fixed number of events. The results are displayed in Fig. 6, presented as the time speedup gained over the sequential run. In most scenarios, our system performs significantly better than the sequential algorithm, and we achieve a speedup gain from 0.5 up to three orders of magnitude, compared to the sequential run. We observe the speedup increase is relative to the increase in pattern complexity, as apparent both in larger time windows

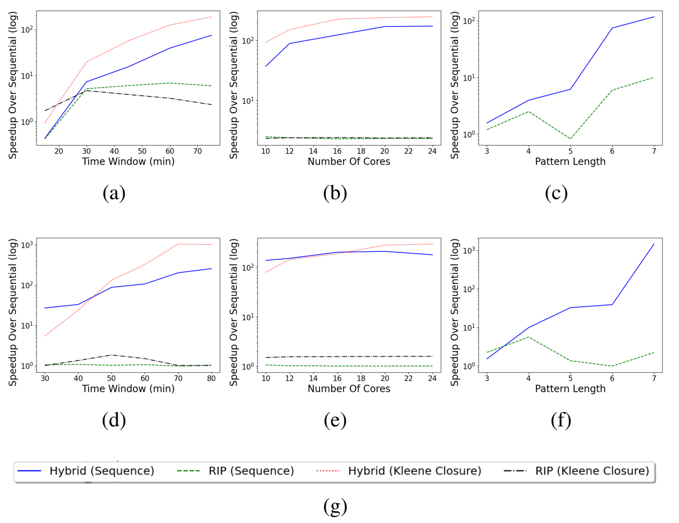


Fig. 6: Speedup (higher is better) relative to the sequential algorithm of the hybrid-parallel system and state-of-the-art RIP system, applied on the stock dataset ((a)-(c)) and on the sensor dataset ((d)-(f)), as a function of: (a),(d) time window; (b),(e) number of cores; (c),(f) pattern length.

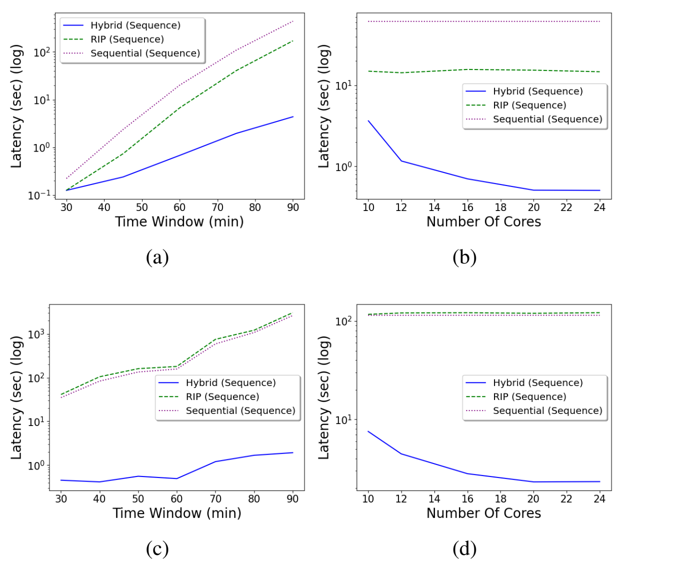


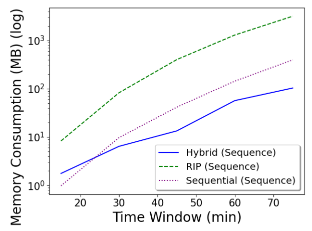
Fig. 7: Latency (lower is better) comparison of the hybrid-parallel system, state-of-the-art RIP system, and a sequential implementation applied on the stocks dataset (a),(b) and the sensors dataset (c),(d), as function of: time window (a),(c) and number of cores (b),(d)

and in larger pattern lengths. A pattern requires more computation actions to process it when it is more complex. We attribute this improvement in speedup to a higher ratio of computation actions compared to synchronizations actions. As presented in Section IV, handling an item and iterating over the complementary buffer require a fixed number of synchronization actions, but the number of computational actions needed depends on the size of the opposite buffer. Both the IB and MB increase in size with an increase in time window. Fig. 6b and 6e shows the scalability potential of the hybrid-parallel approach. We observe that the system scales well when adding more cores, with the exception of a sequence pattern over the sensor dataset which hits a limit at about 16 cores.

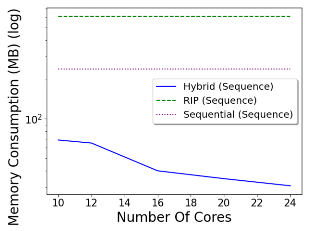
Next, we evaluated our system's throughput as compared to the state-of-the-art RIP system, which is based on the data-parallel approach. We tested the throughput RIP achieved on the same patterns used for testing the hybrid-parallel approach. RIP speedup compared to the sequential run is also displayed in Fig. 6. While our system produce good results with both datasets, the performance of RIP is significantly worse in the sensors dataset than in the stocks dataset. It can be explained by a unique characteristic of the sensors dataset; most partial matches, and especially longer partial matches, are composed of a small set of events. These events arrive relatively close to each other and so they are mapped to a single RIP thread. This thread have to perform a majority of the needed evaluations by itself, while all other threads finish their batch of threads early and wait idle for more events. This leads to a throughput which is very similar to the sequential, and it can be seen that the speedup RIP achieved is very close to 1 and sometime even lower, performing worse than the sequential run. This is a major disadvantage in data-parallel system that our hybrid-parallel approach does not suffer from. In both datasets, our system achieved a significantly higher throughput than RIP in all test cases, with improvement of up to two orders of magnitude in the stocks dataset and up to three orders of magnitude in the sensor dataset. We observe that while RIP improves in speedup when the pattern is more complex, the improvement gained by our system is considerably higher.

Parallel CEP usually algorithms suffer from increased latency due to an out-of-order match detection. While this is also the case in our solution, we detect matches with less latency compared to the state-of-the-art RIP system and compared to the sequential benchmark, as seen in Fig. 7. It shows the results of our latency testing as a function of the the window and the number of available cores with a sequence pattern. Kleene closure pattern was also tested and produced similar results but is omitted from the figure due to space constraints. This improvement in latency over the other tested methods is attributed to the increased throughput discussed above. It allows the hybrid algorithm to find matches faster and keep the overall latency low.

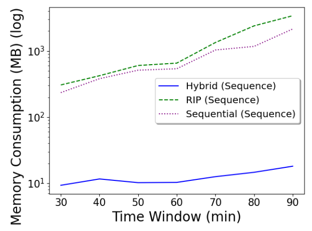
We compared the memory consumption of our system with the memory consumption of RIP and a sequential benchmark. The testing was performed on a changing values of time window and number of available cores, and a sequence pattern was used. Similar to the latency tests, we omitted the Kleene closure results due to space constraints, but they had produces similar results. In our system, we measured the highest memory consumption in every thread by itself and



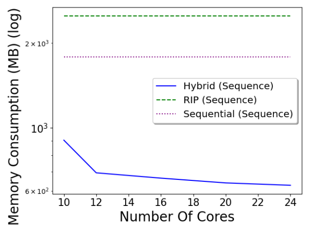
(a)



(b)

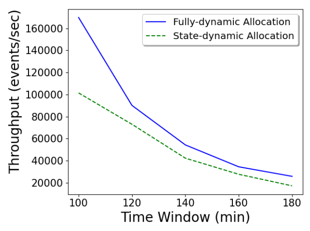


(c)

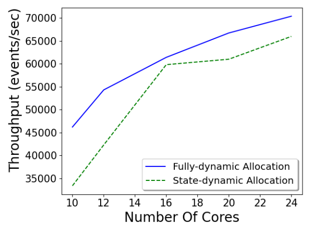


(d)

Fig. 8: Memory usage (lower is better) comparison of the hybrid-parallel system, state-of-the-art RIP system, and a sequential implementation applied on the stocks dataset (a),(b) and the sensors dataset (c),(d), as function of: time window (a),(c) and number of cores (b),(d)

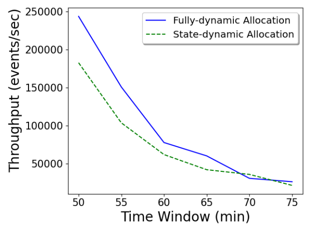


(a)

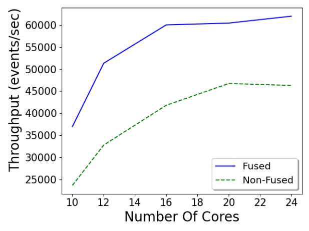


(b)

Fig. 9: Throughput (higher is better) comparison of the fully dynamic allocation extension as function of (a) the time window and (b) the number of cores, applied on the stock dataset.



(a)



(b)

Fig. 10: Throughput (higher is better) comparison of the fusion extension as function of (a) the time window and (b) the number of cores, applied on the stock dataset.

then summed the results, which is a worst-case measurement instead of an accurate one. It was done so to not add additional synchronization action between threads which would hurt performance. We used a similar method for RIP for the same reason. While it is expected that the parallel algorithms will have a significantly higher memory consumption due to events and partial matches that are handled simultaneously across different threads, we observe that our hybrid implementation actually uses less memory than the sequential algorithm.

Our system's low memory consumption can be explained by having a low amount of waiting partial matches. Memory consumption of a CEP system comprises mostly of partial matches waiting to be evaluated with incoming events. It is also the case for our implementation, however in many scenarios those partial matches are removed almost as soon as they are created. Looking at some agent in the system, we observe that at a single point of time, its handled events will have a later timestamp than the timestamp of the handled partial matches, if it is not the initial agent of the sequence. It is because partial matches have to pass through the preceding agents before they arrive at that agent, while the events simply arrive from the event stream. It means that when a partial match arrives it is possible that any event arriving after it will have a timestamp so late that it will not be evaluated with it because of the time window constraint. In such case the partial match is removed as soon as the worker finished handling it, which is not long after it was created in the preceding agent. Comparing that to the sequential algorithm where a partial match stays in memory until enough events have been handled explains why the sequential algorithm consume more memory.

The second goal of the our experimental evaluation was to test the effects of the proposed extensions on the system. We measured the throughput our system achieved with an enabled extension and compared it to the throughput of a run with that extension disabled. Fig. 9-10 summarize the results of the experiments for testing the two extensions discussed in Section VI. We used the stocks dataset, and measured the throughput both as a function of the pattern's time window, and as the total cores available to the system.

Fig. 9 displays the results of our fully-dynamic allocation test. We observe improvements in throughput over the state-dynamic allocation for every value of time window and every number of cores used. However, the figure shows that system configurations with low number of cores benefit the most from fully-dynamic allocation. We infer there are two reasons for this phenomenon. First, an idle thread degrades the performance of the system by a larger portion when the overall number of thread is lower. Because the fully-dynamic allocation utilize those idle threads, the gained throughput is more in those systems with fewer threads. The second reason is that the fully-dynamic allocation requires more synchronization actions than the fully-dynamic allocation, because there are more agents in the system. The added number of synchronization actions depends on the overall number of threads. Thus, this allocation incurs less overhead in environment with fewer threads.

In Fig. 10 we display the results of the fusion extension tests. For this test, we fused two agents out of the 5 required to detect a pattern of size 6. The results shows a throughput gain when using the extension over the "non-fused" version in most cases tested. The most notable improvement is when using just 10 threads, which is the minimal number needed for the non-fused version. The fusion extension allows more threads to be allocated and used by agents that can benefit them more. Thus, an extra thread has more relative impact when the overall number of threads is lower. This observation is also evident from Fig. 10b, where it is shown that relative throughput gain of fusion degrades with an increase in the number of cores.

VIII. Related Work

In recent years, CEP has been an active field of research [12]. Different systems were proposed to detect patterns in the input stream. Most use an NFA as a detection model [31], [2], [32], [19], and some are based on different detection models, such as trees[23], [19] or general graphs [3], [22]. While we presented our work with the commonly used NFA, our solution can apply to any detection model that uses separate building blocks for its operators. The agents we discuss can be deployed on any operator with their input and output streams split and merged as needed.

High-volume input streams raise the problem of processing incoming data in an efficient matter. Extensive work was done to address this issue using various optimizations. These solutions include pattern reordering [19], [23], [27], predicate optimization [31], [32], and memory management solutions to reduce memory usage [2]. While these works aim to optimize the pattern matching process, they are orthogonal to our solution as we use a parallel approach. Our work is complementary to these optimizations and can be used together with them.

Stream processing is a broad area of research that focuses on performing continuous queries over any stream of data and is therefore closely related to CEP research. In recent years, a parallel approach was used to improve the performance of stream processing systems. There are works that aim to improve the system's throughput, such as StreamIt [14], [29], which provides coarse-grained task parallelism, and SABER [20], which runs queries on heterogeneous hardware of both CPU and GPU cores. Other papers have suggested load balancing solutions [25], [24] and solutions for minimizing latency [4].

In addition to the above works, research on parallel implementations of automata also exists [28], [18]. Our work differs in the fact that we parallelized the pattern matching process and not the evaluation mechanism itself.

Parallel detection of complex events is another important optimization of CEP systems [13], aimed at improving throughput. It is usually categorized into distinct classes. For example in state parallelism [5], [30], [8] an execution unit handles a state of the NFA mechanism. In data parallelism [5], [16], [21], the event stream is split and sent to different execution units. Both approaches suffer from limits on the degree of parallelism they can achieve, whether because of a fixed number of states in state parallelism or a limit on the number of partitions into which the stream can be split. In contrast, our hybrid parallelism model can utilize all available cores in the system, while providing efficient load balancing among the execution units.

IX. Conclusions and Future Work

In this paper, we presented a novel approach for parallelizing CEP applications. To the best of our knowledge, our model is the first to combine the two main parallel CEP approaches, state-based and data-based parallelism, uniting the strengths of both approaches in a way that overcomes their major disadvantages. Our extensive experimental evaluation on two real-life datasets demonstrated a significant throughput improvement over the state-of-the-art parallel CEP method, while achieving lower latency and consuming less memory. Our future research efforts will include supporting fully adaptive and distributed use cases.

References

[1] http://www.eoddata.com.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[3] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, 2008.

[4] H. Andrade, B. Gedik, K.L. Wu, and P.S. Yu. Processing high data rate streams in system s. *Journal of Parallel and Distributed Computing*, 71(2):145–156, 2011.

[5] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 3–14, New York, NY, USA, 2013. ACM.

[6] M. Blount, M. Ebling, J. Eklund, A. James, C. Mcgregor, N. Percival, K. Smith, and D. Sow. Real-time analysis for intensive care: Development and deployment of the Artemis analytic system. 29:110–8, 05 2010.

[7] Diane J Cook. Learning setting-generalized activity models for smart spaces. *IEEE intelligent systems*, 2010(99):1, 2010.

[8] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *Journal of Parallel and Distributed Computing*, 72(2):205–218, 2012.

[9] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

[10] M. Dayarathna and S. Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):33:1–33:36, February 2018.

[11] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology*, pages 627–644. Springer-Verlag.

[12] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, and M. Mock. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software*, 127:217 – 236, 2017.

[13] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis. Complex event recognition in the big data era: a survey. *The VLDB Journal*, 29(1):313–352, 2020.

[14] M.I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices*, 41(11):151–162, 2006.

[15] M. Hill, M. Campbell, Y. C. Chang, and V. Iyengar. Event detection in sensor networks for modern oil fields. In *DEBS*, volume 332 of *ACM International Conference Proceeding Series*, pages 95–102. ACM, 2008.

[16] M. Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM.

[17] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[18] M. Jung, J. Park, J. Blieberger, and B. Burgstaller. Parallel construction of simultaneous deterministic finite automata on shared-memory multicores. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 271–281. IEEE, 2017.

[19] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45. ACM, 2015.

[20] A. Koliousis, M. Weidlich, R. Castro Fernandez, A.L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, pages 555–569, 2016.

[21] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, 2(4):274–286, Aug 2015.

[22] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel. Graphcep: Real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 309–316, 2016.

[23] Y. Mei and S. Madden. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*, pages 193–206. ACM, 2009.

[24] M.A.U Nasir, G.D.F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148. IEEE, 2015.

[25] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 80–91, 2015.

[26] N. P. Schultz-Møller, M. M., and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*. ACM, 2009.

[27] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[28] R. Sinya, K. Matsuzaki, and M. Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *2013 42nd International Conference on Parallel Processing*, pages 220–229. IEEE, 2013.

[29] Z. Wang and M.F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 307–318, 2010.

[30] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with fpgas. *Proceedings of the VLDB Endowment*, 3(1-2):660–669, 2010.

[31] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418. ACM, 2006.

[32] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, pages 217–228, 2014.

[33] Q. Zhou, Y. Simmhan, and V. K. Prasanna. Incorporating semantic knowledge into dynamic data processing for smart power grids. In *International Semantic Web Conference (2)*, volume 7650 of *Lecture Notes in Computer Science*, pages 257–273. Springer, 2012.