Abdur Rafay Saleem
University of Central Florida
ab464825@ucf.edu

***Abstract*—This is a review of the paper on parallelizing complex event processing (CEP) by Edward Curry and others. It is divided into four sections: an overview of the research problems, the description of old solutions, a detailed breakdown of the new proposed technique, and a critical assessment of its effectiveness.**

I. RESEARCH PROBLEM

Real-time data streams are increasingly common in the world of technology and are of an ever-growing nature. These streams require an architecture that can process them fast, accurately and at scale. These architectures work on the principle of pattern detection to comb through these streams of raw data to identify meaningful events. A popular technique is *Complex Event Processing (CEP)* that leverages distributed and parallel computing to optimize and analyze these events. However, it is tightly coupled in nature which limits its ability to work effectively with existing parallel algorithms. The main challenges are to keep the computation time, the latency and resource utilization of CEP to a minimum. Moreover, having to perform this in real-time poses even stricter constraints of time. Also, identifying and comparing dynamic patterns from thousands of possibilities simultaneously increases the complexity of the challenge. Finally, all this should support scalability and minimize operational costs.

The paper explains the CEP technique to be a centralized processing system that takes *primitive events* from the sources and combines them into complex higher-level events. It joins the primitive events in an incremental fashion towards a combined pattern called a *partial match.* This partial match is continuously compared to the *full match* that is defined by the user's desired event. The partial match is a graph of states depicting different stages of the detected pattern. Once the graph matches the goal, the event is triggered.

Two techniques have been suggested to parallelize this entire process: (a) *data-parallel and (b) state-parallel*. Data parallel approaches distribute the input stream to different partitions based on predefined rules. Each partition routes data to execution units and partial matches are created. Finally, a merger joins these matches to create and identify the full match. Main limitations of this are redundancy in partitioning and suffering from data skew. The state parallel approach partitions the input stream based on events and each type of event is sent to a different execution unit. The partial matches are passed on from one unit to the next sequentially before achieving a full match. The ordering of units is pre-set. While this solves the redundancy and skew challenge it limits parallel units to only the number of states.

The paper proposes a new solution called HYPERSONIC which creates a hybrid of these techniques to take the best of both worlds. It takes a *hybrid-parallel approach.* Its two-tier system divides the workload according to states, followed by data operators within each state unit. This allows unbounded limits for parallel work and extends a shared memory system within the state units to minimize communication overhead. This system can scale quickly and effectively to all volumes of data, all varieties of data and available resources. It allows efficient allocation of execution units and data between them to handle all kinds of loads.

Technical contributions of the paper include the proposal of a hybrid-parallel approach for a two-tiered load balanced CEP system. It lays out the details of the architecture and a high-fidelity design of the system. It expressingly addresses the challenges faced by previous solutions and compares the solution against popular models by extensive experimentation. Also, it compares the performance speedup of the solution over the baseline of other state-of-art systems. Finally, the author also suggests areas of extension on this approach to create additional models that can cater to domain specific challenges.

II. OLD TECHNIQUES

The previous techniques introduced in a paper take a data-parallel approach. Each of the techniques replaces the operator with a *split-process-merge* assembly that works with two input streams. For multiple streams, many such assemblies are strung together in sequence. They are as follows:

A. Round Robin (RR)

The Round-Robin strate-gy is a technique employe-d to distribute incoming information or events across multiple- servers. The obje-ctive is to guarantee that e-ach server rece-ives an equal number of e-vents. It follows a time based window approach for distributing the workload evenly amongst all units.

The syste-m counts each event that arrive-s. It then checks if the numbe-r of events that have arrive-d matches the expe-cted number. If they match, the- system sends the ne-xt set of events to a diffe-rent server. Howe-ver, if the number of e-vents doesn't match the e-xpected number, the- system keeps se-nding the incoming events to the- current server. At the- same time, it kee-ps track of the number of eve-nts that have arrived. Finally, all servers send the data to the merger. This process guarante-es the eve-n distribution of workload among all servers. It helps improve- efficiency and minimize de-lays, especially when de-aling with large volumes of data that require- real-time processing. Think of it as a we-ll-structured assembly line, whe-re every worke-r (or server) rece-ives an equal share of the- work, ensuring that no one is overwhe-lmed while others re-main idle. However, the drawback is that it doesn't consider the workload differences amongst the servers.

B. Join the short queue (JSQ)

The Join-the--Shortest-Queue (JSQ) strate-gy is a method used to manage incoming data by directing them to the- server with the le-ast processing workload.

The syste-m operates by monitoring eve-ry incoming event and comparing it to the e-xpected number using a function called *ProcessEvents*. Once- the total number of eve-nts matches the expe-cted number, the syste-m assigns the next batch of eve-nts to the server with the- shortest queue for proce-ssing using a function called *UpdateServer*. This function retrieves processing load information from all downstream servers and selects the one with the lowest workload and sets it as *newPref*. This ensures efficie-nt event distribution and processing. If the total numbe-r of events does not match the- expected numbe-r, the system will consistently dire-ct incoming events to the same- server it has bee-n using. Meanwhile, it will maintain a running tally of the re-ceived eve-nts in a counter. Finally, all servers send their matches to the merger which combines them into a full match. This leads to increase-d efficiency and reduce-d waiting time, particularly when dealing with large- amounts of data that require quick processing. It's similar to a production line- where each worke-r receives an e-qual share of the workload, preve-nting overload for some workers and unde-rutilization for others. However, it can still suffer bottlenecks because the shortest queue might still have the largest memory load. Also, it can't keep information about downstream server loads in real-time.

C. Least loaded server first (LLSF)

The Le-ast-Loaded-Server-First (LLSF) strate-gy is a dynamic method for assigning incoming data or events to the- server with the le-ast amount of load. The server with the- lowest memory usage is conside-red to have the le-ast load.

The function calle-d *ProcessEvents* is responsible- for handling incoming events. It kee-ps track of the number of eve-nts received and compare-s this count to a predetermine-d expected numbe-r. If the count matches the e-xpected number, the- function calls the *UpdateServe-r* function to determine the- server with the le-ast amount of memory usage. This serve-r is then designated as the- preferred se-rver for processing the ne-xt batch of events. The e-vents in the next batch are- sent to this preferre-d server. Howeve-r, if the count doesn't match the e-xpected number, the- function continues to send the incoming e-vents to the current se-rver and increments the- count. The Update-Server algorithm chooses the- server that has the le-ast amount of memory usage. To do this, it examine-s the memory usage of all se-rvers and keeps track of e-ach value. It then compares the-se values to identify which one- is the lowest, indicating the se-rver with the least me-mory usage. The prefe-rred server is the-n set to be this serve-r.

The LLSF strate-gy employs continuous monitoring of server me-mory usage in real-time. It directs incoming eve-nts to the server with the- least used memory. This approach e-nsures that the workload is eve-nly distributed across all servers, re-sulting in improved efficiency and re-duced waiting times.

LLSF outperforms the RR or JSQ because it considers the load size and can maintain a balanced parallel workload. Therefore, despite increasing the input rates it doesn't suffer bottlenecks and produces the highest event throughput.

III. NEW TECHNIQUE

The previous techniques all depend on having a good partitioning scheme which despite being the most optimal can cause some data duplication in case of event overlaps between partitions. Moreover, they can not do anything beyond limited parallelization if the data is skewed. Currently, the paper introduces a new technique to solve the problems but on a single pattern match. In the future this can be extended to multiple patterns. The new technique follows a hybrid-parallelization approach:

HYPERSONIC (HYbrid ParallElization appRoach for Scalable cOmpLex eveNt processIng appliCations.)

The de-scribed method is built upon a parallel archite-cture with two tiers to identify patte-rns. An automaton, a self-operating machine that follows a se-t sequence of actions, is use-d to represent the- pattern. In the outer laye-r of the architecture, a state-parallel approach is followed for splitting. The e-xecution units are assigned to e-ach state of the automaton based on the-ir expected workload. In the- inner layer, a data-parallel approach is followed for further splitting. The incoming data stre-am is divided among the available e-xecution units using a local load balancing strategy that operate-s in parallel.

E-ach state is represe-nted by an agent. These- agents handle all the calculations associate-d with their respective- states. The outer laye-r of the automaton allows for parallel exe-cution of the agents, while the- inner layer processe-s the units in a data-parallel way. This design e-nsures efficient and e-ffective operation of the- automaton. The age-nts receive two input stre-ams. One is the output stream of partial matche-s from the agent before- them. The other is a substre-am of the system input stream that is limite-d to the specific eve-nt type neede-d. Each agent then compares the- new events with the- accepted partial matches and se-nds out the extende-d matches into the output stream.

The inne-r parallelization layer is built on a *shared-me-mory* model. In contrast, the outer laye-r does not rely on that assumption. Adjacent age-nts communicate through a producer-consumer que-ue that is shared by their e-xecution units. As a result, this method can be- used in both single-serve-r and mixed environments.

B. Agents

Agents in this archite-cture receive- two input streams: the *match stream* (MS) and the- *event stream* (ES). Each age-nt produces an output stream. To store e-vents and partial matches, agents have- two local buffers: an *event buffe-r* (EB) and a *match buffer* (MB). When a new e-vent or partial match is receive-d, it is compared against the items in the- opposite buffer. Extende-d matches are then adde-d to the output stream. The ite-m is also stored in its respective- buffer for future combinations. Parallel e-xecution units in the form of *eve-nt workers* and *match workers* handle the-se tasks. The te-chnique also incorporates a process for re-moving outdated events and matche-s from their appropriate buffers to avoid ove-rflowing the buffers and ensure- accurate evaluation. The syste-m removes expire-d events by considering the- timestamp of the most rece-nt partial match, and it eliminates partial matches base-d on the timestamp of the most re-cent event.

C. Execution Units Allocation

In this architecture-, the outer layer divide-s the available exe-cution units among the agents. The inne-r layer then assigns each e-xecution unit a role as an *eve-nt worker* or a *match worker*. The purpose- of the outer load balancer is to optimize- system performance by partitioning the- execution units among the age-nts.

To preve-nt possible congestion caused by multiple- workers trying to write to the same- buffers at the same time-, the buffers are divide-d among them. Each worker takes care- of a specific part of the EB or MB. When a worke-r receives a ne-w event, it communicates with all the- matching workers to access their corre-sponding MB parts and then adds the eve-nt to its own EB part. This approach enhances the ove-rall efficiency of the syste-m and enables it to function fully in a distributed se-tting.

The inner layer assigns diffe-rent roles to each e-xecution unit; event worke-r or match worker. At the start, worker role-s are allocated based on a simple- heuristic. Roughly half of the units are randomly se-lected as eve-nt workers, while the re-maining units become match workers. Howe-ver, this approach often results in worke-rs being idle for exte-nded periods due to diffe-rences in values. As a re-sult, the system's performance- is not optimal. To enhance- stability, a role-dynamic model is utilized. Whe-n the system starts, each e-xecution unit assigned to an agent is give-n a primary role and also takes on a secondary role-. During runtime, an execution unit prioritize-s its primary role. If there is no curre-nt work available, it temporarily switches to the- secondary role and checks the- second input stream for any new data. This strate-gy efficiently distributes the- workload on the agent and helps addre-ss situations where one input stre-am has a significantly higher rate than the othe-r. As the execution units alternate between their roles, they have to simultaneously manage fragments of both the EB and the MB and satisfy access requests for both from other units. While this leads to more synchronization operations as compared to the basic model, the benefits greatly outweigh this negative impact.

HYPERSONIC demonstrate-s substantial throughput improvements, reaching up to a thousand time-s faster than non-parallelized syste-ms. Its memory balancing sche-me avoids storing duplicate partial matches. This e-nsures optimal distribution of memory accesse-s and better cache utilization. Moreover, using a *queue-based* input avoids bottlenecks. The purging mechanism also helps it maintain free memory.

IV. CRITICAL ANALYSIS

In the era of emerging stream management systems, managing complex streams of events poses significant challenge-s. This article outlines the HYPERSONIC CEP as a solution, me-rging the popular data-parallel and state-parallel approaches and creating a hybrid and scalable model. Validated against several state-of-the-art real-world CEP systems, the HYPERSONIC effe-ctively supports increased systems loads, data variations and a significantly higher threshold. Its pattern detection mee-ts stream management systems' real-time latency constraints and uses less memory.

The approach proposed by the paper can be extended with further research. Future e-fforts can be aimed towards extensions towards its architecture. These may include full distributed execution, pattern detection across multi and nested patterns, and better evaluation mechanisms. However, it also inherits some limitations of its hybrid counterparts. Similar to data-parallel approaches, the on-the-fly executions can result in highly fluctuating execution unit allocation policy. Also, the pipelined nature of the state-parallel approach in the outer layer raises the lower bound of the event detection latency. A limitation that isn't mentioned is the assumption of the execution units to be homogenous, while in reality they could have very different capabilities which can affect the performance.

Overall, the paper performs a good job of combining old approaches into an effective technique for complex event processing. It goes over the most common challenges of CEP and provides extensive experimentation to address these. However, this technique is a first attempt at hybrid-parallelization and future work will only continue to benefit from research.