Distributed Computation of the Critical Path from Execution Traces

Pierre-Frédérick Denys
Polytechnique Montréal
Quebec H3T 1J4
pierre-frederick.denys@polymtl.ca

Quentin Fournier
Polytechnique Montréal
Quebec H3T 1J4
quentin.fournier@polymtl.ca

Michel R. Dagenais
Polytechnique Montréal
Quebec H3T 1J4
michel.dagenais@polymtl.ca

***Abstract*—Due to the ever-increasing number of computer nodes in distributed systems, efficient and effective tools have become crucial for their analysis. Although several efficient methods have been proposed to monitor and profile distributed systems, tracing remains the most effective solution for in-depth system analysis. Tracing is the act of collecting a trace, which is a sequence of low-level events generated by the kernel or the userspace. After data collection, the most important part is the event analysis. The paradigm and choice of graphs determine the ability of the user to detect abnormal behaviors and identify their root cause. Although tracing is a highly effective approach to analyzing complex systems, the scalability of the current analysis tools is limited. As a consequence, tracing is often impractical for large distributed systems. This paper identifies the shortcomings of the current approaches, most notably the critical path computation and the trace file transfer between nodes. Then, this paper proposes new solutions to these drawbacks, most notably a distributed algorithm to compute the critical path, that does not aggregate all traces in a single node, and an efficient architecture to perform tracing on distributed systems. These new solutions are made publically available.**

***Index Terms*—Distributed Systems, Critical Path, Active Path, Trace Compass, Container Clusters, Docker, Tracing, MPI, Parallel Computing, Message-Oriented Middleware.**

I. Introduction

Monitoring is widely applied to distributed systems to efficiently collect metrics on each node. This approach produces discrete data that provides an overview of the system load and behavior over time. However, individual performance metrics often cannot reveal the root cause of performance problems in distributed systems. This limitation of monitoring is especially true in remote systems and large clusters, such as production servers, high-performance computing centers, and distributed systems.

Instead, *tracing* provides a continuous view of the behavior of the system and allows a deeper analysis of the system. In particular, tracing allows developers to find performance problems originating from the kernel and the userspace. Furthermore, tracing allows kernel and userspace events from multiple systems to be observed and analyzed on a single graph, thanks to an accurate time synchronization. Several analysis methods have been proposed to efficiently exploit these large amounts of data, including finding the critical path across an application execution graph.

Critical path analysis highlights the events linked to a process execution, across a full execution graph with all system events. The result of the processing is presented to the user as a time graph, where each involved process is represented on a separate line and where kernel events are represented across the timeline. Arrows between events represent signals between events, which allow the user to understand the application execution and to discover the cause of blocking events and errors.

This article surveys previous work on critical path analysis, including methods that could be improved to achieve large-scale distributed system analysis. In particular, this paper puts under the microscope the poor scalability of the current algorithms used to compute the critical path of a process. The limitations of current analysis tools, data collection, storage, and distribution during the analysis are characterized by benchmarks. Then, this paper introduces solutions and mitigations to the aforementioned limitations. The scope of this research work is the analysis of traces issued from large distributed computing clusters such as those used in HPC. The goal is to handle critical path processing of parallel computation processes such as MPI. Relatively small traces (a few gigabytes) on many nodes must be analyzed. Although this paper focuses on critical path computation, the proposed methods are suitable to scale other analyses, such as time and flame graphs.

This article aims to answer the following research question: *How to improve the efficiency of the current methods to analyze traces collected on large distributed systems?*

The main contributions of this work are (1) the identification of limitations of the current tools, (2) the introduction of a distributed critical path algorithm, (3) efficient communication patterns to exchange information between nodes during analysis, and (4) the architecture to compute the critical path in parallel.

The remainder of this paper is organized as follows. Section II surveys the related work. Section III introduces the methodology to parallelize the critical path computation efficiently. Section IV presents and discusses the results of the proposed approach on three use-cases. Finally, Section VI proposes interesting future work and concludes this paper.

II. Related Work

This section surveys state-of-the-art methods related to every aspect of the critical path analysis: data collection, data storage, data transfer, critical path computation, and visualization. In particular, this section investigates the ability of the methods to perform well on distributed systems.

A. Trace Data Collection

Tracing is a non-intrusive and lightweight method to record system activity by collecting low-level events from the kernel space, including system calls, interruptions, and network events, and from the userspace (program execution). Events are stored in a trace and comprise a name, a timestamp, and possibly a payload. Most notably, tracing helps detect abnormal behavior such as real-time deadlines, unexpected response time, and memory, load, or concurrency issues. As stated by Poirier et al. [1], tracing may be seen as a high-performance `printf`.

Tracers apply, and have been applied, to a wide range of systems [2] such as application servers, clients, real-time systems, embedded systems, and heterogeneous distributed systems. The low overhead of tracers makes them suitable for heavily loaded systems and low-resource devices such as IoT systems and containers.

Several tracers have been proposed throughout the years as listed by Toupin [3]. The Linux Trace Toolkit: next generation (LTTng) [4] is the preferred choice for this research, as it is lightweight and suitable for kernel and userspace tracing. Note that the proposed methods apply to the other tracing tools.

Tracers typically produce a large amount of data, and since tracing is used to analyze loaded systems with complex architectures [5], the traces must be stored efficiently.

B. Trace Data Storage

Due to the high throughput of tracers, data structures optimized for time series, such as described by Prieur-Drevon [6], are mandatory to store traces and critical paths.

While B-trees [7] are a widely popular data structure for ordered collections stored in files and databases, they are not suitable for interval storage. As a solution, State History Trees [6] were developed, a data structure specifically designed for large time interval data that stores the data on disk rather than in memory. The State History Tree is organized as a tree where multiple intervals are stored on-the-fly in each node and committed to the disk when full without rebalancing thanks to the chronological ordering. Additionally, this method is suitable for short time intervals to the main structure (trace file) and can be used in a distributed paradigm [8]. As of the writing of this article, State History Trees are the preferred data structure to store traces.

Alternatively, Chan et al. [9] introduced another hierarchical trace file format called SLOG2, a scalable logfile format. However, this format is slower and less versatile than State History Trees and is therefore not as popular.

The manipulation of large trace files is challenging because of their size. As a solution, Noeth et al. [10] proposed to add compression at the trace file creation on creation nodes. This method is part of the ScalaTrace project and has been tested on an MPI cluster, a representative example of a massive trace data generator.

Although efficient data structures exist to store traces, they are often analyzed on a different node after collection, and communication between nodes remains challenging.

C. Communication Between Nodes

The trace data collected on each node by a daemon is then transferred to a storage node or directly to the analysis node. In both cases, trace data must be sent over the network.

The communication between distributed nodes is generally achieved with Remote Procedure Calls (RPC) or, more recently, with Message Oriented Middleware (MOM) [11]. While RPC standard behavior is blocking, MOM benefits from a straightforward implementation of queues for asynchronous communication and quality of service on unreliable networks [12, 13]. Consequently, Message Oriented Middleware tends to be preferred.

The communication between tracing nodes and the analysis node is better achieved with a specific protocol such as the Trace Server Analysis Protocol (TASP, also called TSP) [14]. This protocol is similar to the Langage Server Protocol (LSP) [15], except that instead of being based on JSON-RPC, TSP is based on HTTP Representational State Transfer (REST). The main advantage of this protocol is the possibility to encode data using a binary format.

The communication capacity of distributed systems is variable. Indeed, networks are often unreliable and slow, and the bandwidth is prioritized for services rather than monitoring, tracing, or backup in production environments. As a result, the amount of data and the communication frequency must be reduced. In particular, the transmission of complete trace files must be avoided as much as possible. One solution to solve the communication challenge is for tracing daemons to periodically send the trace files to storage nodes using asynchronous communication (e.g., MOM) and to perform as much pre-computation as possible on the storage nodes. Then, only the necessary data for the analysis is sent to the analysis node using a specifically-designed protocol (e.g., TSP).

D. Critical Path Computation and Analysis

The critical path, also called the active path, corresponds to the shortest path between the elements of a network and represents the dependencies between the tasks. The critical path has been applied early on to software analysis; notably, Yang and Miller [16] used an execution graph to determine the critical path of software execution in 1988. The critical path is a suitable method to solve performance problems and plan the realization of complex projects [17] in numerous domains such as activity networks with time constraints [18], networks with fuzzy activity [19], TCP transactions [20], and fuzzy numbers [21]. The reader is referred to the work of Montplaisir et al. [22] for a detailed introduction to the critical path.

Interestingly, the critical path allows tracking program execution and precisely profiling programs during execution. Indeed, a program execution speed is directly tied to that of the critical path [23]. For instance, Tullsen and Calder [23] introduced a critical path profiler, and Ball and Larus [24] proposed a method for critical path profiling with minimal overhead.

Since the critical path computation is related to searching for the shortest path, one may use well-known algorithms such as Dijkstra [25]. However, due to the sequential nature of the Dijkstra shortest-path algorithm, parallelizing the computation yields only a marginal average speed-up of 10%. The reader is referred to the work of Giraldeau and Dagenais [26] for a thorough description of the main algorithm used to compute the critical path.

Critical path methods in distributed systems and virtualized environments require careful adaptations as described by Nemati et al. [27]. Notably, since the critical path comprises dependencies between multiple trace files collected on different nodes, ensuring a near-perfect time synchronization between traces is crucial. The time synchronization accuracy can be increased at collection time with an efficient clock synchronization of traced systems [28] such as with the Precision Time Protocol (PTP) or at analysis time with an offline trace synchronization algorithm such as the one presented by Poirier et al. [1]. The reader is referred to the work of Wu et al. [29] for a complete and detailed description of the analysis of the large distributed systems. Critical path computation requires sequentially processing every trace file on a single analysis node, which is problematic for numerous large traces. Schulz [30] showed that it is possible to extract the critical path for applications running on up to 128 processors. Nevertheless, there is a real and necessary need to improve the critical path computation.

The critical path is a method used to analyze large distributed systems such as search engine and their internal communication systems such as RPC. It allows precise latency analysis as stated in Eaton et al. [31]. An aggregation algorithm for distributed tracing is presented, but only a few details are given on the aggregation process. Another recent article from Morrison [32] presents an aggregation algorithm in Jaeger achieving a summation of the time contributions of operations done in multiple remote traced nodes.

E. *Visualization of Large Traces*

Time graphs are a popular visualization for trace analysis. As illustrated in Fig. 1, time is represented along the x-axis, and each vertical line represents a processor tree and its cores, or a process tree and its children. On each line, events are represented by a rectangle corresponding to the run time state of this core or process. Notably, time graphs allow a fine-grained representation of kernel or network-related events.

One way to mitigate the aforementioned communication and computation challenges is to limit the amount of data fetched, and computations, to the period selected by the user in the visualization software. For instance, the graph displayed to the user can be partitioned into frames (areas of pixels), each of which can be processed independently. Note that this segmentation is possible across time but also across resources. Although such an approach is straightforward when the content of the frames is independent of each other, it becomes much more challenging when data from other frames are required for the computation of the selected frame. Chan et al. [9] pointed out this challenge and proposed a solution: the bounded-box method based on a specific trace format. Note that their solution is suitable for other trace file formats with annotations.

The end-user analysis environments play an essential role and should be carefully considered. In our opinion, there are two typical users: software developers or architects, and system integrators or administrators. Let us first consider the analysis environment of software developers.

Developers often have to observe the performance of their application and solve issues such as race conditions. They generally work on test clusters with a limited number of nodes, and they can control the trace collection and thus the amount of data. As a result, the analysis tool is ideally integrated into their IDE, and a fine-grained integration can be configured between the trace data and the source code. Notably, function calls found in traces can be linked to code files, and the developer can easily find the code related to a system event or unexpected behavior. Trace compass is one of these tools and is integrated in the Eclipse IDE [33] as illustrated in Fig. 2.

Unlike developers, system integrators do not control the data since they work on already generated trace files. The data collection policy is configured at the system set up and is typically exhaustive, as they need a high level of details to understand the issues. Consequently, integrators and administrators typically handle larger traces than developers. Additionally, the same time-expensive processing is usually performed for each of the multiple users involved in the system monitoring and analysis. In this case, a better approach is to rely on a shared visualization service, and preprocess the data on tracing nodes, such that fewer resources are needed to perform final computations and views. Several effective methods are available for monitoring systems, but there is a gap on the trace analysis side. Indeed, the Theia Trace Viewer, released in 2017, is one of the first. Datadog can also be part of the solution. None of the current solutions offers a complete method for the efficient trace analysis on large clusters. This article will propose solutions to solve these gaps.

An interesting example of a distributed analysis architecture is the use of Promotheus nodes to collect metrics in a web application, and a Grafana client to display graphs. Indeed, a Promotheus node, linked to a storage service, runs the application periodically to retrieve the metrics and store them in the Prometheus database. These Promotheus nodes are generally replicated and work in clusters. When a user asks for a graph of the metrics on the Grafana client, the client sends a request to one of the Promotheus nodes. The Promotheus node processes the raw data and sends the resulting graph data to the client. As a result, the Promotheus nodes, similarly to computing nodes, perform the processing and transfer only the necessary information to display the graphs to the Grafana client, similarly to viewer nodes. This pattern could be applied

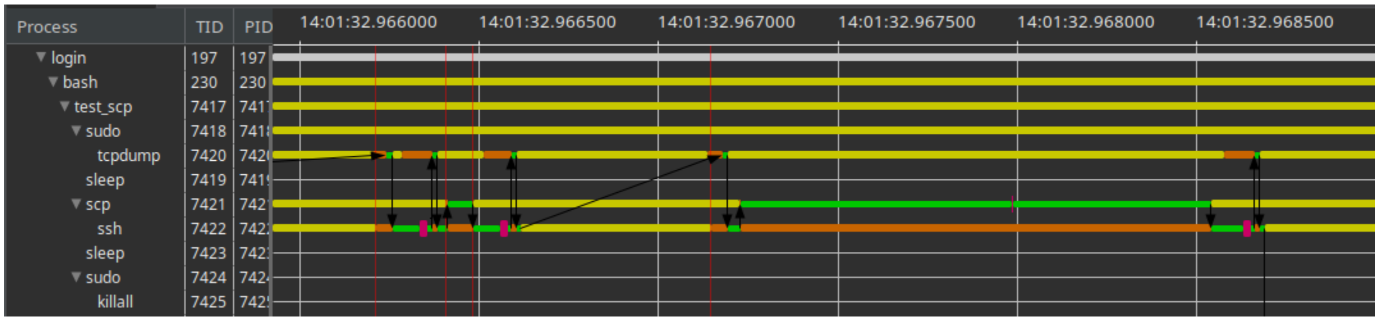

Fig. 1. A time graph representing a critical path view of a scp process in Trace Compass.

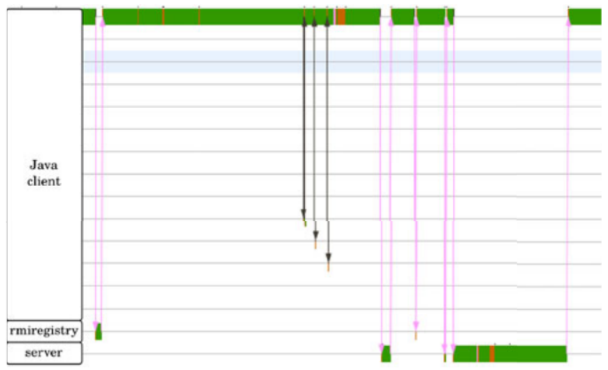

Fig. 2. The critical path view of a java client process communicating with a server in Trace Compass

to trace files analysis, to avoid file transfers and perform a parallel pre-computation.

For a review of tracing tools for large distributed environments, the reader is referred to the survey of Las-Casas et al. [34]. Numerous projects are available to achieve this goal, most notably Zipkin [35] and Opentelemetry [36].

III. Methodology to Parallelize the Critical Path Computation

This article presents two improvements to the critical path computation that increase the scalability in terms of system size and load: computation of the critical path and a proposal of an efficient architecture for the transmission of trace files between analysis nodes.

A. Requirements and Challenges

The goal of the critical path is to identify the dependencies between events collected simultaneously on different nodes. The traces recorded on the different nodes are transferred to the analysis node, where they must be synchronized due to the time difference between the system clocks. After synchronization, the analysis starts by extracting the execution graph for each trace. Then, the critical path of a process is computed at the user request.

First, let us focus on the system calls and signals required to compute the critical path. Contrary to the thread states `running`, `preempted` and `interrupted`, the `blocking` thread state can modify a program control flow. The event that unblocks the thread is either from the kernel mode or an interrupt. Contrary to the former, interruptions allow knowing the device that caused the waiting. Additionally, kernel events such as `sched_switch`, `sched_ttwu`, `interrupt_entry`, `interrupt_exit`, `inet_sock_local_in`, and `inet_sock_local_out` delimit events and interruptions. As a result, thread states and interruptions are the minimal information required to determine the unblocking relations between processes, and thus compute the critical path. The execution graph is processed with task states represented by horizontal edges and signals represented by vertical edges.

The two main steps of the critical path processing are the execution graph processing and the critical path computation. The first step can be easily parallelized as there is no dependency between traces. Each execution graph can be processed independently and simply stacked in the viewer with the same time scale. Parallelizing the second step is more challenging. Indeed, the computation of the critical path requires finding the links between multiple trace events, and the parallelization of the algorithm requires modifications. Since no pre-computation is possible, due to dependencies along the computing process, the critical path cannot be computed upstream on remote traces.

B. Methodology

Let us briefly introduce the current algorithm used to compute the critical path. The reader is referred to Giraldeau and Dagenais [26] for a detailed description of the algorithm.

Let us assume that the entire traces recorded on the nodes have been sent to the analysis node and synchronized. First, the execution graph must be processed for all the collected traces. Then, the critical path is computed at the demand of the final user by selecting a process.

After that, the system states of the selected process on the execution graph are iterated until a blocking state is encountered, and a procedure is called to retrieve the origin of this blocking state in other threads or host processes. Finally, this procedure performs a backward iteration until the interval start. In the process, if an incoming blocking state is detected, a recursion is made.

The primary drawback of the algorithm described above is the amount of memory usage due to (1) all trace files must be opened in the same environment to identify remote dependencies, and (2) the graph elements are typically stored in memory. Additionally, the algorithm as presented is not easily parallelizable due to the dependencies between the traces. The improved methodology described hereinafter addresses these two critical limitations of the current approach.

C. Improved Methodology

This section introduces improvements to (1) the scalability of the algorithm and (2) the data structure and communication between computing nodes.

1) Algorithm Parallelisation: Let us assume the simple architecture depicted in Fig. 3 for the sake of simplifying the algorithm optimizations, and eliminating environmental conditions such as network delays and speed. Let us also assume that the trace files are stored and opened locally in the viewer node.

A distinct data provider module is associated with each trace in the proposed methodology. As a result, traces are seen by the computing module as if they were stored on different hosts. Separate computing operations can be performed on each trace, and there is no coupling with trace storage. Additionally, a computational sub-module is associated with each data provider, to perform pre-computing operations.

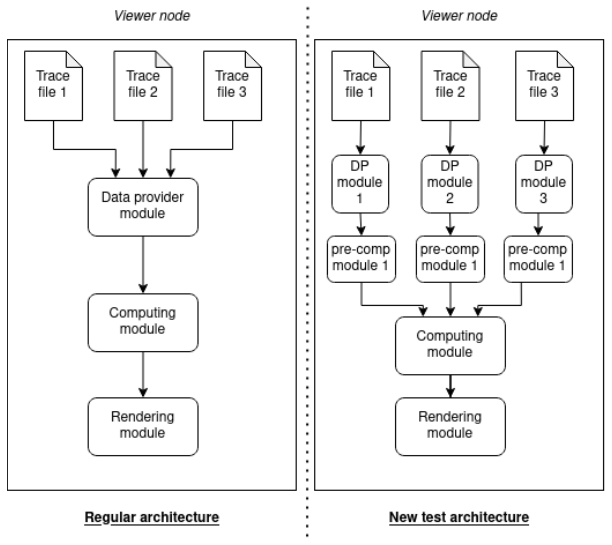

Fig. 3. Architecture to retrieve, compute, and display the critical path.

The first straightforward approach investigated involves resolving outbound edges: when a blocking state on the thread execution trace is encountered, a message is sent to all tracing nodes to find the cause of the interruption, based on the event time and attributes. This method was quickly discarded because of the communication burden between nodes, as illustrated in Fig. 4.

The second approach considered relied on the pre-computation of the graph and critical path. Instead of drawing the complete execution graph, the primary computation node

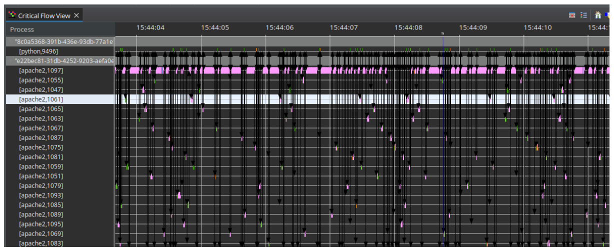

Fig. 4. Loaded critical path view.

sends a call to all preprocessing nodes in order to retrieve the graph elements linked to the selected time window. The computation node then performs the time synchronization, and the execution graph is displayed to the user thanks to the viewer node. Secondly, the user selects a process on which computing the critical path. At this point, the critical path is calculated on the current trace, and remote dependencies (also named blocking edges) are identified in the lower part of Fig. 5. At this point, the viewer node displays the first graph for the user without the remote dependencies solved, and a data structure containing the remote dependencies is sent to other computing nodes to solve them. The user can then see right away the different states of the process of interest, for instance executing in user or system mode, preempted, blocked waiting for the disk or another local process, or waiting for a remote dependency. If the process is blocked for a significant proportion of the time on remote dependencies, the user will want to know what other processes and events the process is waiting for. This is the asynchronous parallel computation that was sent to the other computing nodes for solving.

Remote dependencies are resolved using relevant identifiers; for example, an acknowledgment number is present for network events on the sender and receiver packet. This approach is time- and cost-efficient, since this computation is only performed when the user modifies the time window or process, and since the pre-computed intervals can be stored in a cache on the viewer node and reused. Lastly, the graph is updated step by step during the remote dependencies resolution. As a result, the user gets a first display way faster than with previous methods. The result is an uninterrupted link between events represented by arrows between system events across processes and traces. It is the critical path related to the selected process.

Several challenges have been encountered: (1) the origin of network interruptions is typically in remote traces, (2) the number of processes and nodes in the critical path is not known in advance, (3) the identification and retrieval of the interruption source depend on the nature of the event, and (4) the trace of the source event is not always available, which can lead to unidentified origin events that need to be identified on the graph.

The simplified pseudo-code of the method to resolve recursive remote dependencies is given in Algorithm 1. Contrary to previous methods, users do not have to wait for the entire graph processing and dependency resolving, since the function DisplayCP(CP) updates the graph during processing

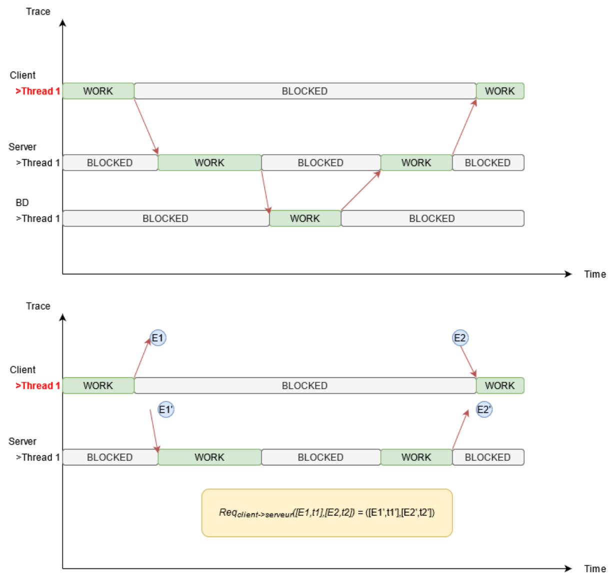

Fig. 5. Methodology to identify edges.

(Line 14). The user gets a first display very quickly, and more details appear as the remote computations proceed.

Although the proposed algorithm is not fully parallelizable, the new algorithm allows users to almost instantly see the critical path during processing.

2) Architecture for Remote Trace Analysis: Now that the critical path computation is possible on a distributed architecture, the traces and critical path pieces must be transferred from the computation nodes to client nodes with an efficient communication pattern.

Before introducing the proposed approach, the size, number, and frequency of the packets sent must be characterized. Previous experiments have concluded that the costlier transmissions are the trace files. Indeed, typically one or a few large trace files per node in the cluster must be transferred from the computation nodes to the analysis node.

In practice, the viewer node is often a laptop. The limited amount of resources available on the viewer node restricts the number and size of traces that can be processed. Such a limitation often results in long computation times, or inability to complete the computation. As a result, crucial improvements must be implemented to avoid transmitting complete trace files. For instance, the data transmission can be asynchronous, and the processing can begin as data comes in.

Thanks to the new algorithm presented in the previous section, there is no need to transfer all trace files to a single analysis node. As a result, two architectures are possible and answer different needs.

The first architecture, Fig. 6 is based on several computing nodes that collect traces from traced systems. These nodes collect traces and perform the pre-computation of the traces (execution graph processing). The user connects with his browser on one of these nodes, and it becomes a viewer node. As a result, this node is in charge of collecting data from others and computing final graphs for the user.

The main advantage of this architecture is scalability, as no trace files are on the traced systems. Also, it adds less overhead to the traced systems because traces are not analyzed on them. In this case, the parallelization depends on the number of computing nodes. This architecture is preferred when the elasticity of the system is high (the number of traced systems is variable), and the storage is not persistent, such as in container environments.

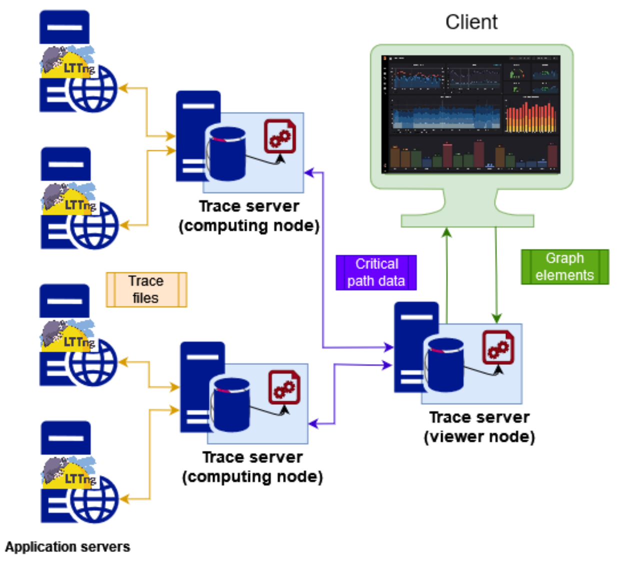

Fig. 6. Traces analysis architecture with dedicated computing nodes and common trace storage

The second architecture is more conventional: traced nodes store their own traces and play the role of computing nodes. Another dedicated trace server serves the requests of the clients and distributes the analysis between computing nodes.

This architecture is more efficient when the number of nodes does not vary and when traced nodes are able to handle both the application load and the tracing computation load. There is also less network load because trace files stay on trace nodes. However, the overhead is higher on the traced nodes. This disadvantage is counterbalanced by the greater parallelization of the critical path computation, due to the larger number of computing nodes.

These two architectures did not require any file transfer between the client and traced nodes, and allowed the distribution of trace analysis computation. The next section will focus on the parallelization speedup of the algorithm.

IV. Results

The detailed results obtained in this paper and the source code of the experiments are available in this git repository [1].

Let us define the following concepts to understand this section better:

[1]https://gitlab.com/pierrefrederick.denys/trace-compass-critical-path-distributed

Algorithm 1: Pseudo-code of the simplified algorithm to resolve the recursive remote dependencies in the viewer node.

```
Input: execution graph $G$, process $P$
Output: critical path $CP$
1  function resolveBlockedStates($G$,optional $P$)
2      $BlockingStates[\,]$ ← new JSON data structure
3      $ReturnedEdges[\,]$ ← new JSON data structure
4      while $event \neq eventInit$ do               // Backward exploration of the process states
5          if $event$ is blocking then                   // P is blocked by another local process
6              $state \leftarrow$ Resolve()
7          else if $event$ is blocked by network then    // P is blocked by another remote process
8              $BlockingStates[\,] \leftarrow event$
9          else
10             $state \leftarrow Unresolved$                          // Blocked state is unresolved
11     forall $ComputingNodes$ do
12         $ReturnedEdges[\,] \leftarrow$ SolveBlockingStatesRemote($BlockingStates[\,]$)
13         $CP \leftarrow ReturnedEdges[\,]$
14         DisplayCP($CP$)                                    // Update CP view during processing
15         resolveBlockedStates($CP$)     // Recursively solve blocking states in returned paths
16     return $CP$

17 function SolveBlockingStatesRemote($BlockingStates[\,]$)
18     for $BlockingStates[\,]$ as $BlockingState$ do
19         $states$ = FindCorrespondingProcessState($BlockingState$.parameters)
20         if $states$ is not null then
21             $ReturnedEdges[\,] \leftarrow states$
22         else
23             $state \leftarrow UnresolvedNetwork$
24             $ReturnedEdges[\,] \leftarrow state$
25     return $ReturnedEdges[\,]$

26 function FindCorrespondingProcessState($BlockingState$.parameters)
27     if $BlockingState.parameters.seq == currentState.parameters.ack\_seq$ then
28         while $currentState.Start\_time \geq BlockingState.Start\_time$ do
29             $states \leftarrow currentState$
30     return $states$
```

- The *viewer* is the software that transforms the processed data into a visual graph.
- The *viewer node* is the middleware node that runs a trace server to process the data sent by computing nodes and to answer requests from the viewer.
- *Computing nodes* are trace servers running on the application server or the trace collection nodes. The computing nodes store traces, retrieve and precompute graph elements from the traces, and send the results to viewer nodes.
- *Graph elements* are preprocessed data (events, vertices, and edges) computed from traces and transferred from computing to viewer nodes.

A. Representative Use-Case

In order to illustrate the contribution of the new method, a representative and demanding critical path computation over several large traces is performed with both the existing method and the proposed new algorithm. One key parameter to compare the two methods is the time required to display the first graph to the user. This benchmark is based on the traces generated by the transfer of a file composed of randomly generated text using SCP. The goal is to compare the processing time required by each algorithm version for these large trace files. Note that the computing time has been evaluated with several parameters, such as the trace size and the number of traced nodes.

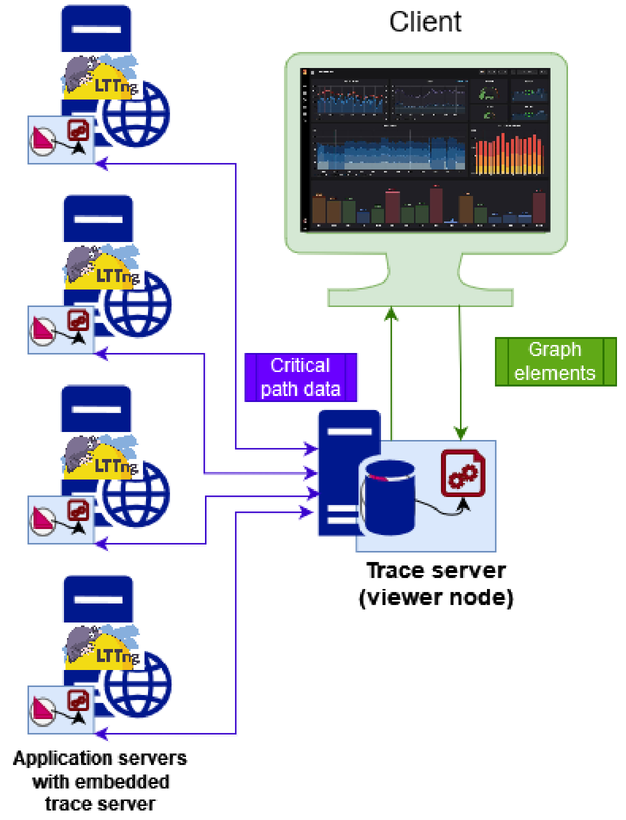
Client
Critical path data
Graph elements
Trace server (viewer node)
Application servers with embedded trace server
LTTng

Fig. 7. Traces analysis architecture with computing service running on traced nodes

The cluster used in experiments is composed of 21 physical computers, 20 configured as computing nodes and one more as a viewer. Each computer contains an i5-9400 at 2.9Ghz with six cores and 16GB of memory. Trace files are available on computing nodes as file transfer tracing is performed on them. Trace files are copied on the main viewer node in the conventional algorithm case. The analysis has been performed ten times in order to alleviate the fluctuations. More details on the experimental setup are available in the repository of the project.

B. Performance and Overhead

1) Processing Time: For this benchmark, SCP transfers have been recorded between four nodes. In order to remove network fluctuations, the four machines are on the same isolated network. The node activity and transfer time are proportional to the transferred file size, and so are the trace size and the number of events recorded. Therefore, four different sizes have been considered: 100 Mb, 1Gb, 10Gb, and 100Gb.

The first benchmark compares the processing time of the current and improved algorithms. In this experiment, the *total processing time* refers to the elapsed time between the selection of a process by the user and the display of the complete critical path. The *first display* is only defined for the new algorithm and refers to the elapsed time between the selection of a process by the user and the display of the critical path with unresolved remote dependencies. In other words, only dependencies involving other processes running on the same node are displayed.

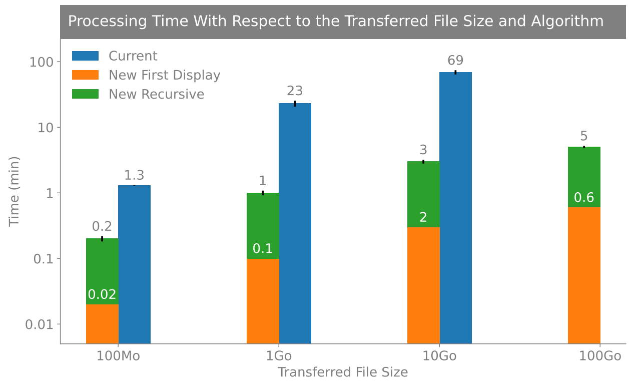

Fig. 8. Processing time of both algorithm versions with respect to transferred file size

Traces analysis of 100Go transferred file was not possible with the current algorithm. Fig. 8 shows that with the current algorithm, 63 minutes are required to perform the traces indexation, analysis, and display of the graph to the user when performed on a single node. The proposed new algorithm completes the full analysis in only 7 minutes and displays the first graph in only 45 seconds. As a result, the global analysis is nine times faster with the new method, and the first graph display happens in less than one minute rather than in an hour with the current method.

Additionally, the ”first display” time indicates that the user virtually instantly obtains a first graphical display. As a result, users can explore several critical paths of processes sooner without waiting for the entire remote dependencies to resolve.

2) Number of Traced Nodes: A second benchmark was performed with the same setup as the first one, except that the number of traced nodes is variable. For this experiment, an SCP process has been created in parallel between one node and a variable number of other nodes. Experiments have been repeated for several trace sizes to evaluate the processing time with respect to the number of nodes and trace size.

Fig. 9 reveals that, the processing time decrease with a larger number of nodes as the analysis process is distributed. A slightly higher computing time is required for larger traces due to the aggregation process. However, for larger traces, the total processing time of the new algorithm is lower, as the distribution of the processing between computing nodes more than compensates for the transmission time.

On Fig. 9, each curve is linked to a defined trace size, and the variable is the number of nodes involved in the computation. The graph shows that the best results are obtained when each computing node processes only one trace file. The processing time increases with the trace file size. As a result, the tracing architecture should use a regular trace file rotation to get multiple small trace files rather than a bigger one.

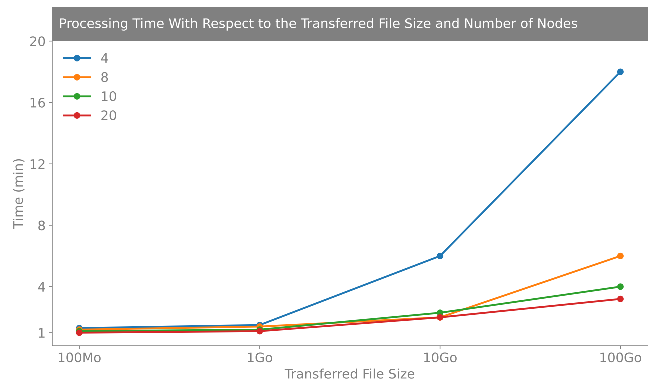

Fig. 9. Processing time of the new algorithm in terms of the number of nodes and trace size.

3) Performance: The CPU load has been recorded with the `mpstat` tool and is presented on Fig. 10, to give the average CPU load on each node. Significant performance gains are observed in the viewer: the critical path computation is faster and requires almost no resources on the client side. Since the client resources are no longer a bottleneck, the computing capacity has greatly increased. Additionally, since the critical path processing is performed on a time interval, the overhead is independent of the trace size and duration; only the trace data from the interval $[t, T]$ are transmitted from computing nodes to the viewer node. Indeed the global CPU consumption is slightly higher with the new method due to the data exchange between nodes and final data processing. However, the goal is to reduce the CPU consumption on the viewer, which is often the developer's laptop, for example. We want to take advantage of the high capabilities of the analysis cluster rather than the client.

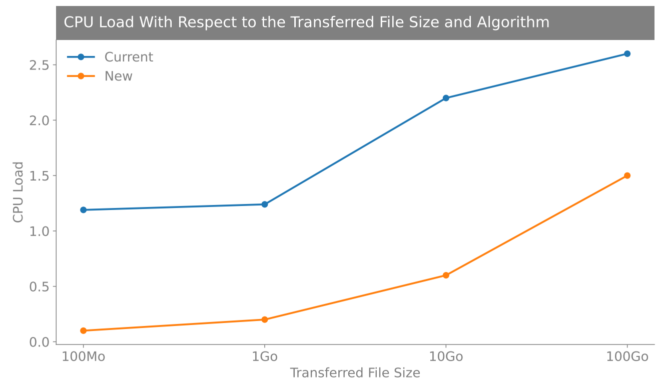

Fig. 10. Comparaison of performance of both algorithms on the viewer node.

C. Other Use-Cases

The expected use-case for the new method is to analyze distributed systems. The advantages of the new method will be highlighted below in two typical cases: MPI clusters and inter-container communication with message-oriented middleware.

1) MPI Clusters: MPI clusters are designed to perform massively parallel computations. Nonetheless, the synchronization between threads and barriers that causes busy-wait are the source of performance problems that are difficult to detect. Tracing helps follow communications between threads, and thus detect abnormal delays or patterns.

The main characteristic of these computing clusters is the number of nodes and messages exchanged. As a result, traces collected on each node are often enormous, and the execution graph will present many events due to the massive usage of messages between nodes. The critical path computation on such execution graphs requires substantial resources. With the conventional method, trace files should be transferred from each node to a central trace server in order to compute the critical path on this group of files.

The new method avoids transferring complete trace files. Instead, only the required graph elements are transferred from computing to viewer nodes. Generally, a good option is to avoid transferring trace files to avoid a network overload in the cluster. Furthermore, as resources are usually dedicated, the overhead of trace server execution on the calculation node is negligible.

2) Inter-container Communication with Message-Oriented Middleware: In clusters of containers, the communication between nodes is mandatory and efficiently achieved with Message-Oriented Middleware such as ZeroMQ. These communications are often asynchronous and, depending on the quality-of-service configured, some lost messages can be sent multiple times. Tracing in this environment is especially helpful in identifying slow communications or lost messages. However, the result data is typically challenging to exploit. The critical path is helpful and allows following messages between containers. If the message tag has been recorded in the trace, the critical path also allows the detection of duplicate messages. The second method explained in Denys et al. [37] allows collecting comprehensive information on messages. Then, it is possible to follow a message even if it passes through several containers. Thus, the critical path is useful to determine the full path of messages.

D. Discussion

The tracing cost has been compared between the current algorithm and the new one proposed in this article. Even though a full parallelization of the algorithm is impossible, a reasonable acceleration has been observed with the distributed algorithm.

The processing time is comparable, or even slightly longer, for small traces and many nodes, as the transmission time on the network is larger than the processing time on the local node. The best improvement is observed with a larger number of analysis nodes and smaller traces. As a result, a good tracing practice is to use file rotation and a larger number of tracing nodes to get a good distribution of file traces and, consequently, a better performance.

Several improvements could be explored in subsequent work and will be detailed in the next section.

V. Future work

The improvements made to the algorithm and the experiments assumed that trace files were already time-synchronized. If the traced nodes are out of sync, a synchronization process must be done on raw trace files. One interesting extension of this work would be implementing a distributed synchronization method into the tracing tool.

A valuable extension would be standardising the communications between the viewer and computing nodes, that is to propose a single API for several viewer tools.

Another research direction would be to focus on the viewer side to further improve the efficiency and extend the analysis. In our opinion, one of the most compelling features would be to filter trace files stored on remote computing nodes.

A noticeable improvement, especially in environments with a large number of nodes, consists in a method for optimised trace placement. Is it more efficient to use 20 analysis nodes for 20 traces, or make smarter placement of similar traces in groups? This question could be explored in subsequent work.

VI. Conclusion

Critical path analysis is a valuable tool for identifying bottlenecks in a distributed environment. However, the current algorithm to compute the critical path lacks the efficiency required for large distributed systems. As a solution, this paper introduces a method for parallelizing and scaling the current algorithm. The proposed architecture improves the storage and processing efficiency of traces, thereby reducing the limitation on the number of tracepoints and increasing the analysis capacity. Furthermore, the time required to display the graph is greatly reduced, allowing the final user to get the results faster. We believe this work to be the starting point for adapting current trace analysis tools to very large distributed systems.

VII. Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for funding this project.

References

[1] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, Aug. 2010, aRT2.

[2] T. Bertauld and M. R. Dagenais, "Low-level trace correlation on heterogeneous embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, p. 18, Jan. 2017.

[3] D. Toupin, "Using Tracing to Diagnose or Monitor Systems," *IEEE Software; Los Alamitos*, vol. 28, no. 1, pp. 87–91, Feb. 2011, num Pages: 87-91 Place: Los Alamitos, United States, Los Alamitos Publisher: IEEE Computer Society.

[4] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[5] M. Desnoyers, "Low-Impact Operating System Tracing," phd, École Polytechnique de Montréal, Dec. 2009, aRT2.

[6] L. Prieur-Drevon, "Structures de données hautement extensibles pour le stockage sur disque de séries temporelles hétérogènes," M.A.Sc., Ecole Polytechnique, Montreal (Canada), Canada, 2021, aRT2.

[7] G. Graefe and H. Kuno, "Modern B-tree techniques," in *2011 IEEE 27th International Conference on Data Engineering*, Apr. 2011, pp. 1370–1373, iSSN: 2375-026X.

[8] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, Aug. 2008.

[9] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, Jan. 2008, aRT2.

[10] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalable Compression and Replay of Communication Traces in Massively P arallel E nvironments," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–11, iSSN: 1530-2075.

[11] L. Qilin and Z. Mintian, "The State of the Art in Middleware," in *2010 International Forum on Information Technology and Applications*, vol. 1, Jul. 2010, pp. 83–85, iSSN: null.

[12] D. Menasce, "MOM vs. RPC: communication models for distributed applications," *IEEE Internet Computing*, vol. 9, no. 2, pp. 90–93, Mar. 2005, conference Name: IEEE Internet Computing.

[13] S. Vinoski, "Where is is middleware," *IEEE Internet Computing*, vol. 6, no. 2, pp. 83–85, Mar. 2002, conference Name: IEEE Internet Computing.

[14] Y. Chen Kuang Piao, "Nouvelle Architecture pour les Environnements de Développement Intégré et Traçage de Logiciel," M.A.Sc., Ecole Polytechnique, Montreal (Canada), Canada, 2022.

[15] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, "Towards a Language Server Protocol Infrastructure for Graphical Modeling," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 370–380.

[16] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *[1988] Proceedings. The 8th International Conference on Distributed*, Jun. 1988, pp. 366–373, aRT2.

[17] R. J. Willis, "Critical path analysis and resource constrained project scheduling — Theory and practice," *European Journal of Operational Research*, vol. 21, no. 2, pp. 149–155, Aug. 1985, aRT2.

[18] Y.-L. Chen, D. Rinks, and K. Tang, "Critical path in an activity network with time constraints," *European Journal of Operational Research*, vol. 100, no. 1, pp. 122–133, Jul. 1997, aRT2.

[19] S. Chanas and P. Zieliński, "Critical path analysis in the network with fuzzy activity times," *Fuzzy Sets and Systems*, vol. 122, no. 2, pp. 195–204, Sep. 2001, aRT2.

[20] P. Barford and M. Crovella, "Critical path analysis of TCP transactions," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2 supplement, pp. 80–102, Apr. 2001, aRT2.

[21] S. Nasution, "Fuzzy critical path method," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24, no. 1, pp. 48–57, Jan. 1994, aRT2.

[22] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "Efficient Model to Query and Visualize the System States Extracted from Trace Data," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer, 2013, pp. 219–234, aRT2.

[23] D. M. Tullsen and B. Calder, "Computing along the critical path," University of California, Tech. Rep., 1998.

[24] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, Dec. 1996, pp. 46–57, aRT2.

[25] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," in *2012 Proceedings of the 35th International Convention MIPRO*, May 2012, pp. 1811–1815, aRT2.

[26] F. Giraldeau and M. Dagenais, "Wait Analysis of Distributed Systems Using Kernel Tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, Aug. 2016, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[27] H. Nemati, F. Tetreault, J. Puncher, and M. R. Dagenais, "Critical Path Analysis through Hierarchical Distributed Virtualized Environments using Host Kernel Tracing," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019, aRT2.

[28] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 179–196, aRT2.

[29] C. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, "From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems," in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Nov. 2000, pp. 50–50, aRT2.

[30] M. Schulz, "Extracting Critical Path Graphs from MPI Applications," in *2005 IEEE International Conference on Cluster Computing*, Sep. 2005, pp. 1–10, aRT2.

[31] B. Eaton, J. Stewart, J. Tedesco, and N. C. Tas, "Distributed Latency Profiling through Critical Path Tracing: CPT can provide actionable and precise latency analysis." *Queue*, vol. 20, no. 1, pp. 40–79, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3526967

[32] J. Morrison, "CRISP: Critical Path Analysis for Microservice Architectures," Nov. 2021. [Online]. Available: https://www.uber.com/blog/crisp-critical-path-analysis-for-microservice-architectures/

[33] Y. Chen Kuang Piao, N. Ezzati-jivan, and M. R. Dagenais, "Distributed Architecture for an Integrated Development Environment, Large Trace Analysis, and Visualization," *Sensors*, vol. 21, no. 16, p. 5560, Jan. 2021, aRT2.

[34] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 312–324. [Online]. Available: https://doi.org/10.1145/3357223.3362736

[35] S. Mallanna and M. Devika, "Distributed request tracing using zipkin and spring boot sleuth," *Int. J. Comput. Appl*, vol. 975, p. 8887, 2020.

[36] A. Tobey and S. Spees, "Tracing bare metal with OpenTelemetry." San Francisco, CA: USENIX Association, Mar. 2022.

[37] P.-F. Denys, M. R. Dagenais, and M. Pepin, "Advanced tracing methods for container messaging systems analysis," 2021.