Scalable Distributed Computation of the Critical Path from Execution Traces

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

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

***Abstract*—Large distributed systems are challenging to analyze because latencies can quickly accumulate. This typically requires efficient tools such as tracing, to collect with low overhead a sequence of low-level events generated at runtime by the operating system (kernel space) and applications (user space). Most notably, tracing allows computing the longest path of dependencies between threads and resources (critical path), a powerful approach to investigate and understand performance issues in large clusters. However, the computation of the critical path across multiple nodes is costly and more difficult to scale. Several solutions have recently been proposed, such as a distributed algorithm for computing the critical path that bypasses the transfer of large trace files between nodes, thus speeding up the rendering of visualizations. Nevertheless, the current algorithms for the parallel computation of the critical path require much more computational resources for a moderate acceleration. This paper introduces a new algorithm to compute critical paths based on pre-computed indexes of the inter-node communications. It greatly reduces the computational cost of later queries for the critical path of different threads and requests. The algorithm has been implemented in Trace Compass and is publicly available.**

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

I. Introduction

Nowadays, a wide range of applications, such as scientific simulations, depend on large distributed systems. Such environments often comprise tens of thousands of nodes, each comprising multiple CPU cores and possibly GPUs. Due to their scale and complexity, analyzing their performance poses several new challenges: most prominently, the algorithms and data structures must be efficient and scalable. However, analysis tools for large distributed systems are challenging to develop and optimize, as they are large parallel applications themselves.

One suitable approach for analyzing the performance of large distributed systems is *tracing*. Tracing collects a sequence of low-level events, generated by the operating system (kernel space) and applications (user space) whenever a specific instruction, called tracepoint, is encountered at runtime. Notably, when the thread scheduling related events are traced, it is possible to compute the critical path corresponding to the longest path of dependencies between threads and resources. The critical path, for a thread or request, is an efficient and effective approach to identify and explain the root cause of bottlenecks in large-scale environments, such as network latency, storage contention, and cache issues [1].

The critical path computation can ultimately depend on all the threads in a system, and thus has been the main bottleneck for analyzing large traces lately. The initial algorithm introduced by Giraldeau and Dagenais [1] computes the critical path from the complete execution graph stored in memory. The available memory thus became the limiting factor to analyze very large traces. A new optimised disk based structure was developed to store the execution graph on the disk, while allowing quick access for traversing the graph to compute the critical path[1], thereby mitigating memory limitations.

More recently, a parallel algorithn to incrementally compute the critical path from distributed traces was proposed [2]. It brought interesting improvements, as larger traces could be handled by a cluster of parallel trace analysis nodes. Furthermore, the incremental computation improved the user experience by quickly showing the status of the thread for which the critical path computation was requested, including the periods where it was idle, waiting for some external events. Then, incrementally, were shown the details of the critical paths leading to those external events, recursively. However, the computation of the critical path remains a global computation, on the combined execution graph, and involves work from all the parallel analysis nodes.

This paper presents a new algorithm that can quickly prune the critical path computation, to only involve a small subset of the parallel analysis nodes, by indexing trace files using an *external communication index* (ECI). The three contributions of this research are (1) the identification of the limitations of the current tools, (2) common patterns of distributed trace analysis problems, and (3) a new algorithm that indexes the external communications, in order to reduce the computational cost of the distributed critical path computation.

The remainder of this paper is structured as follows. Section II surveys the related work and discusses the limitations of the existing tools. Section III introduces a method that remotely indexes the external communications in trace files to compute the critical path more efficiently. Section IV evaluates the new proposed method with real use cases, and discusses

[1]Unpublished work by Genevieve Bastien https://git.eclipse.org/r/c/tracecompass/org.eclipse.tracecompass/+/183418

its benefits as well as limitations. Finally, Section V concludes this paper by discussing relevant future research directions.

II. Related Work

The following literature review briefly surveys the related work on (1) the trace collection and storage, (2) the communication between nodes, (3) the computation of the critical path, (4) the visualization of distributed traces, and (5) the use of indexes in distributed systems. Since this paper focuses on large distributed systems, the efficiency of the methods is the main focus.

A. Trace Collection and Storage

Tracing is often used as a non-intrusive and lightweight method to record the system activity by collecting low-level events generated at runtime by the operating system (kernel space) and by programs (user space). Kernel space events include system calls, interruptions, and network events, which are necessary to build the execution graph and, subsequently, the critical path. Tracing is particularly helpful in detecting abnormal behaviors such as unmet real-time deadlines, unexpected response times, and memory, load, or concurrency issues. As observed by Poirier et al. [3], tracing is similar to a high-performance `printf`.

Tracing has been applied to a wide range of systems [4], including real-time, embedded, heterogeneous, and distributed systems. Due to their low overhead, tracers are particularly suitable for investigating heavily loaded systems, such as HPC clusters, and low-resource devices, such as IoT systems and containers. Several tracers have been proposed throughout the years as listed by Toupin [5]. The preferred choice for this research is the Linux Trace Toolkit: next generation (LTTng) [6], as it is lightweight and suitable for both kernel and userspace tracing. Nonetheless, the proposed methods apply to other tracing tools as well.

As tracers collect low-level events, they typically produce a large amount of data. Consequently, traces must be stored efficiently, especially on heavily loaded systems with complex architectures [7], and manipulating large trace files is challenging. As a solution, a compression of trace files has been used on the traced nodes during the collection. These methods have been compared by Noeth et al. [8] and are used to trace MPI programs on a large number of nodes, one of the use cases targeted by this project and a representative example of a source of massive traces. Nonetheless, scaling remains challenging, even with compression and efficient data structures, as the volume of traces generated grows with the number of nodes. After the traces are collected, further communication is necessary to merge the information or metadata from all the nodes, to compute global properties like the critical path. For this reason, traces are often centralized in order to be analyzed, which further poses a communication challenge.

B. Communication Between Nodes

Traces are either analyzed on the node where they were collected, or sent over to analysis nodes. Regardless, the result of the analysis must be transferred to the viewer nodes, and thus some data must be sent over the network [9]. However, the communication capacity of distributed systems is variable. Indeed, some networks are unreliable and slow or already saturated. Moreover, 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 should be avoided as much as possible. We discussed earlier the importance of having an efficient trace format. One solution to this 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 precomputation as possible on the nodes on which the traces are stored, such as filtering, compression, and snipping. Then, only the necessary data for the analysis is sent to the analysis node.

Different approaches are available to transfer the results from analysis requests. The communication between distributed nodes is generally achieved with Remote Procedure Calls (RPC) or, more recently, with Message Oriented Middleware (MOM) [10]. While RPC standard behavior is blocking, MOM benefits from a straightforward implementation of queues for asynchronous communication, and quality of service on unreliable networks [11, 12]. Consequently, Message Oriented Middlewares tend to be preferred. The communication between analysis nodes and the user is better achieved with a specific protocol such as the Trace Server Protocol (TSP) [13]. This protocol is similar to the Langage Server Protocol (LSP) [14], 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 of encoding data using a binary format.

C. Critical Path Computation and Analysis

The critical path, also called the active path, corresponds to the longest non idle path between the elements of a network and represents the wait dependencies between the tasks. A thorough description of the main algorithm used to compute the critical path on trace files is provided by Giraldeau and Dagenais [1] and Montplaisir et al. [15].

The critical path algorithm has been executed on a wide range of applications in order to detect and explain performance issues or logic faults. Rezazadeh et al. [16] presents a use case of the critical path for multi-threaded applications. It has also been applied to Chromium by Ezzati-Jivan et al. [17] to detect and analyze the root causes of performance issues. The critical path is also used by Nemati et al. [18] in clusters of virtual machines to diagnose performance issues, without interacting directly with the kernel of the guest machines. The execution states extracted from the results of the active path have also been used by Fournier et al. [19] to detect latencies automatically.

Critical path methods in distributed systems require careful adaptations, as described by Nemati et al. [18]. 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. Critical path computation also requires sequentially processing every trace file on a single analysis node, which is problematic for numerous large traces. Schulz [20] 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 algorithm from Giraldeau and Dagenais [1] has been tested [2], and several performance issues have been identified. As a result, this article proposes an improved distributed version of the algorithm.

D. Visualization Tools for Large Traces

A layout view can give a good overview of the system on a small scale, with one or two nodes. However, as the number of nodes increases, the analysis needs to scale in order to provide a readable picture to the user.

Moreover, even if tracing tools are able to collect information about the performance of the system and communications, the main issue is the ability to analyze such an amount of data. More often than not, visualization tools do not run on multiple nodes and, therefore, only scale vertically. As traces become larger, the user interface suffers from a lack of performance. In that case, the only solution with these tools is to reduce the amount of tracing data recorded, which impacts the ability to find the root causes of performance issues.

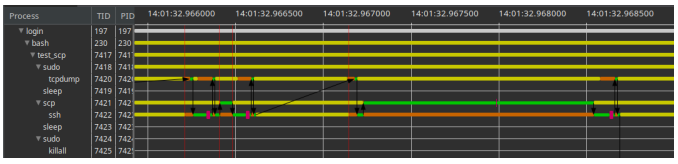

Fig. 1. An example of a time graph displayed in Trace Compass.

Time graphs are a popular visualization for trace analysis. As illustrated in Fig. 1, time is represented along the x-axis, and each horizontal line represents a resource that changes state over time (e.g., a thread, a core, a program). Usually, an event signals this state change. The period of time during which the state of a resource stays the same is called a state interval. On each row, rectangles represent state intervals, and a specific color represents each different state. Time graphs help keep track of the state of a system.

Trace Compass is integrated into the Eclipse IDE [9] and provides a critical path analysis module, as illustrated in Fig. 2.

When performing trace analysis, which is time-expensive, most tools require the users to process the trace on their own machine. This is usually performed multiple times, as each user involved in the system monitoring and development is trying to find performance defects or faults. In that case, a better approach would be to rely on a shared visualization service and use preprocessing of the data on tracing nodes. This way, fewer resources are needed to perform final computations and views.

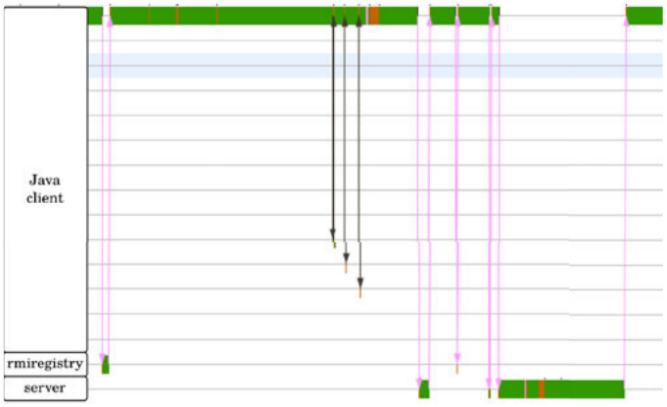

Fig. 2. The critical path view of a process as displayed in Trace Compass.

None of the current solutions offer a complete method for efficient trace analysis on large clusters. This article will propose solutions to solve these gaps. It is important to note that there are several approaches to parallelize the analysis of traces in Trace Compass [21]. The work of Martin [22] introduces a multi-threading algorithm for analysis. For a review of tracing tools for large distributed environments, the reader is referred to the survey of Las-Casas et al. [23]. Numerous projects are available to achieve this goal [24], most notably Zipkin [25] and Jaeger.

E. Index usage in Distributed Systems

Data locality is often the bottleneck in distributed and multi-threaded applications [26]. In our case, the traces are distributed among multiple nodes, making critical path computation more difficult. In order to balance the load on the different analysis nodes, an efficient method has to be applied to find file locations across different servers. In particular, we will focus on the use of an external communication index.

Data is often organized using a tree hierarchy. Moreover, striping data across many disks or data servers is common for data replication or system performance. Indexation, using meta-data associated with files or system elements, helps the organization and search of files [27, 28].

On the distributed file systems side, Vesta [29] and InterMezzo use pathname hashing for data allocation and location. [30] evaluates a metadata management system for in-memory space efficiency. Lazy hybrid (LH) is a method presented in [31] that uses hashing for metadata distribution across the metadata servers. It uses a Metadata Look-Up Table, a file replicated between the clients and metadata servers.

In many current parallel file system implementations, clusters are organized with several file servers on which the files are effectively stored, and one or more servers that store information about the files. This information is then used to build indexes and provide an efficient way to know the location of a file.

In distributed systems, trace files are generated locally and often stored on the same machine, or moved to several nodes dedicated to trace storage and analysis. Trace file storage is

a challenge, and most options that are available are used for micro services architectures [24].

As a result, a new method is needed in order to improve the indexation of trace files and improve the performance of distributed trace computation algorithms. Ultimately, a global overview of the tracing data available for the whole cluster should be available. Indexing is a good option to achieve that, as this method is widely used in distributed file systems.

III. Methodology to Optimise Critical Path Computation Using Index

This article presents a new distributed algorithm for the critical path computation. The focus is not on the critical path computation algorithm itself, but on the efficient communications between the distributed nodes, through the indexation of external communication events in trace files. In this section, we identify the challenges with the current methodology, and propose an improved method.

A. Requirements and Challenges

The goal of the critical path is to identify and link the dependencies between events collected simultaneously on different nodes. The first step to compute the critical path is to transfer the traces recorded on the different nodes to the analysis node, if necessary, where they must be synchronized due to the time difference between the system clocks. Then, the analysis begins by extracting the execution graph for each trace. At this point, the user needs to select a thread, or a section thereof that represents a query, on which to compute the critical path.

The execution graph of a trace is processed, with task states represented by horizontal edges, and signals represented by vertical edges, as illustrated in Figure 3. It is currently implemented by looking at kernel signals, scheduling events, and network calls of running processes. In the case of the Linux kernel, the `blocking` thread state can indicate a wait dependency on other processes, unlike the `running`, `preempted`, and `interrupted` thread states. 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. Interrupt requests allow determining the device causing the wait. As a result, thread states and interrupt requests are the minimal information required to determine the dependencies between processes and thus compute the critical path. Note that the critical path analysis may be applied to other operating systems, network interactions between virtual machines, as well as distributed software executions such as with the Message Passing Interface (MPI).

During the computation of the critical path of a thread, the local execution graph is scanned backward to replace all idle states, initiated by blocking events and ended by unblocking events, with the execution segments in other threads that created the unblocking events. For example, when a thread is waiting on a socket and is unblocked by receiving the expected data, the network events are responsible for the unblocking. The unblocking event origin, and the associated thread segment, should be identified in the trace of the remote node sending the data. An example of this would be a client, requesting a Web page. The client initiates the request and waits for the answer. The critical path computation is thus initiated for the client thread. The client thread is unblocked when receiving the answer from the remote Apache Web server. The Web server itself may be waiting for a while for information requested from a remote MySQL database. The critical path is composed of the client preparing the request, the Web server serving the request (while the client is blocked), the database replying to the request from the Web server (while the client and Web server are blocked), and finally the Web server sending the reply to the client and the client displaying the result.

In summary, the first step for computing the critical path is getting the execution graph from each trace, which can be easily parallelized if we initially do not follow the dependencies between traces. Then, the execution graph for the distributed system comprising multiple nodes can be obtained by stacking and connecting the graph from each node, after synchronizing the different traces to the same time reference Poirier et al. [3]. However, distributing the computation of the critical path is more challenging. Indeed, it requires finding the links between multiple trace events, and the parallelization of the longest path algorithm. Since no precomputation is possible, due to the trace data being distributed, the critical path cannot be computed without communication.

When analyzing a distributed system trace, the user will interactively navigate through the different threads and select a thread and time interval of interest (e.g. a request exhibiting an undue latency) for which computing the critical path. Precomputing all critical paths, for the whole duration of all threads, is therefore a waste of time. However, when preprocessing the traces, identifying all the wait dependencies and associated edges of the execution graph is worthwhile; all that remains is scanning the execution graph to identify the critical path for the thread and time interval specified by the user.

In our opinion, the three main contributions necessary to significantly improve the efficiency of the critical path computation, and enable analyzing large trace files, are (1) the efficient distribution of the critical path computation, (2) the distribution of traces, and (3) an efficient communication protocol to link the trace analysis nodes. The following section describes the work introduced to complete these milestones.

B. Current Methodology

Instead of performing the critical path computation on a single machine, Denys et al. [2] introduced a method to distribute the processing across multiple nodes. This section provides a brief overview of their algorithm and highlights the remaining limitations.

The original algorithm for computing the critical path synchronizes the time, computes the execution graph, and performs a backward iteration over every blocking state, to

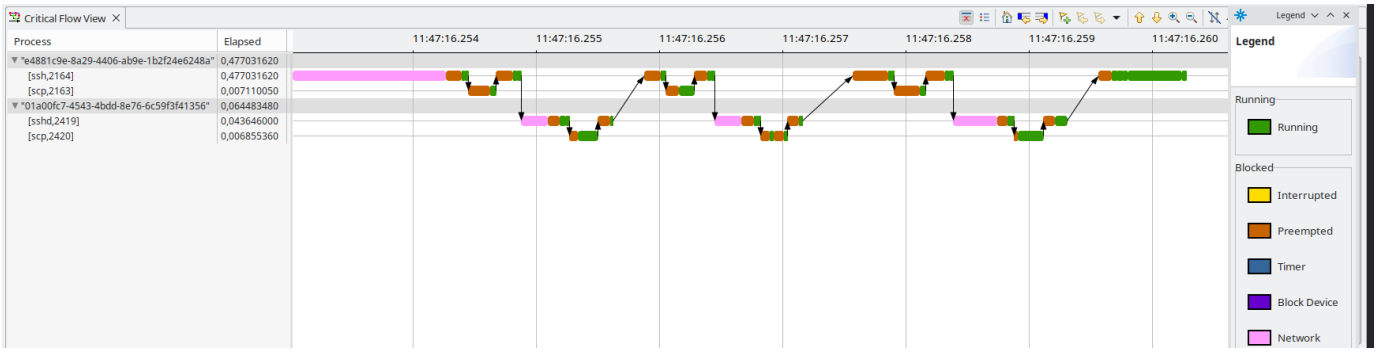

Fig. 3. The critical path as displayed in Trace Compass

look for their origin in other threads or host processes, as detailed by Giraldeau and Dagenais [1]. However, the original algorithm presents two notable drawbacks: the large memory footprint due to the in-memory computation of the complete execution graph, and the poor scalability due to the centralized nature of the computation. First, the large memory footprint was solved previously by storing the graph on disk. Then, Denys et al. [2] introduced an improved methodology based on a distributed computation of the critical path, and a new data structure for the communication between nodes.

The distributed algorithm precomputes the execution graph on each node, and synchronizes the time. Then, when the user selects a thread interval on which to compute the critical path, the execution graph is parsed, and remote dependencies are stored in a list. After completion, this list is broadcasted to all analysis nodes. Next, every node will try to resolve the remote dependencies, with their own precomputed execution graph, by matching events using relevant identifiers, such as TCP port and sequence numbers. Finally, a list of graph elements containing states corresponding to remote dependencies is sent back to the main node. The original execution graph and the remote events are then merged together. The result displayed to the user is an uninterrupted event sequence which is the critical path related to the initially selected thread interval.

The distributed algorithm enables the precomputation of trace files, thus avoiding to transfer complete trace files between nodes. However, several challenges have been encountered while using this method on large distributed traces, as identified by Denys et al. [2]. Firstly, it is currently impossible to know whether a trace located on a remote host contains some events related to the currently analyzed thread. As a result, the execution graph must be processed for each remote trace. Furthermore, this method requires broadcasting the remote dependencies list to each node. As a consequence, the whole network is loaded, and all analysis nodes are busy during those computations. Ideally, only a small number of analysis nodes would be involved in a critical path computation, leaving the other nodes available to perform other tasks, like answer other trace analysis requests from other developers.

C. Improved Methodology

This section introduces new algorithms for (1) the indexation of trace files using an *external communication index* (ECI) and (2) the use of preprocessing for the critical path computation optimization.

1) External Communication Index Computation: This first step aims to create an ECI for each trace file, located on analysis nodes, based on metrics about the traces, such as the number of tracepoints.

Efficient methods such as PTP (*Precision Time Protocol*) are available to synchronize traces located on different nodes while tracing and Poirier et al. [3] devised an algorithm to synchronize traces offline. Moreover, time synchronization is not necessary on systems based on the same clock, such as containers, as well as systems accurately synchronized, depending on the granularity level of events. We will not dwell on this aspect, as it is not a limitation, and traces stored on every node are considered time synchronized in all experiments in this paper.

When the preprocessing is triggered by a user request or a script, the system calls and kernel signals are parsed, in each trace file located on analysis nodes. Note that the traces have either been generated in place, or previously transferred to analysis nodes from other traced hosts. Specifically, during the trace preprocessing, system calls and signals such as `irq_handler_entry`, `irq_handler_exit`, `irq_softirq_entry` `irq_softirq_exit` and `irq_softirq_raise` are parsed, and `blocking` thread states are targeted. For events related to network communications, the algorithm targets events such as `inet_socket_local_in` and `inet_socket_local_out`, or `net_dev_queue` and `net_if_receive_skb`.

These signals allow identifying the remote nodes involved in the network exchanges. A loop iterates over every blocking state and collects information about the remote node involved. After reduction, an ECI is produced, containing metrics about the trace in the header, and a list of node identifiers, involved in network exchanges, found in the trace file.

```
{
  ...
  {nodeId : "0xc0a81e02", traceFile : "8949844
    vdfd98", nodeLocation : "0xc0a81e19"},
  {nodeId : "0xc0a81e04", traceFile : "8949844
    vdfd98", nodeLocation : "0xc0a81e19"}
  ...
}
```

Listing 1. Example of external communication index entries

This new method is inspired by the lazy hybrid method and its Metadata Look-Up Table [31]. Thanks to this index, it it then possible to know in advance if a trace contains network exchanges with a particular remote node. Other information is also usable for a better file distribution on nodes, which may be the subject of future work.

2) External Communication Index Usage for an Optimised Critical Path Computation: In a cluster of multiple analysis nodes, on which multiple trace files have already been preprocessed, and the ECI files have already been generated, the first challenge is related to the communication index propagation between the analysis nodes.

Every time a trace file is open or closed on an analysis node, a data reduction process in the algorithm merges all the local ECIs into one. The result is a matrix on which each line is a combination of traced node identifier, trace file name and analysis node (where the file is stored) identifier.

Then, the propagation of data between the analysis nodes is needed. A file exchange is then triggered in order to share or update the ECI between each analysis node. A first method consists in sending a first request to the nodes to get the serial number of the ECI. The serial number is a combination of the current date of update and a unique identifier placed at the beginning of the ECI. If the serial number is the same as the copy stored locally, no further action is required. If the serial number is more recent, a request to get a new copy of the ECI is sent to the node. The second method consists in sending only one request to get the latest version of the ECI each time.

As the file size of the ECI is typically negligible, it adds more overhead to send two requests, one for the update verification and another for the file transfer. As a result, the second option is preferred, and the algorithm on the viewer node merges all the received files together. At this step, the viewer node has obtained the up to date information about the traced node identifiers for each trace, and their location on analysis nodes.

When the new proposed critical path computation method is triggered by a user, the backward iteration on the execution graph related to the current analysed trace begins. If a blocking event is encountered and cannot be solved locally, and is network related, the packet origin is searched in the local copy of the ECI. The identifier of the analysis node on which the related traces are stored is determined, using the ECI, as illustrated in Figure 5.

As a result, the viewer node is able to know which trace file on which analysis node contains the resolution of a blocking state located in the currently analyzed trace. Thus, the main

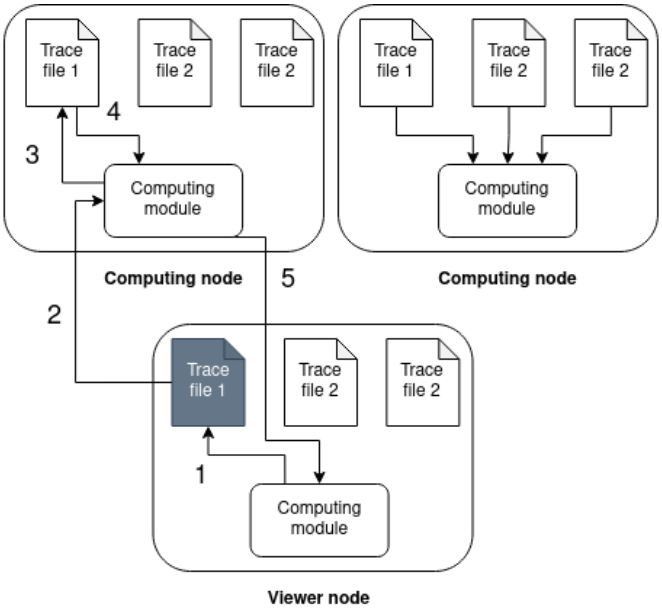

Fig. 4. The algorithm steps illustrated on the architecture elements.

node does not interrogate other analysis nodes with trace files without events related to the unblocking.

The previous distributed critical path computation algorithm was finding the origin of an unblocking (network event) in remote traces hosted on different nodes by sending messages to all nodes.

Rather than using a broadcast paradigm, we want to use a unicast or at least a multicast messaging system. Instead of sending a message to each remote host, we can send a message only to nodes with traces in which outgoing packets correspond to the identifier of the analysis node. It leads to faster critical path processing, and less load on the cluster of analysis nodes, as only nodes involved in the computation are contacted.

Furthermore, thanks to the ECI information, instead of displaying to the user that the blocking origin has not been found, it is now possible to notify the user that the origin may be on a server that is not online at the time of analysis. This allows a better indication of missing information.

IV. Results

This section demonstrates the effectiveness of the proposed method, on representative use cases, and shows a comparison with two earlier methods. The results of the two benchmarks detailed hereafter, and the source code of the experiments, are available in a GitHub repository[2].

Let us introduce this part with a simplified theoretical use-case, based on a large cluster with 20 nodes communicating together. Suppose that a thread `a` in node A sent a message to two other threads `b` and `c` in nodes B and C, respectively. Furthermore, let us consider that traces are recorded and

[2]https://gitlab.com/pierrefrederick.denys/critical-path-distributed-with-preprocessing

Algorithm 1: The pseudo-code of the simplified algorithm to resolve the recursive remote dependencies in the viewer node using the ECI file.

```
Input: execution graph G, process P
Output: critical path CP
...
function resolveBlockedStates(G,optional P)
    BlockingStates[ ] ← new JSON data structure
    ReturnedEdges[ ] ← new JSON data structure
    ...
    forall BlockingStates[] as BlockingState // Identify the node which causes blocking state
     do
        currentNodeId = getNodeIdFromSignals(BlockingState)
        if currentNodeId is in IndexCorrespondingNodes[ ] then
            IndexFileCorrespondingNodes[ ] ← currentNodeId
        else
            state ← UnresolvedNetwork
    forall IndexFileCorrespondingNodes[]          // Send blocking edges to each respective
     identified nodes
     do
        ReturnedEdges[ ] ← SolveBlockingStatesRemote(BlockingStates[ ], nodeId)
        CP ← ReturnedEdges[ ]
        DisplayCP(CP)                                   // Update CP view during processing
        resolveBlockedStates(CP)      // Recursively solve blocking states in returned paths
    return CP
...
```

stored locally, a trace server is running on each node, and the execution graph is preprocessed on each host. The user wants to obtain the critical path of thread `a`. This triggers the execution graph Analysis. Locally sourced blocking events are replaced by their cause found in other threads, and network blocking events are stored in a list. This list is sent to the 19 other nodes to find the origin. Each of the 19 other nodes triggers their execution graph Analysis, to find the origin of these blocked states. If the origin is found, the node replies with the sequence of events causing the blocking state. Therefore, apart from the 3 relevant nodes, A, B and C, 17 other nodes trigger their execution graph Analysis for nothing. This costly operation could be avoided if the nodes involved in each blocking were indexed during the trace analysis.

With the new proposed approach, only two requests are sent to nodes B and C, because A was aware that B and C had events related to blockings in trace A. As a result, only the minimum number of requests are sent, each time a user triggers a critical path analysis on a thread:

$$T(j) = \max_{j} \mathcal{F}(i,j) + \alpha \mathcal{C}(n), \tag{1}$$

where j is the number of nodes, α is the number of search iterations, $\mathcal{F}$ is the processing time, and $\mathcal{C}$ is the transfer time between nodes. We will now confirm this assertion on real usecases.

A. Test Architecture

For the two benchmarks presented hereafter, and in order to have a reasonable basis for comparison, the experiments are executed on the three algorithms for the critical path analysis: the conventional algorithm currently used in Trace Compass [1] (noted **AL1** in the results), the more recent distributed computation algorithm [2] (noted **AL2**), and finally, the new distributed computation algorithm proposed here, using ECI files, (noted **AL3**). Each experiment was repeated ten times for good accuracy, and the average is displayed along with the standard deviation. The cluster comprises 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. More details on the experimental setup are available in the repository of the project.

A first small and simple use case has been used in order to characterize the overhead and gain of the new algorithm. It can be viewed as good case for the new algorithm. Indeed, 20 remote file copies are issued in parallel, and the main script then waits for all the copies to complete. Therefore, there is a single *rendez-vous* and the critical path computation will only involve the node that finished last its copy. A more representative use case, based on an industry standard benchmark that executes real parallel scientific applications, will be presented after.

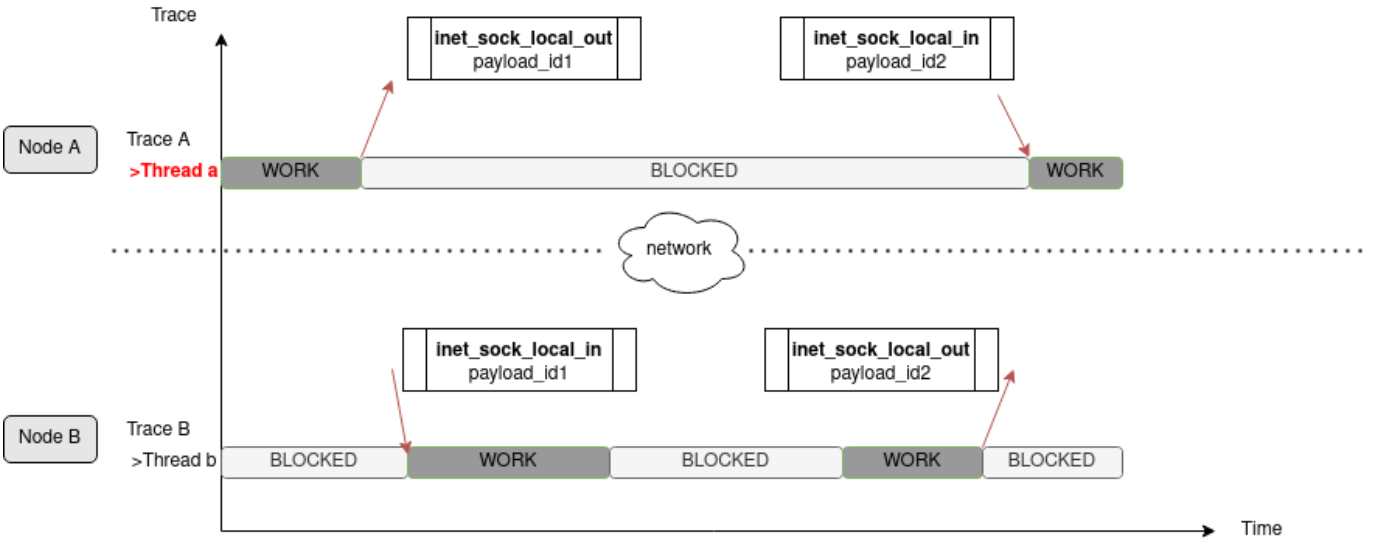

Fig. 5. The remote node matching with common ID in the payload of events.

B. SCP usecase : Performance and Overhead

The algorithms were tested on traces generated during a parallel SCP file copy. The copy operations were run in parallel, from the main node to a variable number, from 1 to 20, of computing nodes. As a result, a maximum total number of 21 traces are generated. Even if the machines, on which the file is copied, are identical in terms of software and hardware, there is some variability on the time needed to complete the copy operation, because of network delays, background system tasks, and the variability of processes scheduling. The parallel copy script on the main node is waiting until all copies are completed, and thus eventually waits after the slower node on which the file is copied. In addition to the number of nodes, the other variable tested with this use case is the size of the file copied and, thus, the execution duration and the size of the trace.

1) Trace Files Characteristics: This part presents a characterization of the traces generated with the benchmark, and the processing time observed during the analysis with the different algorithms.

Table I details the trace files used for the experiments. Each line gives the size of the copied file and the number of nodes on which it is copied in parallel. The size of the trace file generated on the main node, and the average size of the trace file generated on each of the computing nodes, are given in columns 2 and 3. The standard deviation (noted SD) is based on the 10 executions for the main node, and on the average trace file size across the 10 executions and all the computing nodes for the computing node entry.

We can observe that the number of events increases, in the trace generated from the main node, when the number of computing nodes increases. The trace file size given in the second column is for each computation node, as their size is almost equal between nodes. For example, the last row of Table I relates to the case where a 10GB file is copied in parallel to 20 nodes, generating a total of 20 trace files of about 19.5GB, and one trace file of 5.2GB on the main node, for a total of more than 390GB of trace files to analyse. This was impossible to process with the conventional version of Trace Compass, due to memory limitations when the execution graph was stored in main memory, in the approach from Giraldeau and Dagenais [1]. Finally, the last columns contain the percentage of events, within traces, used to identify blocked states, caused by interrupt request events (`irq`) and remote dependencies (`net`), as explained in the methodology. As the size of the file transferred is the same for each repetition of the tests, the standard deviation for the proportion of network events is close to zero.

TABLE I
THE TRACE FILE USED FOR THE BENCHMARKS : AVERAGE TRACE SIZES AND EVENTS NUMBER

	File size	Trace file on each Computing node (SD)	Trace file on main node (SD)	Events Main	Events Dest	Events IRQ	Events Network
1	100 MB	28 MB (1.86)	61 MB (1.68)	1,026,089	2,205,019	9.1 %	1.0 %
	1 GB	181 MB (1.92)	277 MB (1.83)	6,757,000	9,981,329	8.9 %	1.6 %
	10 GB	1.8 GB (0.33)	3.8 GB (0.07)	69,414,305	124,240,618	9.5 %	1.5 %
2	100 MB	68 MB (1.93)	105 MB (2.12)	261,926	388,210	7.5 %	1.0 %
	1 GB	346 MB (1.18)	252 MB (1.56)	12,494,815	9,113,923	8.2 %	1.2 %
	10 GB	3.2 GB (0.12)	2.4 GB (0.56)	118,495,717	86,933,604	9.2 %	1.4 %
3	100 MB	71 MB (1.58)	75 MB (1.96)	2,644,924	2,793,396	8.4 %	1.1 %
	1 GB	443 MB (1.12)	266 MB (1.36)	15,984,498	9,604,204	9.2 %	1.3 %
	10 GB	4.4 GB (0.18)	2.4 GB (0.45)	158,781,607	88,181,239	8.6 %	1.7 %
10	100 MB	83 MB (1.69)	78 MB (1.85)	2,993,919	285,664	8.8 %	1.8 %
	1 GB	4.1 GB (0.09)	262 MB (0.66)	43,285,500	9,565,111	7.2 %	1.9 %
	10 GB	8.8 GB (0.06)	6.8 GB (0.05)	317,427,586	248,481,236	9.1 %	1.1 %
20	100 MB	120 MB (1.20)	71 MB (1.81)	3,205,774	269,877	7.8 %	1.4 %
	1 GB	9.1 GB (0.08)	271 MB (1.76)	334,953,487	9,854,156	9.4 %	1.2 %
	10 GB	19.5 GB (0.05)	5.2 GB (0.06)	698,340,689	222,199,310.2	8.7 %	1.9 %
					Average	8.6 %	1.4 %

According to Table I, kernel events related to interrupt requests (irq) represents less than 10%, and network events less than 2%. As a result, only 2% of the events need to be transferred for remote dependencies computing.

2) External Communication Index Computation Overhead: In this first benchmark, the computing time to display the full critical path is compared between each algorithm. The number of nodes on which the file is transmitted with scp process varies from 1 to 20. Table II gives the average total time

needed to display the entire critical path of a thread of interest. The columns *Mean Processing Time* gives the time needed to fully display the critical path for each algorithm. The processing time for AL2 and AL3 is given for the first execution (column "First") and subsequent executions (column "Sub"). Indeed, the ECI processing and execution graph generation is only done once, at the first critical path request, and remains available on disk thereafter. The subsequent requests are faster (provided that the same trace set is selected) as preprocessing is already done. The last column gives the overhead induced by the ECI file pre-computing in AL3, compared to AL2, for the first critical path request. It is computed as the time difference in the trace initial processing. The average overhead is around 2% of the trace indexation and synchronization time. It is important to note that this overhead is only applied at the first opening of the trace. Indeed, once the ECI file is computed, it is reused for further critical path analysis, as long as new trace files are not added or deleted. It is important to note that AL2 and AL3 are incremental algorithms; a first display is available to the user after a few seconds. The display is then updated incrementally, as the blocked states are replaced by the remote threads and their state, while remote dependencies are resolved.

TABLE II
THE EXTERNAL COMMUNICATION INDEX COMPUTATION OVERHEAD : MEAN TOTAL PROCESSING TIME OF EACH ALGORITHM.

Nodes	File size	Mean Processing Time (s) AL1	AL2		AL3		Time overhead (SD)
		First	First	Sub	First	Sub	
1	100MB	7.35	3.01	1.26	3.07	1.25	2.11% (0.10)
	1GB	43.6	15.7	1.96	16.0	2.01	2.02% (0.07)
	10GB	570	210	3.23	214	3.42	2.11% (0.05)
2	100MB	23.2	4.98	1.34	5.11	1.38	2.25% (0.23)
	1GB	69.2	19.4	2.01	19.3	2.08	2.03% (0.12)
	10GB	632	235	3.28	240	3.34	2.41% (0.06)
3	100MB	212	67.9	1.36	69.3	1.39	2.01% (0.18)
	1GB	687	210	2.59	214	2.21	2.17% (0.09)
	10GB	958	297	3.32	303	3.39	2.12% (0.06)
10	100MB	685	80.6	1.58	79.0	1.68	2.03% (0.19)
	1GB	9890	356	2.63	364	2.64	2.41% (0.12)
	10GB	NP	1,916	4.01	1,957	4.18	2.15% (0.08)
20	100MB	1360	75.2	1.62	76.6	1.65	2.14% (0.21)
	1GB	NP	38.3	2.67	38.8	2.69	2.06% (0.14)
	10GB	NP	3,780	3.89	3,867	3.81	2.32% (0.07)

Algorithm (AL1) refers to the conventional sequential algorithm, for which the full processing of the execution graph is needed, on a single analysis node, for displaying the critical path. That explains the impossibility to handle the larger traces and the "NP" mention in the table for *not possible*.

The new proposed algorithm AL3 does not really improve the elapsed time for computing the critical path, when the computing nodes are not doing anything else, as compared to AL2. However, by only sending remote requests to the remote nodes containing trace files affecting the critical path, a lot of computing resources are saved. This is measured both in terms of network bandwidth and CPU time usage, across the cluster of analysis nodes. The results are presented in the next two subsections.

3) Transferred Data Amount Improvement: The total amount of data exchanged over the network is compared for algorithms AL2 and AL3, in order to characterize the gain of the new algorithm. These results has been obtained with the network traffic capture capabilities of the Wireshark tool, and its "conversation" filter, to only measure the amount of data related to the critical path computing, and exclude unrelated packets.

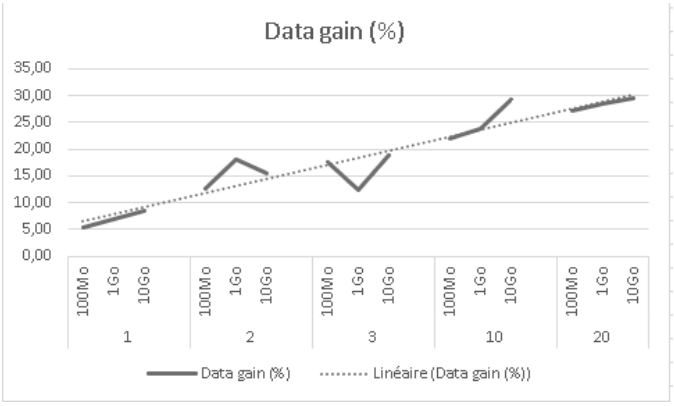

Fig. 6. The data gain on transferred data processing

We observed a gain in the amount of data transferred between nodes across the network when a critical path is computed. This is an average gain of 18% of the total transferred data, and the maximum gain observed for this benchmark is near 30%. We can see that the improvement increases with the number of traced nodes. This improvement seems limited because it takes into account the additional data exchanged between nodes on the first iteration. When a user needs to process several critical path analyses, the pre-computing is done only once, and this gain becomes even larger.

4) Computation CPU Time Gain: For this experiment, the CPU usage of the Trace Compass process has been measured on each computing node, during the critical path processing, in the previous experiment. Then, these results are added for both algorithms AL2 and AL3, with and without the ECI. Table 7 gives the percentage of improvement with the usage of the ECI, depending on the number of nodes in the traced cluster. The CPU load has been recorded with the `mpstat` tool, to give the average CPU load on each node.

$$\frac{\sum_{i=0}^{20} CPU_{load}}{20}$$

The main improvement in terms of processing time has been made with the distributed algorithm AL2, the gain in terms of processing time is variable with the ECI. If many nodes are involved in network communications, the computation node must send multiple requests, and the performance improvement is limited. If only two or three nodes are required to be queried for remote dependencies resolution, the gain in performance is substantial. As a result, the new method is particularly efficient for large traces and many nodes.

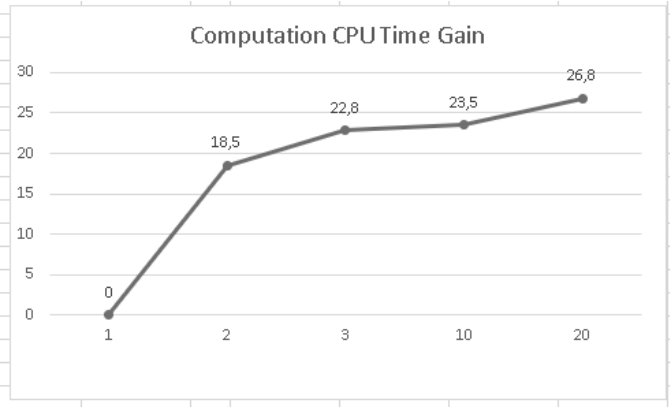

Fig. 7. The gain in terms of global CPU usage of the analysis nodes.

C. MPI usecase : Performance and Overhead

For this usecase, the industry standard SPEChpc™ 2021 benchmark [32] of the "Standard Performance Evaluation Corporation"(SPEC®) has been used. It comprises nine production HPC applications from various scientific domains and several programming languages. This test suite comes with useful benchmarking tools and is suitable for an extensive cluster size range from one to a few hundred nodes. This test suite is used to reproduce a near-to-real computation cluster running HPC workloads and using the Message Passing Interface (MPI) library. Each of the 20 computing nodes has a 6-core CPU. The experiments are run with 120 MPI ranks, one for each core of each node. More details about the environment are available in the public repository.

1) Trace Files Characterisatics: For this use case, four benchmark applications are used and run with different input sizes to generate different trace file sizes.

- *SOMA Offers Monte-Carlo Acceleration*: is a High-Performance Computing Monte-Carlo simulation for soft coarse-grained polymers. The variable load is the number of simulated polymers.
- *Tealeaf*: solves the linear heat conduction equation on a spatially decomposed regular grid. The variable load is the number of cells.
- *Minisweep*: is a Nuclear Engineering and radiation transport simulation. The variable load is the grid cell size.
- *SPH-EXA*: performs hydrodynamical and computational fluid dynamics simulations. The variable load is the power of 3 of the number of particles.

Table III presents the traces obtained during application executions. The parameters refers to the variable used to obtain a variation of runtime. More details about their signification are available in the repository. The most relevant informations are size of the generated trace files. The largest use case is Tealeaf with parameter 32768 which generated 20 times 3.94GB for the remote nodes and 83.2GB for the main node, for a total of 162GB of trace files to analyse to compute the critical path.

TABLE III
THE TRACE FILE USED FOR THE BENCHMARKS

Benchmark	Parameter	Exec. Time (s)	Trace size avg. per node	Trace size Total	Event Irq (%)	Event Network (%)
Soma	1400	6	7.5 MB (3.52)	158 MB (1.21)	12.3	1.08
	14000	21	29 MB (1.82)	613 MB (1.38)	12.9	1.14
	140000	261	256 MB (1.42)	5.4 GB (1.24)	13.1	1.85
	1400000	2355	2.83 GB (1.02)	63.8 GB (0.88)	13.6	1.90
Tealeaf	512	61	73MB (2.12)	1.5 GB (1.28)	13.6	1.91
	1024	78	93 MB (1.21)	1.9 GB (1.02)	12.6	1.92
	4096	411	494 MB (1.52)	10.45 GB (0.56)	13.2	1.34
	32768	3120	3.94 GB (0.83)	83.2 GB (0.08)	12.9	1.41
Minisweep	32	75	90 MB (2.04)	1.9 GB (1.21)	13.6	1.75
	64	150	182 MB (1.84)	3.85 GB (0.81)	13.6	1.28
	128	315	379 MB (1.62)	20.3 GB (0.93)	12.9	1.71
	768	1890	2.27 GB (1.13)	48.2 GB (0.38)	12.3	1.05
SPH-EXA	20	246	296 MB (1.57)	6.26 GB (1.09)	13.6	1.00
	40	470	565 MB (1.84)	11.9 GB (1.28)	12.8	1.02
	60	705	846 MB (1.92)	17.9 GB (0.52)	13.7	1.13
	120	1410	1.69 GB (1.07)	36 GB (0.31)	13.0	1.54

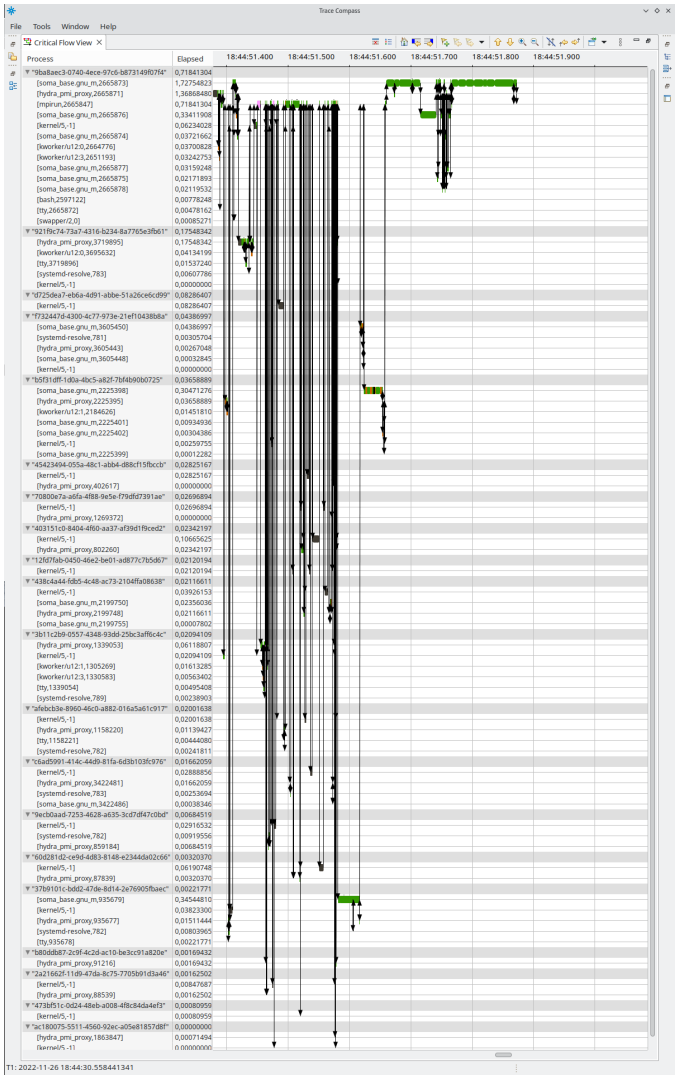

Fig. 8. Example of global critical path view for Soma benchmark

2) External Communication Index Computation Overhead: The table IV gives the global average overhead on computation time observed for several trace files sizes and different traced programs using MPI.

Although the elapsed processing time is slightly higher with AL3 as compared to AL2, there is no significant impact on

TABLE IV
THE OVERHEAD OF EXTERNAL COMMUNICATION INDEX COMPUTATION USAGE

Benchmark	Parameter	Mean Processing Time (s) AL1 First	AL2 First	AL2 Sub	AL3 First	AL3 Sub	Time overhead (SD)
Soma	1400	143	68.3	1.08	69.5	1.16	1.8 (0.12)
	14000	501	203	1.59	206	1.68	1.6 (0.15)
	140000	6223	1244	2.58	1267	2.79	1.9 (0.11)
	1400000	NA	2807	3.11	2857	3.30	1.8 (0.09)
Tealeaf	512	1455	715	1.63	729	1.69	1.9 (0.16)
	1024	1860	695	1.68	711	1.73	2.3 (0.17)
	4096	9800	1960	2.54	1999	2.68	2.0 (0.14)
	32768	NA	3719	3.68	3787	3.74	1.8 (0.08)
Minisweep	32	1788	894	1.72	912	1.75	2.0 (0.15)
	64	3577	1192	1.88	1213	1.98	1.7 (0.16)
	128	7511	2504	3.12	2555	3.21	2.1 (0.19)
	768	NA	2253	3.19	2295	3.28	1.9 (0.12)
SPH-EXA	20	5866	2704	2.06	2757	2.16	1.9 (0.06)
	40	11207	3735	3.39	3825	3.48	2.4 (0.10)
	60	16810	3362	3.59	3443	3.68	2.4 (0.14)
	120	NA	4689	3.17	4779	3.29	1.9 (0.13)

the user experience, as the display of the critical path is incremental, being updated during processing. The first display of the critical path begins a few seconds after being requested. The times in the table are the total elapsed time to obtain the full critical path graph.

3) Computation CPU Time Gain: The graph IV-C3 shows the global average CPU load reduction, with AL3 as compared to AL2, observed across the computing nodes. We can see that

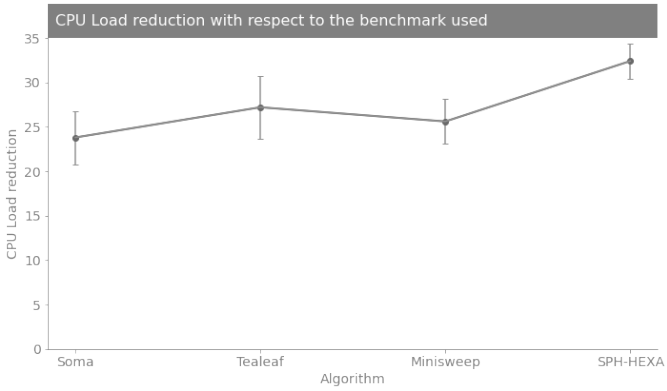

Fig. 9. CPU load reduction with respect to the benchmark used

the CPU load reduction is around 25% on average with AL3. The gain in terms of CPU load is appreciable and limits the load on the computing nodes.

D. Discussion

In real-world use cases, such as MPI programs on HPC clusters, calculations are performed by a large number of nodes with the same architecture and resources. The challenge is to properly distribute the computations over the massively parallel cluster. Bottlenecks are created when the load is not perfectly balanced among all the nodes. This appears as nodes being blocked by synchronization barriers, where the computation cannot proceed and some nodes are blocked until all the nodes have finished. In such cases, the slowest node is the bottleneck in the system, and a blocking states will be found in the other nodes, especially in the coordinator node in charge of distributing the computation. Tracing helps to identify the slowest node and, ultimately, the cause of this slowdown. The MPI benchmarks presented in this article are representative of real usecases and give an overview of the ability of the new critical path distributed method to handle large traces, while computing the critical path in a reasonable amount of time.

The results show a 2% overhead on the computation time for executing the first request for the critical path. This time includes the preprocessing of traces, the ECI file computation, and building the execution graph. However, there is no significant overhead on subsequent requests. This is advantageous in the case where multiple users work on the same set of traces. The traces preprocessing is only needed once and subsequent requests by the same user of other users can proceed very quickly. For this usecase as well, the new proposed algorithm AL3 brings a 25% reduction in data transmissions and a 27% CPU time gain observed for a cluster of 20 nodes.

V. CONCLUSION

In conclusion, the new algorithm presented in this article allows a more targeted and efficient distribution of the critical path computation, especially for large distributed systems with many traced nodes and analysis nodes. This method based on indexing the remote dependencies for the blockings identified in each trace file, to avoid useless computations and communications between nodes. It leads to a better response time when the critical path analysis is triggered. It is another step for adapting the current trace analysis tools to very large distributed systems analysis.

In future work, three aspects will be examined for possible optimisations. First, it would be interesting to study the trace files placement between nodes, and propose a technique to efficiently and automatically migrate trace files to balance the load and minimize the communications. Notably, such an improvement would add elasticity to the system and improve the resources usage. Second, the performance of a distributed analysis system depends on a reliable communication pattern. Therefore, a more efficient communication protocol and messaging system between nodes would further optimize the proposed method. It would be relevant to consider tests over larger clusters. Finally, the proposed technique can be extended to other distributed analyses, to improve their efficiency by speeding up information retrieval.

VI. 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] 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.

[2] P.-F. Denys, Q. Fournier, and M. R. Dagenais, "Distributed computation of the critical path from execution traces," 2022.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] M. Rezazadeh, N. Ezzati-Jivan, S. V. Azhari, and M. R. Dagenais, "Performance evaluation of complex multi-thread applications through execution path analysis," *Performance Evaluation*, vol. 155-156, p. 102289, Jun. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531622000049

[17] N. Ezzati-Jivan, H. Daoud, and M. R. Dagenais, "Debugging of Performance Degradation in Distributed Requests Handling Using Multilevel Trace Analysis," *Wireless Communications and Mobile Computing*, vol. 2021, p. e8478076, Nov. 2021, publisher: Hindawi. [Online]. Available: https://www.hindawi.com/journals/wcmc/2021/8478076/

[18] 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.

[19] Q. Fournier, N. Ezzati-jivan, D. Aloise, and M. R. Dagenais, "Automatic Cause Detection of Performance Problems in Web Applications," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2019, pp. 398–405.

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

[21] F. Reumont-Locke, "Méthodes efficaces de parallélisation de l'analyse de traces noyau," Master's thesis, École Polytechnique de Montréal, Aug. 2015. [Online]. Available: https://publications.polymtl.ca/1899/

[22] M. Martin, "Analyse détaillée de trace en dépit d'événements manquants," Master's thesis, École Polytechnique de Montréal, Aug. 2018. [Online]. Available: https://publications.polymtl.ca/3248/

[23] 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

[24] L. Cheng, Y. Wang, Q. Liu, D. H. Epema, C. Liu, Y. Mao, and J. Murphy, "Network-aware locality scheduling for distributed data operators in data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1494–1510, 2021.

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

[26] 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, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/16/5560

[27] J. Smith and P. Schirling, "Metadata standards roundup," *IEEE MultiMedia*, vol. 13, no. 2, pp. 84–88, Apr. 2006, conference Name: IEEE MultiMedia.

[28] A. Devulapalli and P. Wyckoff, "File Creation Strategies in a Distributed Metadata File System," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–10, iSSN: 1530-2075.

[29] P. F. Corbett and D. G. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems*, vol. 14, no. 3, pp. 225–264, Aug. 1996. [Online]. Available: https://doi.org/10.1145/233557.233558

[30] N. Y. Song, H. Kim, H. Han, and H. Y. Yeom, "Optimizing of metadata management in large-scale file systems," *Cluster Computing*, vol. 21, no. 4, pp. 1865–1879, Dec. 2018. [Online]. Available: https://doi.org/10.1007/s10586-018-2814-7

[31] S. Brandt, E. Miller, D. Long, and L. Xue, "Efficient metadata management in large distributed storage systems," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, Apr. 2003, pp. 290–298.

[32] J. Li, A. Bobyr, S. Boehm, W. Brantley, H. Brunst, A. Cavelan, S. Chandrasekaran, J. Cheng, F. M. Ciorba, M. Colgrove, T. Curtis, C. Daley, M. Ferrato, M. G. de Souza, N. Hagerty, R. Henschel, G. Juckeland, J. Kelling, K. Li, R. Lieberman, K. McMahon, E. Melnichenko, M. A. Neggaz, H. Ono, C. Ponder, D. Raddatz, S. Schueller, R. Searles, F. Vasilev, V. M. Vergara, B. Wang, B. Wesarg, S. Wienke, and M. Zavala, "SPEChpc 2021 Benchmark Suites for Modern HPC Systems," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. Bejing China: ACM, Jul. 2022, pp. 15–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3491204.3527498