ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 154

Systems Group, Department of Computer Science, ETH Zurich

Real-Time Performance Analysis of a Modern Data-Parallel Stream Processing Engine

by

Ralf Sager

Supervised by

Prof. Dr. Timothy Roscoe
Dr. John Liagouris
Dr. Desislava Dimitrova

March 2016–September 2016

inf | Informatik
Computer Science

Abstract

Identifying performance bottlenecks of modern, distributed stream processing engines is a serious challenge. At the same time, such engines are widely used to perform data-parallel tasks such as machine learning, graph processing or sophisticated streaming data analysis. Oftentimes, these tasks are required to produce low-latency results as well as to achieve a high throughput. As computations often consist of complex, iterative dataflows and are distributed over multiple physical machines — with hundreds of worker threads in total — finding the source of a performance problem is a difficult task. While profiling can be used to quantify the time spent in the various steps of a parallel computation, it does not take into account the dependencies between the steps. As a result, optimization efforts are often wasted on components that have little to no influence on query latency or the overall runtime of a program.

In this work, we offer a more effective alternative by applying critical path analysis, a dependency-aware technique. The critical path is defined as the longest sequence of dependent steps in a parallel program's execution. Any increase in the execution time of a step on the critical path will therefore result in an equal increase in the total runtime of the computation. We refine existing critical path-based models and apply them to data-parallel systems, which often share common low-level principles. We provide guidelines on the instrumentation necessary to apply our model, as well as a set of trace properties that help verify the correctness of that instrumentation. Furthermore, we develop a novel method to identify phases in a worker thread's execution during which it is waiting — e.g. for a message from a different worker — even in the absence of blocking system calls. Through critical path analysis, we can then identify performance bottlenecks in system components, dataflow operators as well as in network communication.

To demonstrate our ideas, we implemented a prototype system capable of performing a critical path analysis of the Timely Dataflow stream processing engine. We show that our system can effectively identify the factors limiting a data-parallel computation's overall performance. Furthermore, we demonstrate that our analysis is both efficient and scalable, and can even be performed in real-time in certain configurations.

Contents

Chapter 1

Introduction

Modern distributed stream processing engines such as Storm [37, 2], Flink [1, 21] or Timely Dataflow/Naiad [32, 6] hide much of the complexity of a distributed system's implementation from the application developers. They enable the rapid development of highly scalable stream processing systems for data-parallel workloads that offer both low latency as well as high throughput. Alongside batch-oriented frameworks like Spark [40], they are often used to perform large-scale data analytics.

As much as these frameworks facilitate the development of complex data-parallel systems, identifying the causes of performance issues and selecting optimization candidates are still difficult tasks, just as they are for other large-scale distributed systems. One well-studied, intuitive method of modeling the performance of parallel- and/or distributed systems is *critical path analysis* [39].

In this work, we applied critical path analysis to modern data-parallel systems. They key contributions of this work are:

- We introduce a unified mathematical model for analyzing the performance of data-parallel systems which is applicable to systems with different computational models. Our model is based on existing trace-based critical path models.

  We describe a small set of program activities which are typically performed by most modern data-parallel systems and can be traced with a low amount of instrumentation and little performance overhead. Furthermore, we describe a way of performing critical path analysis for partial traces (slices), which permits the analysis of continuously running computations or of only specific parts of a computation (e.g. query executions).

The formal model is general enough that it can be applied to many data-parallel systems, including batch processors such as Spark. We demonstrate the use of our model by applying it to Timely Dataflow [32, 6].

- We define a set of properties which the instrumentation must satisfy so that our critical path model is well-defined. These properties can be checked efficiently based on the collected execution traces.
- We introduce a method to identify phases during a program's execution in which it is waiting for messages or input data. Our method is applicable when the program does not use blocking system calls, but is busy-waiting (i.e. spinning) instead. The waiting phases, which need to be known in order to perform a critical path analysis, are identified solely based on the collected execution trace. Our method is targeted at the Timely Dataflow system [32, 6], but the general principles could also be applied to similar systems.
- We implemented a prototype system capable of performing critical path analysis of Timely Dataflow [32, 6] computations. The prototype takes the raw instrumentation logs as input and is capable of performing all necessary steps in real-time in certain configurations. As the most important stages of the performance analysis are parallelized, our prototype scales reasonably well.

In Chapter 2, we provide a brief introduction to Timely Dataflow's computational model as well the system itself. Also, we review the related work in this chapter. Chapter 3 introduces the generalized, formal performance model we use to perform critical path analysis. In Chapter 4 and Chapter 5, we describe how we applied the formal model to Timely Dataflow computations. Chapter 4 includes a detailed description of Timely Dataflow's runtime behavior, the instrumentation used, and of the preprocessing of the raw execution traces as preparation for the critical path analysis. Chapter 5 describes the identification of waiting phases, as well as the critical path computation itself. Chapter 6 contains a brief description of the implementation of our prototype performance analysis system and of our trace/critical path visualization tool. We evaluate both the usefulness of our prototype in finding performance issues as well as the performance of the prototype itself in Chapter 7. Chapter 8 describes a list of remaining issues and suggestions for future work. Finally, we draw our conclusions about this work in Chapter 9.

Chapter 2

Background

In this chapter, we will discuss the fundamentals of the Timely Dataflow/-Naiad system, to which we want to apply critical path analysis (Sections 2.1 and 2.2). Furthermore, we will go over the related work regarding critical path analysis in Section 2.3.

2.1 Computational Model of Timely Dataflow / Naiad

Naiad [32] is a high-performance distributed system for processing data-parallel workloads. Similar to stream processing engines like Storm [2, 37] and Flink [1, 21], it offers a form of transparent data-parallelism. While an application has to specify how the input data is partitioned among workers, the system hides most of the remaining complexity of a distributed system, e.g. the exchange of data messages, buffering, progress tracking and so on.

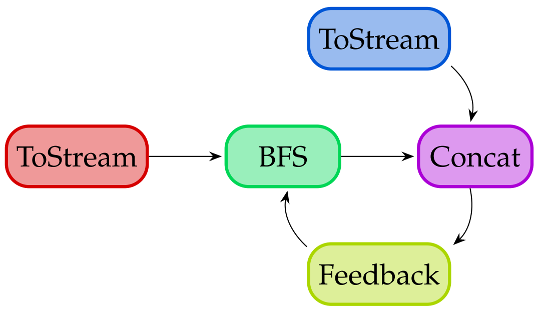
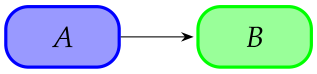


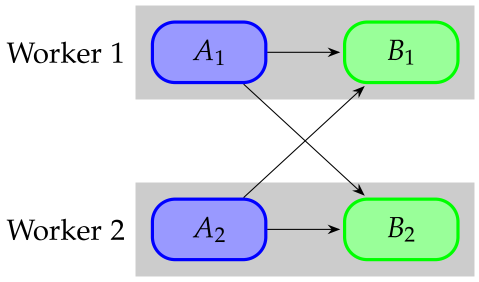
Figure 2.1: Dataflow graph of a simple breadth-first search (BFS) computation. This example computation is included in Timely Dataflow's distribution [6].

In Naiad's underlying computational model, called *Timely Dataflow*, computations are defined as a directed *dataflow graph*. Each vertex in the dataflow graph denotes an *operator*, and each edge denotes a communication *channel*. Operators receive messages from other operators along their incoming edges and can send messages along their outgoing edges. Furthermore, operators are also allowed to keep a local state. In Naiad, dataflow graphs are allowed to contain arbitrarily nested cycles, which makes the implementation of complex iterative algorithms straightforward. An example of a simple dataflow graph is shown in Fig. 2.1.

In order to provide data-parallelism, Naiad transforms the logical dataflow graph into a physical dataflow graph, which is distributed to multiple worker threads at runtime. Workers each have a copy of the whole graph. Additionally, for each edge (channel) in the logical dataflow graph, corresponding edges are created leading to the target operator on each worker in the system[1]. This expansion is shown in Fig. 2.2. A partitioning function associated with the channel defines the edge along which a particular message is sent.



(a) Logical dataflow graph.



(b) Physical dataflow graph.

Figure 2.2: Expansion of the logical dataflow graph into a physical dataflow graph distributed to two workers.

Naiad also includes a fully distributed progress tracking protocol, which measures the global progress of the whole computation. To do so, each data message is assigned a logical timestamp. An operator which receives a message with timestamp t is then granted the capability to send messages for any timestamp $t' \geq t$. The progress tracking protocol keeps track of outstanding messages and capabilities for each operator and therefore knows which timestamps could still be received by a particular operator. An operator can make use of Timely Dataflow's progress tracking by requesting a notification for a certain timestamp t. As soon as it is guaranteed that the operator will not receive any more messages for timestamp t, a notification will be delivered to the operator by the progress tracking logic. The notification also carries with it a capability for timestamp t, allowing the operator

[1]An exception to this are `Pipeline` channels, which only connect operators on the same worker and cannot be used to exchange data with operators on other workers.

to send any final messages for the particular epoch. For more details about how progress tracking works in Timely Dataflow, see [32, 28].

Note that this computational model can be seen as a specialization of the actor model [23]. Operators in the (physical) dataflow graph can only react to messages from other operators or progress notifications delivered by the system itself. They are allowed to keep a local state, but cannot share state with other operators.

The Timely Dataflow programming model, though relatively low-level, is also very general. This allows one to easily implement many higher-level programming models on top of it, with the added benefit of being able to combine the different programming models in a single computation instead of having to rely on separate, specialized systems for each purpose. An example of a higher-level computational model implemented on top of Timely Dataflow is *Differential Dataflow* [29, 8], a model which allows the processing of continuously changing input data in an incremental way. Since this work is focused on the underlying Timely Dataflow layer, the performance analysis methods discussed can also be applied to any higher-level libraries built on top of it, such as (for example) Differential Dataflow.

It has been shown that Naiad/Timely Dataflow is very efficient, provides a high throughput and delivers results with very low latency [32].

2.2 The Timely Dataflow System

This work focuses on a newer implementation of the Timely Dataflow model which is also called Timely Dataflow [6]. This prototype is functionally identical to Naiad apart from a small number of extensions and improvements. It is implemented in the Rust programming language [5]. Hereafter, "Timely Dataflow" (or "Timely" for short) refers to this prototype and the behavior thereof, not the original implementation (Naiad) nor the theoretical model. A prototype of Differential Dataflow also exists [4].

Listing 2.1 shows how a typical Timely Dataflow operator is implemented. It displays an operator "`WordCount`" which receives words on its input channel and counts how often the same word occurs within each epoch. When it receives a notification indicating that a particular epoch ended (i.e. when no more messages will be received for its timestamp), it will output all the words seen in that particular input batch, along with a count indicating how often a word was seen.

The unary (single in- and output) operator is defined by the `unary_notify` method. Its first parameter defines the partitioning function for the operator's input channel. In this case, a hash function is used, as this will ensure the same word will always be delivered to the same worker. The fourth

```
 1 let mut wordcounts_by_time = HashMap::new();
 2
 3 stream.unary_notify(
 4     Exchange::new(|x| hash(x)),
 5     "WordCount",
 6     vec![],
 7     move |input, output, notificator| {
 8         input.for_each(|time, data| {
 9             let mut wcs = wordcounts_by_time.entry(time.time())
10                                 .or_insert(HashMap::new());
11             for word in data.drain(..) {
12                 *wcs.entry(word).or_insert(0) += 1;
13             }
14             notificator.notify_at(time);
15         });
16
17         notificator.for_each(|time, _num, _notify| {
18             if let Some(wcs) = wordcounts_by_time.remove(
19                                     &time.time()) {
20                 output.session(&time)
21                     .give_iterator(wcs.into_iter());
22                 }
23         });
24     });
```

Listing 2.1: Example of a simple Timely Dataflow operator (`WordCount`)

parameter of `unary_notify` is a closure which defines the actual behavior of the operator. This code is run each time the operator gets scheduled. In the case of our `WordCount` operator, each time it gets scheduled it first processes any potential new input messages (lines 8-15). For each batch of messages, it looks up the already stored word counts for the particular timestamp in the operator's local state (`wordcounts_by_time`). For each word in the batch, it then updates its associated word count. Finally, a notification for the completion of the epoch is requested from Timely. Subsequently, the operator processes any notifications (lines 17-23) it received. For each notification, it removes the stored word counts corresponding to the notification timestamp (if any) from the local state and sends them through its output channel.

Many Timely operators follow a similar pattern as the `WordCount` operator. However, as the example shows, the operators have great freedoms, which means they cannot be trusted to follow a specific pattern for our performance analysis. One important detail to note is that operators are not called when they received new messages or notifications, and there is no individual callback for either. Instead, they are *pulling* messages from their input queues. The operators are simply scheduled repeatedly, even if no messages or notifications are available, hence they are essentially polling the queues and notificators. For more information about Timely Dataflow's operator scheduling, see Section 4.1.1.

2.2.1 Scopes/Subgraphs

Dataflow graphs in Timely can also contain subgraphs, or scopes, which as a whole behave in a similar way as individual operators. For example, scopes are used to implement cycles in the dataflow graph. In such a case, timestamps inside the subgraph are extended by an additional coordinate indicating the loop iteration a message belongs to. The details are largely irrelevant for this work, however.

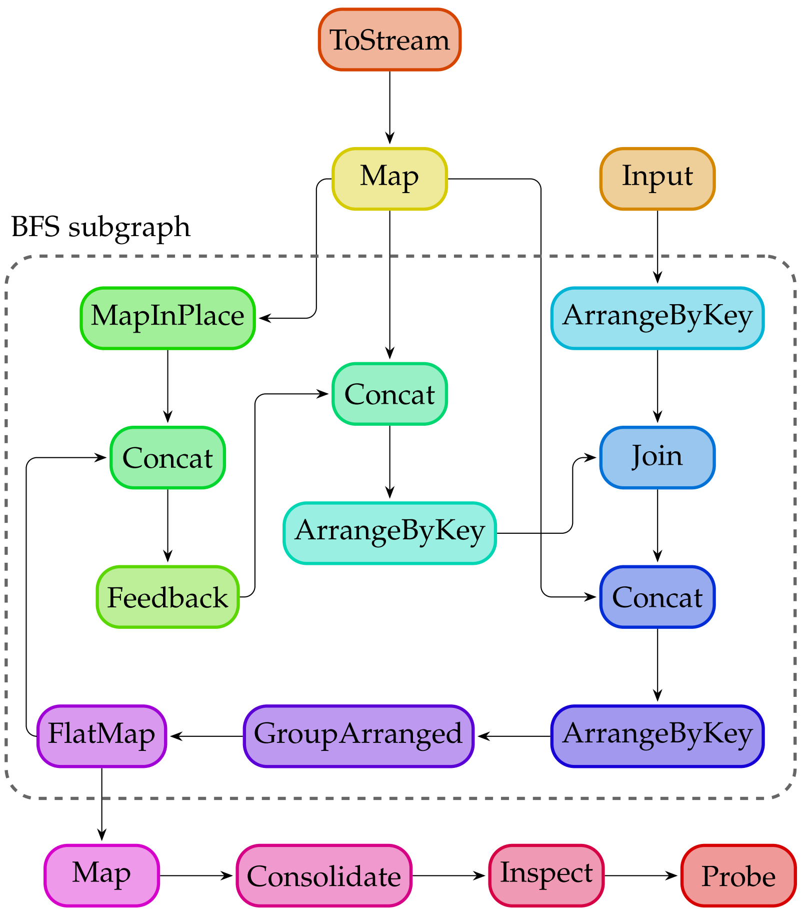


Figure 2.3: Dataflow graph of the Differential Dataflow-based breadth-first search (BFS) implementation. This computation is included in the Differential Dataflow distribution [4].

Figure 2.3 shows a Differential Dataflow-based BFS dataflow graph, which includes a subgraph. The outermost scope, which is not explicitly shown, is called the *root scope* and encompasses the whole dataflow graph.

Most of the example traces shown in this work are either based on the Differential Dataflow-based BFS implementation shown in Fig. 2.3, which is included in the distribution of Differential Dataflow [4], or the BFS implementation based on pure Timely Dataflow shown in Fig. 2.1 which is included in Timely Dataflow's distribution [6].

Timely Dataflow's runtime behavior, including the relevant parts of its scheduling, progress tracking logic and communication architecture is described in more detail in Section 4.1.

2.3 Related Work

Critical path analysis was first introduced by Kelley and Walker [26] in the context of project planning. In their model, the structure of a project is modeled as a directed graph, wherein the edges represent jobs (activities) and vertices mark the beginning or completion of the connected jobs. The edge weights denote the estimated duration of each job, and the structure of the graph models the dependencies between the jobs. The critical path is defined as the longest path on the graph with regard to the edge weights. It therefore dictates the total duration of the project. The concept of the critical path was later applied to computer systems. Particularly, it is often used for the analysis of parallel- and distributed programs and systems [39, 30, 14, 9, 10, 19, 24, 25, 35, 17], particularly MPI programs [36, 13, 12, 16] as well as concurrent programming languages [34]. Furthermore, it has been applied in areas such as the design of asynchronous circuits [38], processor design [22] or the performance analysis of HTTP/TCP transactions [11].

The critical path model was first applied to the analysis of parallel and distributed computer programs by Yang and Miller [39]. They created the notion of the "Program Activity Graph" (PAG), in which the edges represent the activities of a computer program instead of the jobs in a project. The activities can use any combination of resources, e.g. CPUs, I/O devices or networks. Instead of using critical path analysis to make predictions about certain characteristics of a project (e.g. its estimated time of completion) based on a priori knowledge/estimates about the duration and dependencies of jobs, they used it for post-mortem performance analysis of a program. The PAG is thus constructed from the programs execution history (trace) collected by appropriate instrumentation. To compute the critical path, they used two different longest-path algorithms (adapted from their shortest-path counterparts). The formal model presented in Chapter 3, like many other models used in this field, is heavily based on the original PAG model.

In [30], Miller et al. describe a comprehensive performance measurement system which further develops the techniques from [39], especially by extending the instrumentation to multiple layers of the software in order to increase precision.

Broberg, Lundberg and Grahn [14] introduced an extended critical path analysis for multithreaded programs, which also deals with scenarios in which there are more threads than processors (or cores).

Alexander et al. [9, 10] describe a number of different algorithms to compute the k longest (near-critical) paths. This is useful because in any realistic PAGs, there are usually a number of paths which are of a length almost equal to the length of the critical path. Slightly improving the runtime of an activity on the critical path might therefore result in another path of similar length becoming critical. Therefore, knowing the longest path alone is not always sufficient to assess the actual optimization possibilities of a particular program. However, in data-parallel computations we expect the number of near-critical paths to grow exponentially with the length of an execution trace. As their best algorithms to compute the k longest paths have a time complexity of at least $\mathcal{O}(ke)$, where e is the number of edges in the PAG, they will not scale for an exponentially growing k. Limiting the number of computed paths to a small, fixed number is possible but also significantly reduces the usefulness of the near-critical path concept.

Hollingsworth [24, 25] introduced a highly specialized online algorithm which computes the critical path during the program's execution, without needing to collect an execution trace. The algorithm works by piggybacking instrumentation data onto data messages exchanged by the processes during the execution. Each process also keeps a state which describes the longest path ending in the particular process at any given time. Whenever a process receives a message, it compares the length of the longest path of the remote process (piggybacked onto the message) to the length of the longest path ending at the local process, and updates the local state to represent the maximum of the two. Depending on the number of messages exchanged, this approach can have a quite low overhead. It can also produce intermediate results during the computation. However, since no trace is collected, it is difficult to extend this approach to compute further metrics (e.g. the slack of an activity) or to perform additional offline/online performance analysis of the reference program.

Saidi et al. [35] apply a low-overhead approach similar to Hollingsworth's in order to perform a critical path analysis of a complete system, i.e. including both user- and kernel-space software components as well as hardware devices.

Dooley and Kale [19] discuss a slightly more sophisticated variant of online critical path detection for message-passing parallel programs. Furthermore, they discuss a number of uses for the computed critical path profiles. Especially, they describe how to use the critical path information computed by their online algorithm for automatic performance tuning by adjusting task scheduling priorities accordingly.

Oyama et al. [34] describe a scheme to automatically instrument programs written in high-level parallel languages in order to compute the critical path online, also using a scheme similar to Hollingsworth's.

Schulz [36] introduced a novel algorithm for finding the critical path of an MPI application. It is based on the simple observation that the time a processor spent waiting (i.e. blocked) for an MPI message can never be part of the critical path. The algorithm starts at the end of the trace, and then directly backtracks along the PAG edges which belong to the critical path; it does not need to process any vertices in the PAG which are not part of the critical path. Therefore, the algorithm is very efficient and scales very well as the number of workers/processors of the reference computation is increased, as its runtime mostly depends on the number of edges of the critical path, not on the number of edges/nodes in the complete PAG. Hence, this algorithm was selected as the basis for the critical path algorithm described in this work (see Chapter 3).

Böhme et al. [13, 12] implemented the backtracking algorithm efficiently and also defined a number of performance indicators that help characterizing bottlenecks in MPI programs.

One limitation of critical path analysis is that it is based on the *runtime* of the activities in the program activity graph. Since the runtime of an activity can vary depending on the underlying hardware (and in fact, the runtimes of different activities can vary to a different degree), the portability of critical path analysis is limited. Chen and Clapp [16] address this issue by introducing *critical path candidates*, which are potential critical paths based on their instruction- and communication counts instead of their runtime. Critical path candidates thus capture intrinsic properties of the program and are independent of the microarchitecture of the machines used to run the program. Consequently, their framework also allows them to predict what effects changes in the underlying hardware architecture have on the critical path.

A common problem which occurs when one tries to apply critical path analysis to complex, heterogeneous systems is that the causal relationships between activities are not always known. This is usually solved by applying expert knowledge about the system, which is impractical for large and complex systems. In the context of Internet services, the *Mystery Machine* developed by Chow et al. [17] is able to automatically infer such relationships based on a large amount of pre-existing log data collected from past requests to the service. The calculation of this causal model is parallelized as a Hadoop job. Based on the trace data and the inferred model of the system, the Mystery Machine can then compute the critical paths as well as the slack of particular activities (called segments in their work).

In the context of data-parallel systems (specifically Spark [40]), Ousterhout et al. [33] employ *blocked time analysis*, which is a what-if analysis used to quantify the performance improvement typical data-parallel workloads would achieve under the premise that they would never block on a resource (e.g. a disk or a network device). The results of their work suggests that contrary to widely-held assumptions, such computations are often bottlenecked on the CPU rather than I/O operations. In such cases, critical path analysis could help to further find the specific (computation-) activities which, once optimized, would improve end-to-end task completion time the most. Since the kind of instrumentation needed for blocked time analysis is very similar to the kind this work uses for critical path computation, we argue that future work should focus on combining these two approaches into a comprehensive performance analytics framework for data-parallel systems (for more information about future work, see Chapter 8).

Morton et al. [31] used the critical path concept to provide progress estimates for long-running queries which consist of directed acyclic graphs of MapReduce jobs. Rather than computing the critical path based on execution traces, it is computed based on individual task duration estimates, which allows the *prediction* of the total job completion time.

Chapter 3

Formal Performance Model

The critical path of a distributed program execution is commonly defined on a particular execution's Program Activity Graph (PAG) [39, 30, 9, 10]. Each activity of the program (e.g. the execution of a job or the transmission of data) defines an edge in the PAG, with its weight being equal to the duration of the activity. The structure of the PAG defines the precedence relationship between the activities. The critical path is then simply defined as the longest path in the PAG. Thus, any performance improvement of an activity on the critical path will then result in an equal improvement in the total runtime of the program, assuming the critical path does otherwise not change as a result of the optimization.

In this chapter, we describe an adapted, more detailed version of this model for real-time, trace-based critical path analysis of data-parallel systems. Section 3.1 describes the basics of our model. Section 3.2 describes the types of activities we consider in detail. In Section 3.3 we discuss additional properties of our model which are both helpful to check the correctness of a system's instrumentation as well as for the critical path computation. Finally, Section 3.4 describes an efficient algorithm to compute the critical path in our model.

3.1 Basic Performance Model

We start by precisely defining what we consider to be an activity in our model:

Definition 3.1 (Activity) *An activity is a logical operation performed at any level of the software and hardware stack. Each activity is associated with two timestamps* `[start,end]`, `start` $\leq$ `end`, *that denote the time its execution started and ended with respect to a global clock* $\mathcal{C}$*.*

An activity can be either an operation performed by a *worker* (worker activity) or a message transfer between a source- and destination worker (communication activity). Typically, worker activities are the execution of a collection of code instructions, but can also be I/O operations performed by the worker (i.e. reads/writes to external systems which are not modeled, e.g. mass storage devices, network interfaces, etc.). Communication activities are any interactions between workers, for example via a network or shared memory message passing.

Modern data-parallel systems consist of workers that perform activities, however, the term "worker" in the literature may refer to a node in a cluster, a virtual machine, a process or even a thread in a single multi-core machine. To provide a consistent terminology, workers in our model are defined as follows:

Definition 3.2 (Worker) *A worker is a logical execution unit that performs a series of activities sequentially, i.e. one activity after the other.*

Definition 3.2 simply states that the activities of the same worker cannot overlap in time. Given two activities of a worker, a_i:`[start`$_i$`,end`$_i$`]` and a_j:`[start`$_j$`,end`$_j$`]`, $i \neq j$, the following holds: `end`$_i \leq$ `start`$_j$ or `start`$_i \geq$ `end`$_j$.

At any point in time τ with respect to a global clock $\mathcal{C}$, a worker can be in one of three states: *unborn*, *running* or *terminated*. A worker is considered to be in the *unborn* state before it was started. After it was started, a worker is in the *running* state, and therefore performing a sequence of activities. After the worker has finished performing activities, it is in the *terminated* state. The only state transitions a worker can perform are from *unborn* to *running* and from *running* to *terminated*. Any worker which is not already in the *running* state at the beginning of the computation must be spawned by a different, already running worker. This needs to be reflected by a communication activity from the running worker to the newly spawned worker whose `end` timestamp marks the destination worker's transition to the *running* state.

Definition 3.3 (Activity Graph) *An activity graph $G = (V, E)$ is a directed labeled acyclic graph where:*

- *V is the set of vertices. A vertex $v \in V$ stands for an event, i.e. the start or the end of an activity, and is associated with a timestamp t that equals the respective* `start` *or* `end` *timestamp of the activity.*
- *$E \equiv E_w \cup E_{comm} \subset V \times V$, $E_w \cap E_{comm} = \emptyset$, is the set of directed edges. An edge $e = (v_i, v_j) \in E$ stands for an activity a:*`[start,end]`*, where $v_i[t]$ =* `start` *and $v_j[t]$ =* `end`*, and is associated with a type p and a weight w. The latter equals the total amount of time units spent in performing the activity, i.e. $e[w] = v_j[t] - v_i[t] =$* `end - start`*.*

An edge $e \in E_w$ denotes a worker activity whereas an edge $e \in E_{comm}$ denotes a communication activity, i.e. an interaction between two workers.

The direction of an edge e from node v_1 to node v_2 denotes time precedence between the source and the destination node.

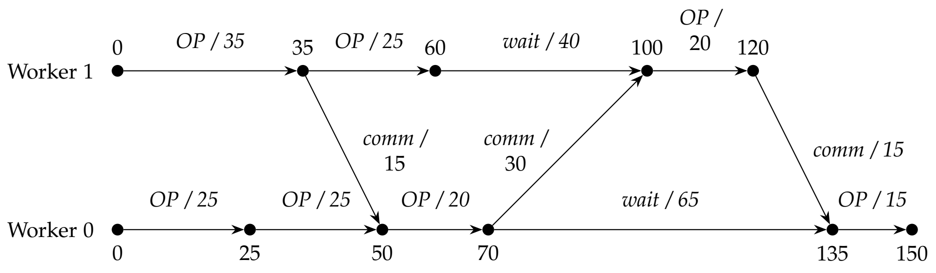


Figure 3.1: Example of an activity graph, including edge types, weights and event timestamps. *OP* denotes *data operation* activities, *comm* denotes *communication* activities, and *wait* denotes *waiting* activities.

Figure 3.1 shows a simple activity graph for a two-worker computation. Note that there are two subgraphs of all the worker activities. The two subgraphs are connected with each other only through communication activities.

Symbol	Description
a:`[start,end]`	activity a with `start` and `end` timestamps
G	activity graph
$G_{[t_s,t_e]}$	snapshot of activity graph G in the time interval $[t_s, t_e]$
$\prod_{t_s}^{t_e}(e)$	projection of edge e on the time interval $[t_s, t_e]$
$v[t]$	timestamp t of vertex v
$e[w]$	weight w of edge e
$e[p]$	type p of edge e
$\|\|\vec{P}\|\|$	total weight of edges in path $\vec{P}$
E_w	set of worker activities
E_{comm}	set of communication activities
E_{wait}	set of waiting activities ($E_{wait} \subseteq E_w$)
N_τ	number of running workers at time τ

Table 3.1: Notation used throughout this chapter.

Since we want to be able to compute the critical path in real-time for long-running computations, we have to define a way to slice the complete activity graph into snapshots which only contain activities in a certain time interval. Doing so also makes it possible to compute the critical path for only a specific part of the whole computation which might be of special interest, e.g. the processing of a query. Definition 3.4 defines how edges from the complete activity graph are projected on a time interval $[t_s, t_e]$. Intuitively, activities which are contained completely in the time interval are left as they are by the projection, whereas activities which overlap with the interval boundaries are cut off to fit into the interval.

Definition 3.4 (Edge Projection) *Let $e = (v_i, v_j)$ be an edge of an activity graph $G = (V, E)$, where $e \in E$ and $v_i, v_j \in V$. Let also $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a global clock $\mathcal{C}$. The projection of e on $[t_s, t_e]$ is an edge of the same type as e and is defined as follows:*

$$\Pi_{t_s}^{t_e}(e) = \begin{cases} (v_i, v_j) & \textit{iff } t_s \leq v_i[t], v_j[t] \leq t_e \\ (u_1, u_2) : u_1[t] = t_s, \\ \qquad\quad u_2[t] = t_e & \textit{iff } v_i[t] < t_s, t_e < v_j[t] \\ (u, v_j) : u[t] = t_s & \textit{iff } v_i[t] < t_s \leq v_j[t] \leq t_e \\ (v_i, u) : u[t] = t_e & \textit{iff } t_s \leq v_i[t] \leq t_e < v_j[t] \\ \textit{none} & \textit{otherwise} \end{cases}$$

Definition 3.5 (Graph Snapshot) *Let $G = (V, E)$ be an activity graph, and $[t_s, t_e]$, $t_s \leq t_e$, be a time interval with respect to a global clock $\mathcal{C}$. The snapshot of G in $[t_s, t_e]$ is a directed labeled acyclic graph $G_{[t_s, t_e]}$ that is constructed by projecting all edges of G on $[t_s, t_e]$.*

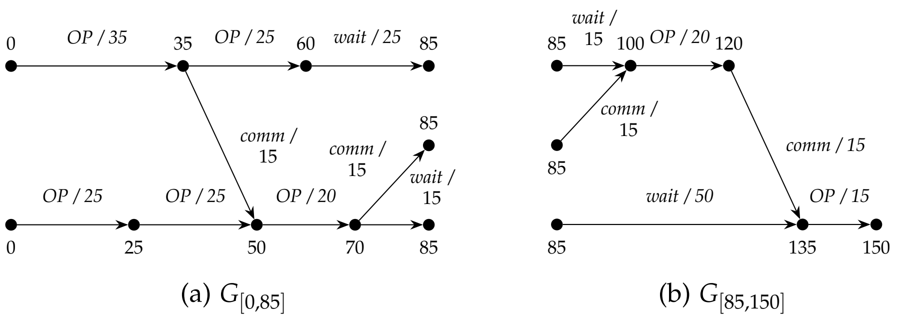


Figure 3.2: Slicing of the activity graph at $t = 85$, resulting in two graph snapshots.

Figure 3.2 illustrates how snapshots are created from the activity graph shown in Fig. 3.1. Note that at the snapshot boundary ($t = 85$), new nodes were introduced.

To simplify the description of properties and definitions in the upcoming sections, we also introduce the following three definitions:

Definition 3.6 (Minimum-Timestamp Vertices) *For an activity graph or graph snapshot $G = (V, E)$, the set V_s is the set of minimum-timestamp vertices in G:*

$$V_s := \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$$

Definition 3.7 (Maximum-Timestamp Vertices) *For an activity graph or graph snapshot $G = (V, E)$, the set V_e is the set of maximum-timestamp vertices in G:*

$$V_e := \{v \in V \mid \nexists v' \in V : v'[t] > v[t]\}$$

Definition 3.8 (Precursor) *For an activity graph $G = (V, E)$, the set of incoming edges (activities) for a vertex $v \in V$ is denoted as* $\textsc{Precursor}(v)$*:*

$$\textsc{Precursor}(v) := \{(u_i, u_j) \in E \mid u_j = v\}$$

3.2 Types of Activities

A logical operation may consist of multiple sub-operations which may span multiple levels of the software stack, including user-defined logic, system-level operations (e.g. serialization), OS scheduling, TCP requests, etc. The level of granularity at which activities are tracked depends on the instrumentation of the reference systems and varies significantly between use cases. Intuitively, a multi-layered activity tracking approach allows for more detailed performance analysis but introduces more overhead in the performance of the reference systems.

In Section 3.1, we have already mentioned the two main classes of activities: worker- and communication activities. In this section, we further describe these classes and divide them into the subtypes we considered for the purposes of this work.

3.2.1 Communication Activities

Any direct interaction between two workers is modeled as a communication activity. Communication activities serve two main purposes: firstly, they indicate dependencies between activities of two different workers and secondly, they model the time an external system (e.g. a network infrastructure) spent to transfer data. Communication events can only take place while a worker is *running*, therefore the source worker must be *running* at the `start` timestamp of the communication activity while the destination worker must be *running* at the `end` timestamp.

This activity type can be further categorized by its transfer mechanism or, orthogonally, by its purpose in the reference system. The transfer mechanism can be a network, IPC or even shared-memory transfer between two threads.

Shared-memory communication is assumed to be instant, i.e. the data is available immediately to the receiving worker after it was copied to the shared-memory buffer. Thus, such communication activities can have a zero weight. Note that the copy operations to- and from the buffer are performed by the source- and target workers respectively, hence they are part of worker activities, whereas the communication activity starts right after the data was copied to the buffer.

Depending on the type of the modeled system, the purposes of communication activities can vary. In the case of data-parallel systems, communication activities show the exchange of messages, which can be further divided into two classes: *progress messages* and *data messages*. The former include meta-data about the worker state (the progress in its computation), whereas the latter amount to batches of data exchanged between workers. Depending on the particular system design, progress messages are exchanged between workers directly, like in Storm [37] and Timely/Naiad [32, 28], or indirectly through a central *Task Manager*, like in Spark [40] and Flink [21, 1]. In the latter case, since communication activities need to connect two workers, the *Task Manager* needs to be modeled as a worker if the intention is to include the progress messages in the critical path analysis.

In some systems, the release of a lock could also be modeled as a communication activity.

3.2.2 Worker Activities

Worker activities are performed by workers during the time they are in the *running* state. We consider the following subtypes for worker activities:

Data Operation. This type models the execution of a data operator, e.g. any operator in the dataflow graph of a streaming engine. In the MapReduce ecosystem, it would for example model a `map` or a `reduce` operation.

In our model, data compression (as well as de-compression) as performed by many systems (e.g. Spark [40]) to reduce the total size of shuffled data, are special types of data operation activities. They often occur before and after a message transmission or an I/O operation (e.g. before spilling data to disk).

Serialization. This type of activity corresponds to serializing and de-serializing data, which is a common operation when messages have to be transmitted over the network to different workers of the system or when messages have to be exchanged between workers that run within different processes on the same physical machine.

Buffer Management. This type of activity is used to capture the time spent in maintaining the worker input and output buffers at runtime. Buffers are widely used in the form of queues by streaming engines like Storm [37, 2], Flink [1, 21], and Naiad/Timely [32], for buffering messages exchanged between different operators in the dataflow graph. Buffers are also used for asynchronous read and write operations when data is moved from/to disk, e.g. in Spark.

The time spent in buffer management corresponds to the time spent in pushing/pulling data into/from the buffers; this includes the time needed to acquire and release the respective locks (if any), and also the time spent in allocating and de-allocating memory in case of dynamic memory management.

Waiting. During this type of activity, the worker is waiting for a communication event of any kind from another worker, e.g. the arrival of a batch of data, a progress message, a response/acknowledgement for a TCP request, a lock release, etc. Waiting means that the worker is either *spinning* (e.g. polling for input like in the Naiad/Timely system [32]) or *blocked* in a function call (e.g. during a `read()` or `select()/poll()` system call on some socket(s)). Excessive occurrence of waiting activities may indicate inefficient synchronization barriers, high load imbalance, or network congestion, among others.

The waiting activities have a special meaning when it comes to critical path analysis. By definition, the worker does not produce anything useful during the time it is waiting, therefore no other part of the computation actually depends on the waiting activity being performed. Also, a waiting activity indicates that there are other activities being performed at the same time, for whose completion the worker is waiting. Therefore, waiting activities can *never* be on the critical path.

Furthermore, the instrumentation needs to make sure that the end event of a waiting activity corresponds to the end event of the communication activity that caused the worker to wake up from the waiting state. See also Property 3.11.

Waiting for Input. This activity type is similar to a waiting activity, the difference being that during this activity, the worker does not wait for a communication activity from another worker, but rather for input data from an external (unobserved) system. In data-parallel systems, it is sometimes the case that a worker can wait for input from an external system and from other workers simultaneously (e.g. in a system which alternates polling message and input queues). In this case, the type of event that terminated the waiting state defines whether the waiting phase was a *waiting* activity or a *waiting for input* activity.

Note that this activity has no special meaning to the critical path analysis like the waiting activity type has, in fact it is treated like any other worker activity and can also be part of a critical path. This activity can thus be used to detect situations in which the reference system's performance is limited by the performance of the external system that supplies input data.

I/O. During this type of activity, the worker is reading/writing input/output from/to an external system (e.g. a disk).

Idle. This is a special type of activity that is used to capture the time a worker is suspended by the scheduler and thus, is *idle*. A worker may be suspended due to various reasons, e.g. when it shares the same physical core with other workers of the system or when the garbage collection starts running in JVM-based systems like Spark and Flink [40, 21, 1].

The time a worker is idle (along with the actual cause) is captured through the appropriate instrumentation of the underlying OS or runtime environment/virtual machine. It is important to note that a worker is still considered to be in the *running* state according to our model, since it is still performing activities, even though it might be temporarily suspended by the scheduler (in which case it will be *performing* the *idle* activity).

Unknown. This type of activity is used for completeness, i.e. to model any worker activity that is not captured by the instrumentation of the reference system. A large number of unknown activities is usually an indicator of inadequate instrumentation. The automatic characterization of unknown activities is an interesting problem but is out of the scope of this work.

3.3 Implications for Instrumentation

Adding instrumentation to a reference system is usually a manual and therefore error-prone task. From the perspective of our performance analysis, the instrumentation of the reference system must satisfy some additional properties discussed in this section, otherwise the critical path is ill-defined. These properties constitute one of the contributions of this work. They can be checked by the system performing the performance analysis; the checks are efficient and can even be performed in real-time.

Property 3.9 (Minimum in-degree) *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in the time interval $[\, t_s\, ,\, t_e\,]$. Any vertex $v \in V \setminus V_s$ has in-degree at least one.*

Remember that V_s is the set of minimum-timestamp vertices as defined in Definition 3.6. Property 3.9 simply states that any event that has prior events must be caused by an activity that occurred earlier in time. In other words, "out-of-the-blue" events which are not in V_s indicate an insufficient instrumentation with respect to the critical path analysis.

Property 3.10 (Communication Existence) *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in $[t_s, t_e]$, and $\tau \in [t_s, t_e]$ be a point in time with respect to a global clock C. Let $S \equiv \{e = (v_i, v_j) \in E_{wait} \subseteq E_w \mid v_i[\mathrm{t}] \;\le\; \tau \;\le\; v_j[t]\}$. If $|S| = N_\tau$, where N_τ is the number of running workers of the reference system at time τ, then $\exists e' = (v_k, v_m) \in E_{comm} \subseteq E$ for which $v_k[t] \le \tau \le v_m[t]$.*

Property 3.10 states that there can be no point in time where all system workers perform waiting activities while no communication activity takes place. By definition, such states are problematic and can be observed only in the case of deadlocks. Hence, the existence of such points in the activity graph of a non-blocked computation indicates an insufficient instrumentation with respect to the critical path analysis. To prevent this, the instrumentation (or any post-processing logic applied to the trace recorded by the instrumentation) should make sure no waiting activities are created as long as the corresponding communication activity — which caused the waiting activity to end — has not been observed.

This property can be easily observed in the activity graph in Fig. 3.1 as well as the graph's snapshots in Fig. 3.2, for example at time $t = 85$. At this point, both workers are waiting, but there is an in-flight communication activity at the same time, which will eventually wake up one of the workers at $t' = 100$.

Property 3.11 (Wait State Termination) *In the complete activity graph $G = (V, E)$, for every waiting activity $(w_1, w_2) \in \textsc{Precursor}(w_2)$, there exists a non-waiting activity $(a_1, w_2) \in \textsc{Precursor}(w_2)$:*

$$\begin{aligned} &\forall w\big((w_1, w_2) \in \textsc{Precursor}(w_2) \wedge w[p] = wait\big) \\ &\quad \Longrightarrow \exists a\big(a \in \textsc{Precursor}(w_2) \wedge a[p] \neq wait\big) \end{aligned}$$

Property 3.11 follows directly from the definition of the waiting activity in Section 3.2. Remember that a waiting activity denotes the fact that a worker is waiting for a communication event (e.g. a message arrival) from another worker. The end timestamp of a waiting activity is the exact time when such a communication event occurs, therefore both the communication activity and the waiting activity have the same end vertex.

This property also applies to the graph snapshot, except for waiting activities at the end of the snapshot which were split by the edge projection. Their termination occurs in a later snapshot. For waiting activities which were only split at the beginning of a snapshot or not at all, the property holds.

3.4 Critical Path Algorithm

We define a critical path as a path on an activity graph snapshot $G_{[t_s,t_e]}$ which does not contain waiting activities and whose length is maximal:

Definition 3.12 (Critical Path) *Let* $G_{[t_s,t_e]} = (V, E)$ *be the snapshot of an activity graph* G *in the time interval* $[t_s, t_e]$*. We define the set of paths* $\mathcal{H}$ *on* $G_{[t_s,t_e]}$ *as* $\mathcal{H} = \{\vec{P} \subseteq E \setminus E_{wait} \mid \nexists \vec{P}' \subseteq E : ||\vec{P}'|| > ||\vec{P}||\}$*, where* $\vec{P}$ *denotes a path in* $G_{[t_s,t_e]}$ *and* $||\vec{P}||$ *denotes the total weight of all the edges in* $\vec{P}$*, i.e.* $||\vec{P}|| = \sum_{\forall e \,\in\, \vec{P}} e[w]$.

Any path $\vec{P} \in \mathcal{H}$ *is a critical path of the activity graph* G *in the time interval* $[t_s, t_e]$.

Any increase of the duration of any activity on a critical path results in an increase of the total runtime of the computation. Any decrease might result in a decrease of the total runtime, although this is not guaranteed, as there might be other critical paths whose length did not change as a result of the decrease.

The critical path as defined in Definition 3.12 is based on the observation that during any waiting activity, some other part of the computation (either another worker or the network) must be performing activities for whose completion/results the worker is ultimately waiting. Therefore, those activities should be on the critical path as they are what is causing the computation to be delayed, and not the waiting activity itself.

Another way of looking at it can be that any delay in a predecessor activity to a waiting activity only decreases the duration of the waiting activity and does not increase the duration of the overall computation. Therefore, it cannot be part of the critical path. This observation specifically, but also the general observation that waiting activities cannot be on the critical path, was originally described by Schulz [36][1].

Note that there may be more than one critical path for a given graph snapshot (i.e. the size of $\mathcal{H}$ may be greater than one). For example, consider a computation in which two workers are processing a perfectly balanced workload and are not communicating with each other. If we assume that both workers start and terminate simultaneously, the corresponding activity graph would be a disconnected graph consisting of two paths of equal

[1]Hollingsworth's [25, 24] algorithm implicitly also makes use of this fact.

length, one for each worker. Since both paths have the same length (and both do not contain waiting activities), they both would be part of the set of critical paths $\mathcal{H}$.

Algorithm 1 Critical path algorithm

```
procedure CriticalPath($G_{[t_s,t_e]} = (V, E)$)
    $\vec{P}_{critical} \leftarrow \emptyset$
    let $E_{critical} = \{e = (u_i, u_j) \in E \mid u_j \in V_e \wedge e[p] \neq wait\}$
    while $E_{critical} \neq \emptyset$ do
        let $a = (v_1, v_2)$ be an arbitrary edge from $E_{critical}$
        $\vec{P}_{critical} \leftarrow \vec{P}_{critical} \cup \{a\}$
        let $E_{critical} = \{e \in \textsc{Precursor}(v_1) \mid e[p] \neq wait\}$
    end while
    return $\vec{P}_{critical}$
end procedure
```

An algorithm that efficiently computes *one* critical path for a graph snapshot is shown in Algorithm 1. It makes direct use of our definition of the critical path in 3.12, which says that waiting activities cannot be on the critical path. This algorithm is heavily based on the backtracking algorithm described by Schulz [36].

The algorithm starts with $E_{critical}$ being the set of non-waiting activities at the end of the activity graph snapshot, i.e. those having an end vertex $u_j \in V_e$. $E_{critical}$ always contains activities that are on *some* critical path. Then, while $E_{critical}$ is not the empty set, the algorithm selects an arbitrary activity $a = (v_1, v_2)$ from $E_{critical}$ and extends the critical path $\vec{P}_{critical}$ by a. Afterwards, it recomputes the set $E_{critical}$ to be the set of non-waiting activities that are precursors to the start vertex v_s of the last added activity a, thus following the edges of the critical path backwards with regard to their event timestamps.

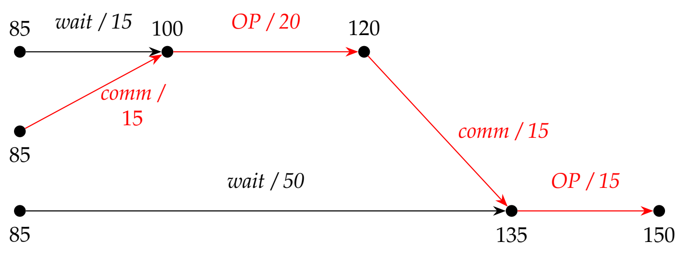


Figure 3.3: Critical path in $G_{[85,150]}$.

Figure 3.3 and 3.4 show the critical paths of the example graph snapshots $G_{[85,150]}$ and $G_{[0,85]}$ as found by the CriticalPath algorithm. For the snapshot $G_{[85,150]}$, which is at the end of the trace, $E_{critical}$ only ever contains one element at each step of the algorithm. This means there is no ambiguity, and therefore the critical path found is the only critical path in $G_{[85,150]}$.

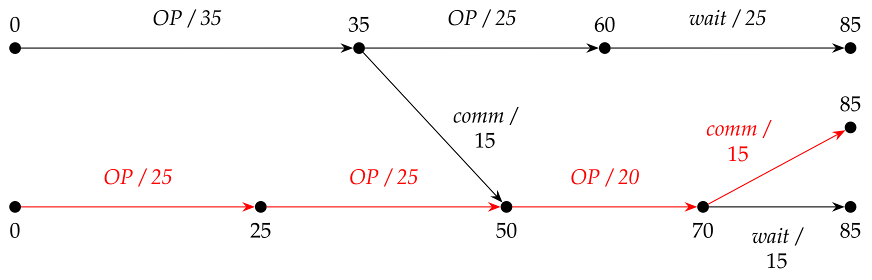


Figure 3.4: One possible critical path in $G_{[0,85]}$.

In the other snapshot, $G_{[0,85]}$, the algorithm also starts out with only a single node in $E_{critical}$, as all other edges ending on a maximum-timestamp vertex are waiting activities. However, at the node with timestamp 50, there is some ambiguity. The algorithm could either follow a critical path backwards along the communication activity or along the worker activity. As illustrated, only one critical path is computed in this case.

Note that this critical path algorithm does not need to take into account the weights of the different edges at each hop at all. Instead, it reduces the problem to following along edges which do not represent waiting activities. For a given graph snapshot, all paths computed by the algorithm will have exactly the same total weight (the length of the critical path), which is already known at the beginning given the maximum- and minimum vertex timestamps of the snapshot.

We now show that Algorithm 1 actually computes a critical path for a given snapshot:

Theorem 3.13 *Any path $\vec{P}_{critical}$ computed by* CriticalPath$(G_{[t_s,t_e]})$ *is a critical path of $G_{[t_s,t_e]}$, i.e. $\vec{P}_{critical} \in \mathcal{H}$.*

Proof sketch. We first show that CriticalPath$(G_{[t_s,t_e]})$ always finds a path $\vec{P}$ with no waiting activities that starts at some vertex $v_s \in V_s$ and leads to some vertex $v_e \in V_e$. It is easy to see that CriticalPath computes a valid path as it only adds edges from the set $E_{critical}$ to $\vec{P}_{critical}$, and after the first edge is added $E_{critical}$ is always defined such that it only contains edges that are connected to the start vertex of the previously inserted edge. Also, $E_{critical}$ as defined by the algorithm never includes waiting edges, so we know that the path $\vec{P}_{critical}$ cannot contain waiting activities.

Now, we distinguish between the case when $E_{critical} \neq \emptyset$ before the **while**-loop is entered and the case in which $E_{critical} = \emptyset$ at that point.

The latter case is trivial: if $E_{critical}$ is empty, it must be that either $E = \emptyset$ or alternatively, the snapshot ended with all workers performing waiting activities and no in-flight communication activity. But Property 3.10 says this cannot happen at any time. Therefore $E = \emptyset$ and only the empty path (with no edges) exists in the graph. This is exactly what the algorithm returns in this case, as the **while**-loop is never entered and therefore $\vec{P}_{critical}$ contains no edges.

In the former case, we know that there is at least one edge added to $\vec{P}_{critical}$ and that it ends on a vertex $v_e \in V_e$. Therefore, what is left to show is that $E_{critical}$ is only equal to the empty set when $\vec{P}_{critical}$ already contains an edge e with a starting vertex $v_s \in V_s$. To show that this holds as well, we make use of Property 3.11 and Property 3.9. If $\textsc{Precursor}(v_1)$ contains a waiting activity, Property 3.11 implies that it must also contain a non-waiting activity, in which case $E_{critical}$ is not empty. If $\textsc{Precursor}(v_1)$ does not contain a waiting activity, Property 3.9 still implies that $\textsc{Precursor}(v_1)$ must contain at least one precursor edge for v_1, unless $v_1 \in V_s$, in which case $\vec{P}_{critical}$ already contains an edge e with a starting vertex $v_s \in V_s$. Therefore, $E_{critical}$ is only equal to the empty set after an edge with a starting vertex from the set V_s is part of the path $\vec{P}_{critical}$.

Hence, we have shown that that $\textsc{CriticalPath}$ does indeed compute a valid path containing no waiting activities from some $v_s \in V_s$ to some $v_e \in V_e$. What is left to show is that any such path is a critical path, i.e. there exists no longer path in $G_{[t_s,t_e]}$. As the vertex timestamps in $G_{([t_s,t_e]}$ correspond to the start- or end-timestamps of the activities connected to a vertex, it is easy to see that the length of any path from a vertex v_s to a vertex v_e is equal to $v_e[t] - v_s[t]$. Because $v_s[t]$ is the minimal- and $v_e[t]$ the maximal timestamp in the snapshot, there cannot exist a longer path than $\vec{P}_{critical}$. Thus, $\vec{P}_{critical}$ is a critical path of $G_{[t_s,t_e]}$.

■

Because this critical path algorithm directly follows the trace along the critical path, it can be implemented very efficiently. For details of how this algorithm was applied to Timely Dataflow, see Section 5.3.

Chapter 4

Critical Path Computation: Preliminaries

While the critical path model discussed in Chapter 3 is sufficiently general, it does not address all issues that arise when one tries to apply critical path analysis to a system such as Timely Dataflow. Namely, it does not specify how to instrument a system to collect execution traces nor how to construct the activity graph in such a way that it can be processed efficiently.

This chapter focuses on Timely's internals, how it was instrumented, and how the event log data produced by the instrumentation is prepared for the critical path analysis.

Section 4.1 discusses Timely's runtime behavior, including operator scheduling, message transfer and progress tracking. The system's instrumentation is discussed in Section 4.2. Section 4.3 focuses on how the event logs are processed to form program activities. It also discusses how clock skew between different physical machines is handled.

4.1 Timely Dataflow's Runtime Behavior

Before we can define which sections of Timely we need to instrument, it is necessary that we have a good understanding of how Timely's runtime works. Later, understanding Timely's behavior will also be important to be able to infer the correct dependencies between events and activities as well as to figure out waiting phases during a worker's execution.

4.1.1 Operator Scheduling

Most Timely programs share roughly the following structure: First, each worker defines the dataflow graph of the computation. Then, it repeatedly reads some input data and calls `root.step()` until the computation is finished[1]. Each time `root.step()` is called, all the operators defined in the dataflow graph are scheduled exactly once. More precisely, `root.step()` causes the whole graph to be scheduled once. Remember that in Timely, the dataflow graph is structured into subgraphs (scopes). Each subgraph behaves like an operator itself, and is thus treated as such by its containing subgraph. The outermost subgraph is called the *root subgraph/scope*. If a subgraph is scheduled, it performs the following actions which are relevant to this work:

1. It schedules all its child operators
2. It exchanges progress messages with other workers
3. It applies any newly received progress information to its child operators

Therefore, if `root.step()` is called once, it recursively schedules all subgraphs and their respective children. This causes all the operators to be scheduled in the same order, in a round-robin way. Figure 4.1 shows an visual example taken from a trace which shows a worker executing a single step of a simple BFS computation (the dataflow graph of this computation is shown in Fig. 2.1).

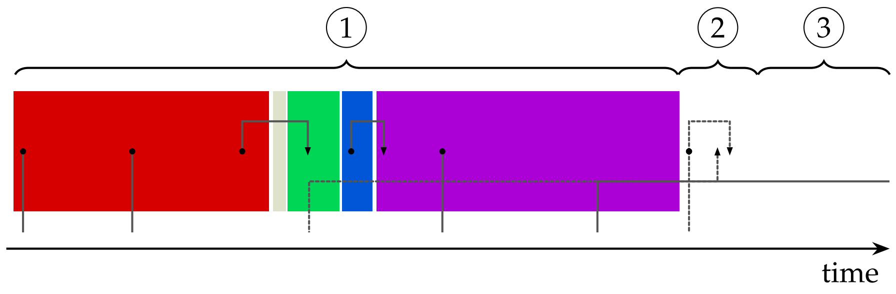


Figure 4.1: A single step of the Timely-based BFS implementation. (1) shows the scheduling of all the operators, including the exchange of data messages (solid lines). The second operator was not active, which is why it is displayed in a lighter color. (2) shows the exchange of progress messages (dashed lines), and (3) indicates when new progress is pushed onto the operators (this part is not instrumented, which is why the area is blank).

Note that Timely does not have a sophisticated scheduler. Instead, operators are scheduled regardless of whether they have any work to do or not. Therefore, it often happens that an operator is scheduled even though its

[1]It is also possible that only one of the workers is responsible for reading input data.

message queues are empty and no new progress notifications are available. In such cases, the operator will only check its message queues and check for any available notifications before stopping again[2]. When an operator does perform useful work, i.e. when it produces/consumes any messages or requests/receive progress notifications, it is said to have been *active* or having *shown activity* (these terms should not be to be confused with program activities; an operator which was executed but inactive also forms a program activity).

4.1.2 Progress Tracking

As mentioned in the previous section, each subgraph is responsible for exchanging progress messages with other workers. After a subgraph has scheduled all its child operators, if any of them have shown activity, the subgraph will continue on to broadcast a progress message to all the equivalent subgraphs on all the workers (including itself) telling them about the progress which has been made. Additionally, it will read any new progress messages originating from other workers (and from itself) from its queues and push new progress information onto its child operators. Finally, it will report back to its containing subgraph any progress the subgraph made, just like regular operators do when they stop. Remember that subgraphs themselves are treated like normal operators; hence different subgraphs on the same worker exchange progress information through this push/pull mechanism.

Note that progress notifications are not equal to progress messages or updates. Each operator which made progress produces a *progress update*. Multiple progress updates are collected and exchanged as *progress messages*. Any progress message, upon the processing of the contained updates, may or may not trigger the availability of a *notification* for a particular operator. A notification is therefore not a message which is exchanged between workers, but rather the potential result of the combined application of multiple progress updates (potentially from multiple progress messages) by each worker's progress tracking logic.

The fact that subgraphs only push progress updates onto their children after they were already scheduled can cause some additional delay between the arrival of a progress message and the time an operator actually sees its contents.

As an example, consider a computation with a nested subgraph, such as the Differential Dataflow-based BFS implementation shown in Fig. 2.3. An excerpt of an execution trace of this computation is illustrated in Fig. 4.2.

[2]Timely's programming model does not enforce this behavior, but properly implemented operators usually follow this pattern.

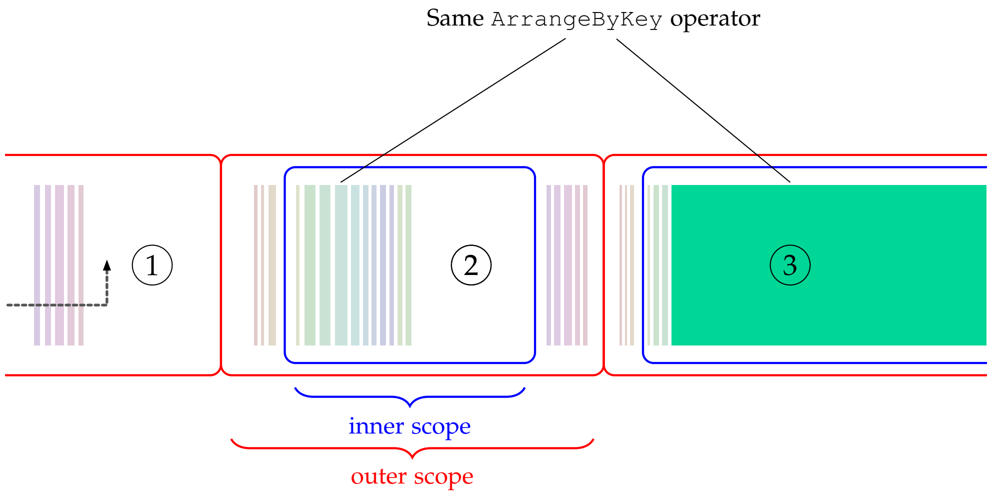


Figure 4.2: Nested scopes and delayed pushing of progress updates onto operators.

The red rectangles indicate the root scope which is scheduled at each step, and the blue rectangles indicates the inner scope of the BFS program.

At (1), the outer subgraph has new progress information available (from the received progress message) and pushes it onto its child operators, including the inner subgraph. In the next step of the computation, the outer subgraph schedules all its child operators, including the inner subgraph. The inner subgraph in turn also first runs all its child operators and only *afterwards* applies any new progress information to them ((2)). Hence, the new progress information which was pushed onto the inner subgraph by the outer subgraph at the end of the previous step ((1)) is only now pushed onto the operators contained in the inner subgraph. Therefore, children of the inner subgraph will have to wait for the next step until they will actually be able to react to any notifications based on progress made by the outer subgraph. This is visible when at (3), an operator becomes active, even though no new message or progress message arrived since the last time it was scheduled. This activity is in fact the result of a notification which was triggered by progress information pushed onto the operator at (2). This propagation delay of progress information makes inferring dependencies between operator activity and the arrival of progress messages more difficult. Also, the exact delay is highly dependent on the time the progress tracking logic is called as well as the specifics of Timely's scheduling algorithm.

4.1.3 Communication between Workers

The only interactions between workers in Timely Dataflow take place through message- and progress message transmissions. Progress messages are exchanged, as described in the previous sections, at the end of a subgraph's execution, while data messages are generally exchanged during the execution of operators. One notable exception is Timely's `Input` operator, which gives the user access to its output channel directly. Thus, the application can send messages while the `Input` operator is not scheduled. Data messages can only be exchanged over the channels defined by the dataflow graph, whereas progress messages are broadcast by every scope/subgraph.

Aside from these differences, progress messages and data messages are transmitted the same way, therefore "message" can mean a progress message or a data message from here on in this section. We differentiate between three types of message exchanges: worker-local, process-local and inter-process/networked.

Worker-local and process-local messages are directly appended to the respective input queues of the target operator on the target worker thread by the sending worker. Timely uses non-blocking queues (which can grow infinitely large) for this purpose.

Messages that need to be sent to a worker thread residing on a different process are sent over TCP/IP. Each Timely process maintains a single TCP connection to each other Timely process. The dataflow- and progress channels leading to workers in a particular process are multiplexed over the single connection to that process. For this purpose, each channel is assigned a communication channel identifier, which is different from the channel identifiers used for logical channels in the dataflow graph. For each connection a process maintains, two dedicated threads are responsible for sending and receiving messages. The sender thread reads messages sent by operators from a queue and sends them over the TCP connection, from which the receiver thread reads the messages and puts them into the target operator's/subgraph's input queue.

When applying the model from Chapter 3, one could model the sender/receiver threads as *workers* themselves. This could be helpful to find communication bottlenecks, and to split the message transfer time into transmission (worker activity) and propagation time (communication activity)[3]. However, treating the sender/receiver threads as workers would also require more instrumentation to detect waiting phases during their execution, and would thus add a lot of complexity to the problem. Since the threads are not doing much interesting work, in fact, they only have one purpose each, we

[3]Technically, it could be split into even more fine grained categories, e.g. to separate the time a message spent buffered by the OS.

decided not to model them as actual workers. Nonetheless, we still want to track the time messages spent in the send/receive buffers and how much time the transmission plus propagation of messages (i.e. the transfer of messages) took. This can be solved easily by extending program activities by two simple attributes for the send- and receive buffer times.

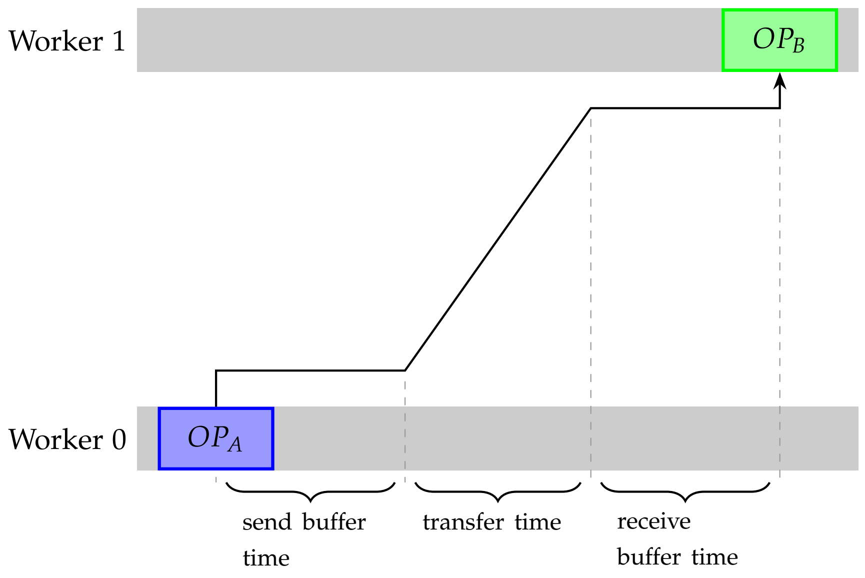
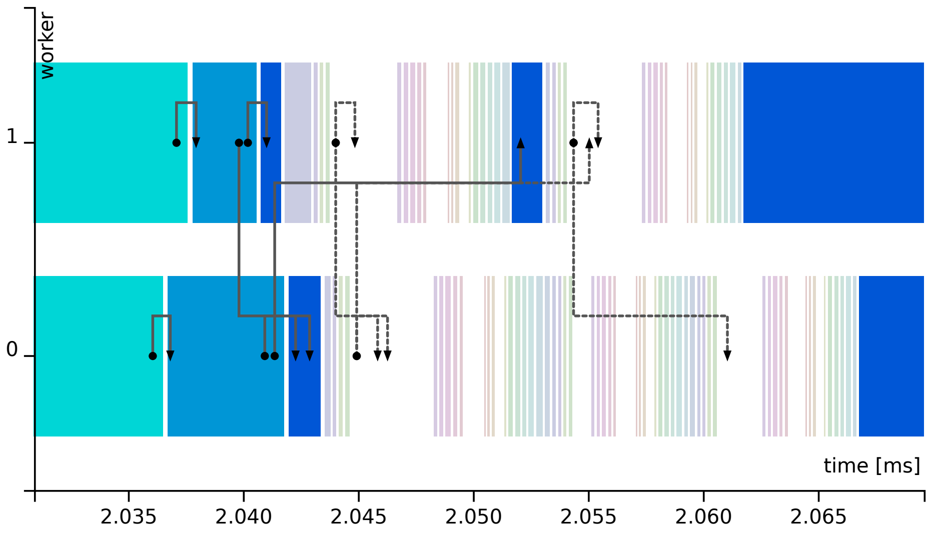


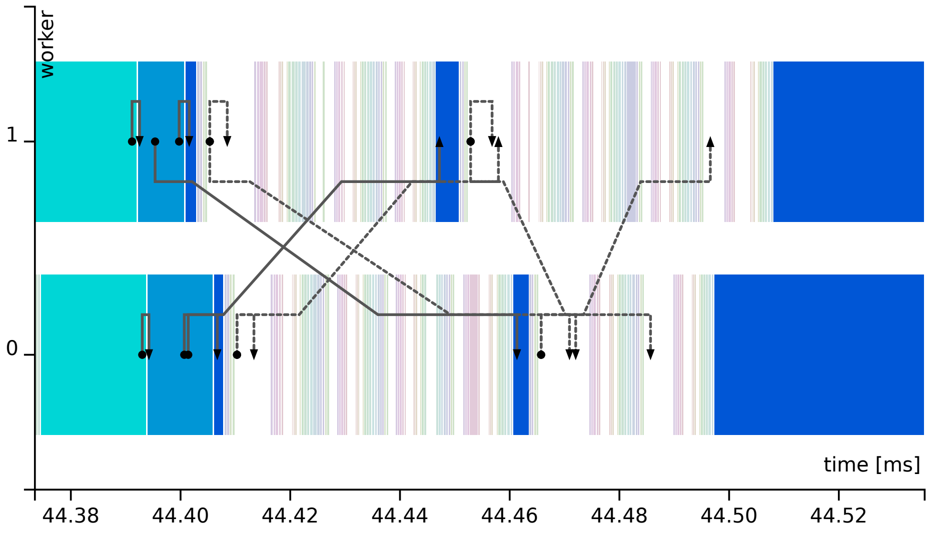
Figure 4.3: Message transfer phases in Timely Dataflow. The *transfer time* includes both the transmission and propagation over the network. The horizontal lines show the time a message spent in a buffer. The vertical lines leading to/from the operators designate the time a message was read/put into a queue.

Figure 4.3 shows how a message transmission is modeled as a result. The *send buffer time* is the time a message spent in a queue waiting to be transmitted by the sender thread. The *transfer time* is the time a message spent on the wire, staring from the beginning of transmission until the whole message arrived at its destination node. Finally, the *receive buffer time* specifies how long a message spent in the input queue of an operator/subgraph until it was read. The actual communication activity is defined to include the *send buffer time* as well as the *transfer time*. However, as mentioned above, both the send- and receive buffer times are kept as attributes to the activity. The receive buffer time attribute is needed to compute the time at which a (progress-) message was actually processed by the worker.

This approach still allows one to detect a potential bottleneck in the sender/receiver threads. For example, if messages have a high send buffer time but their transfer time is normal given the message size, this might indicate that



(a) Process-local communication



(b) Network communication

Figure 4.4: Two 2-worker traces illustrating the differences between process-local and network communication

many workers are sending messages to the target process and the responsible sender thread is becoming a bottleneck.

Because worker-local and process-local messages are put into the target's input queues directly, their send buffer time will be zero. Also, as already mentioned in Section 3.2, communication over shared memory is assumed to be instant (it takes no time on an external system), therefore the transfer time of such activities will also be zero. The time spent actually copying the message into the queues would be a buffer management activity as per Section 3.2, but is currently not tracked in our implementation. This difference between network- and local messages can be seen in the trace visualizations shown in Fig. 4.4.

4.2 Instrumentation

One advantage of Timely's rigid definition of a computation as a dataflow graph is that it makes it straightforward to insert basic, coarse-grained instrumentation. In fact, Timely already has a logging facility as well as static instrumentation for most of the events we would like to record.

4.2.1 Logging Facility

All events recorded through Timely's logging facility include a high-precision timestamp in nanoseconds, which is re-based to resemble a nanosecond-precision UNIX timestamp, in order to make it comparable across different machines (see also Section 4.3.1 on how clock skew is handled). Also, as each worker creates a separate stream of log events[4], the originating worker ID is also known for each recorded event. The events are first kept in a buffer by Timely, which is flushed at the end of every step of the computation (after the root scope was scheduled once). Currently, the logs are written to disk, however, the logging backend can be changed easily such that logs could also be sent over a socket, for example.

A similar, but slightly simpler logging facility was implemented as part of this work in order to log communication events, as those must be recorded on a different layer of the software in which Timely's logging facility is not available.

4.2.2 Recorded Trace Events

When Timely's logging is enabled, it already records the following events:

`ScheduleEvent`. These events are recorded whenever an operator (or subgraph) started running or stopped. A `ScheduleEvent` includes a worker-local operator identifier. When a `ScheduleEvent` designates the stopping of an operator, it also includes a flag indicating whether or not the operator seemed to have performed any useful work, i.e. produced/consumed any messages or requested/received any progress notifications.

`MessagesEvent`. This type of event is recorded whenever a data message is either created and put into a message queue ("send" event) or when one

[4]In fact, each worker creates a separate stream for each log event type.

is read thereof ("receive" event). It includes an ID of the dataflow channel over which the message was sent, the source- and target workers (can be identical for worker-local messages), the number of records contained in the message, as well as a sequence number.

Receive events can only occur during the execution of the receiving operator. Send events *usually* only occur during the execution of the sending operator, with the notable exception of Timely's `Input` operator, which allows the main program to put input data directly into a message queue (i.e. not during the execution of the `Input` operator itself).

`OperatesEvent`. These events denote the creation of any operator during the construction of the dataflow graph, which is normally done at the beginning of a computation. They contain the worker-local identifier of an operator which can be linked to the one recorded in `ScheduleEvents`, as well as the globally unique address of the operator which specifies its position in the dataflow graph. The event also contains a descriptive name of the operator (e.g. "`Join`" for a join operator).

`ChannelsEvent`. This type of event is similar to the `OperatesEvent`, but specifies the creation of dataflow channels instead. The events contain a worker-local channel identifier which can be linked to the one recorded in `MessagesEvents`, as well as a global scope address, source/target operators and source/target ports (an operator with multiple in-/output channels has one in-/output port for each).

For the purposes of this work, we extended the instrumentation to include the following events:

`ProgressEvent`. This type of event is recorded whenever a worker either receives or sends a *progress message*. Since progress messages are broadcasts to all workers, there are N receive events for each send event, where N is the number of workers. Each event includes the source worker ID as well as a sequence number. Moreover, as progress messages are exchanged for each scope in the dataflow graph, these events also include a scope address.

Like `MessagesEvents`, `ProgressEvents` are recorded when progress messages are put into/read from a queue. For progress events, this happens during the time Timely executes its progress tracking protocol. Remember that the progress tracking protocol is executed by each subgraph after its child operators have all been scheduled once.

`PushProgressEvent`. These events are recorded whenever Timely delivers new *progress updates* to an individual operator, as described in Section 4.1.2. For example, this will happen immediately after Timely read new progress messages (after a number of receive-type `ProgressEvents`) when it applied the contained updates to the operators. For more information about progress tracking and scheduling, see Section 4.1.2 and Section 4.1.1.

`CommunicationEvent.` These events are only recorded when a computation uses multiple processes that communicate over a network. They either denote the time when a sender thread started to write a particular (progress-) message to the TCP socket, or when the receiver thread finished reading a message from the socket. A communication event also contains the source- and destination worker IDs, as well as a sequence number which can be linked to the one recorded by the corresponding message- or progress events. Moreover, each event contains a *communication channel identifier*, which is used to uniquely identify the individual channel of the multiplexed TCP connection between two processes. Note that these identifiers are separate from the *dataflow channel identifiers* recorded in message events. If a computation uses network communication, the message- and progress events will additionally contain the ID of the communication channel used for each (progress-) message. This makes it possible to link message- or progress events to their corresponding communication events using the sequence number, source/target workers, and communication channel identifiers.

`ApplicationEvent.` These events are simply start/stop events with a user-defined ID to allow custom, application-specific instrumentation. See Section 4.3.2 for more information about application-defined activities.

4.3 Trace Preprocessing

This section discusses how the recorded trace events are processed as preparation for the wait-state analysis and the critical path computation. The log streams from each worker can be processed in parallel with very little data exchange, which makes the preprocessing step a highly data-parallel task.

4.3.1 Correcting Clock Skew

As mentioned in Section 4.2, the recorded trace events include a nanosecond-precision timestamp which is comparable between workers. However, these timestamps are still based on the local clocks of each physical machine. If a computation uses multiple processes distributed over multiple physical machines, the problem of clock skew between the machines arises.

In order to address this issue, we use a small tool to measure the clock skew between the machines on which the computation will be/was executed. The tool simply sends out a number of probe requests over UDP to each machine. The machines reply with a high-precision timestamp. The client measures the round-trip time the probe took and also records a local timestamp. Using the remote and local timestamps as well as the measured round-trip time, it then computes the clock skew between the two machines. The clock skew is averaged over all the probes. Tests on a single machine with an artificially increased network latency on the loopback device have shown that for a number of 100 probes, the clock skew measurement error is about three orders of magnitude lower than the network latency even using this simple approach.

The clock skew measurements are then used to correct the timestamps of the recorded trace events. The drawback of this solution is that it does assume the clock skew to be static, which is of course not true in practice. For long-running computations, clock drift during the computation can therefore still cause accuracy problems. However, the impact is limited to the decreased accuracy of (progress-) message transfer times. The timestamps of events defining the transfer time are the only ones which need to be compared across machines. Any other communication- or worker-related events are not affected. Also, waiting states are still detected normally, which means also the structure of the computed critical path will still be correct.

Although the remaining accuracy issues are deemed to be acceptable, it would still be beneficial if future versions of Timely Dataflow's logging infrastructure would measure clock drift periodically during the computation and include this information in the logs (or produce corrected timestamps directly). Since high network load from the computation might influence clock skew measurements, maintaining accuracy is difficult if the measurements are performed using a separate tool. Timely's runtime however could decide to perform measurements during idle- or low-load phases of the computation to get accurate measurements. Also, a more sophisticated protocol could be used to provide higher accuracy, an example would be the Precision Time Protocol (PTP) [7]. For more information about future work, see Chapter 8.

4.3.2 Constructing Program Activities

By joining the events recorded by Timely's instrumentation, it is easy to construct the program activities (as in Section 3.2) of a particular execution trace. Consecutive `ScheduleEvents` for the same operator are joined to form *data operation* activities. Schedule events of subgraphs, which overlap with their child operators, are simply ignored. Send- and receive-type `MessagesEvents` are joined based on source/target workers, their channel, as well as their sequence number. The channel is determined by joining the worker-local channel ID of the message events with the corresponding `ChannelsEvents`, which contain the unique address of the channel. Similarly, `ProgressEvents` are joined based on their scope addresses, source workers and sequence numbers. If a (progress-) message was sent over

Activity Name	Formal Type	Constructed Using
Operator	*Data Operation*	`ScheduleEvents`, `OperatesEvents`
Data Message	*Communication*	`MessagesEvents`, `ChannelsEvents`, `CommunicationEvents`
Progress Message	*Communication*	`ProgressEvents`, `CommunicationEvents`
Application	Any worker activity type except *Waiting*/*Unknown*	`ApplicationEvents`
Waiting	*Waiting*	–
Waiting for Input	*Waiting for Input*	–
Unknown	*Unknown*	–

Table 4.1: Activity types used for Timely.

the network, it is also joined with the corresponding communication events based on the communication channel IDs as well as the sequence number. Since the sender- and receiver threads are not treated as workers, the actual *communication activity* is defined from the time the message was put into the sender queue to the time it was received by the receiver thread. However, the send- and receive buffer times are retained as extra attributes of the activity. If the communication is process-local (or even worker-local), the send events alone already defines the start- and end timestamps of the resulting *communication activity* (remember that the activity has a duration of zero in this case).

`PushProgressEvents` do not form actual activities, but are used only as marker events that help in detecting waiting phases (see Section 5.1).

Furthermore, `ApplicationEvents` are used to define application-/user-defined activities. They can be used to achieve a more fine-grained resolution, as they take precedence over most other recorded activities. For example, an operator might itself perform multiple sub-activities which are of interest, which can be logged individually using `ApplicationEvents`. Alternatively, they can also be useful to instrument code which is not executed during `root.step()`, e.g. code outside the scope of any operator. An example of this could be code which is responsible for reading input to feed into the computation, which is typically done by the main program, not an operator. Application activities using different IDs can also be nested; in this case, the innermost activity takes precedence. However, the application must make sure that they are nested properly, i.e. they may not overlap with the boundary of another activity, including predefined ones.

Finally, *Unknown* activities are only used as an abstract construct an are never explicitly created for efficiency reasons.

The result of the trace preprocessing step can be seen in Fig. 4.4. These preprocessed traces might be helpful for manual analysis on their own, however, in order to compute the critical path, one important type of activity is still missing: the *waiting activities*. The next chapter describes how our analysis identifies the waiting phases during the execution based on these traces, and how the actual critical path algorithm is applied.

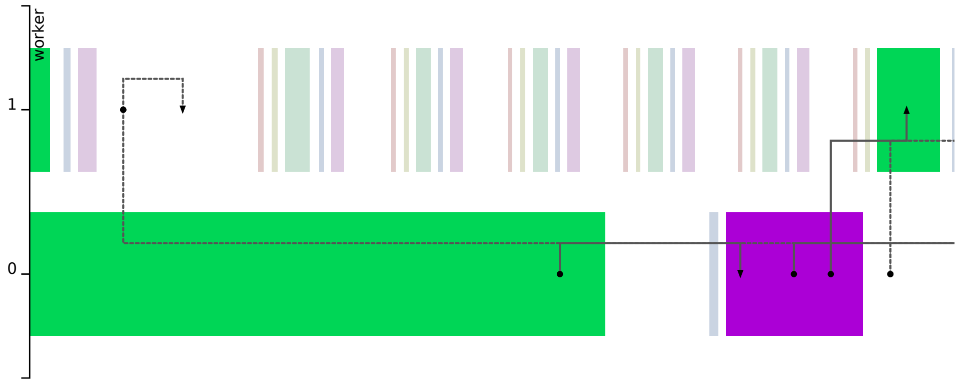
Chapter 5

Critical Path Computation: Essentials

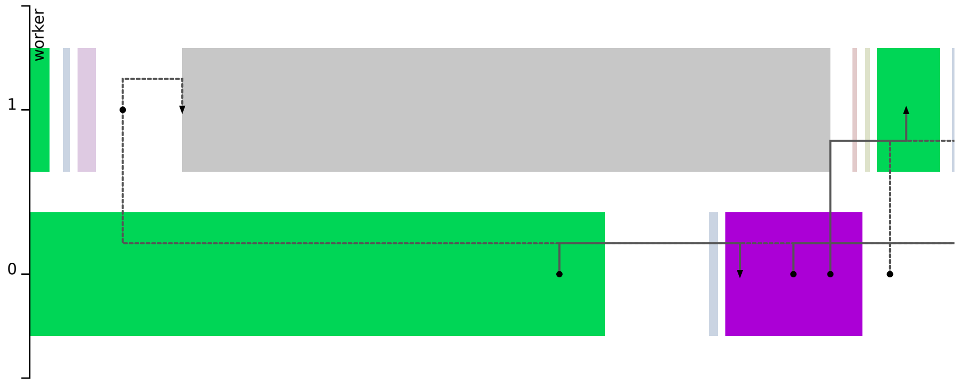
In the previous chapter, we laid the groundwork for the critical path analysis of Timely Dataflow. In this chapter, we now focus on the essentials. Section 5.1 discusses how waiting phases occurring during a worker's execution are detected. It also includes a list of assumptions about the behavior of Timely operators (and the system in general) which are necessary for our approach to work correctly. Furthermore, it discusses the causal relationships between events in the program's execution trace which we consider in our method. Section 5.2 discusses how our implementation relates to the formal activity graph model. Finally, Section 5.3 describes how the actual critical path algorithm is applied.

5.1 Identifying Waiting Phases

One of the biggest challenges in finding the critical path for a Timely Dataflow computation is detecting the waiting phases in a worker's execution, i.e. periods of time in which a worker is not doing any useful work. In previous work, for example by Schulz [36] and Böhme et. al. [13], these waiting phases could be identified by looking at blocking system/function calls. However, unlike in many other parallel or distributed systems, worker threads in Timely never block while waiting for messages. Instead, they are running constantly. As described in Section 4.1.1, the operators are just scheduled repeatedly, polling their input queues, essentially busy-waiting until new messages (or input data from an external system) arrive. The fact that there are no explicit start- or end points of a waiting phase, as they would be defined if a blocking system call was used, makes detecting them considerably more difficult. Hence, we define our own method for identifying waiting phases.



(a) Before the wait-state analysis, showing worker one spinning. The computation was stepped five times without any operator showing any activity.



(b) After the wait-state analysis. The waiting activity, shown in gray, replaces the spinning phase and is terminated by the arrival of a new data message.

Figure 5.1: Traces before and after the wait-state analysis.

First, we have to define when a worker is considered to be in a waiting state:

Definition 5.1 (Waiting State) *A worker is considered to be in a waiting state at time τ if it is not doing any useful work at time τ and if there is no point in time $\tau' \geq \tau$ where the worker performs useful work which is directly caused by a prior event occurring at a time $\tau'' \leq \tau$.*

Intuitively, we consider a worker to be waiting if it will not continue doing useful work unless a future event triggers it. Such an event can either be the arrival of a (progress-) message or of input supplied by an external system. In the former case, the worker would have performed a *waiting* activity as defined in our formal model (see Section 3.2). In the latter case, it would have performed a *waiting for input* activity.

Useful work is defined as any work which contributes to the progress of the computation. Effectively, useful work was performed by operators which have shown *activity*, i.e. produced/consumed any messages or requested/received progress notifications, and by Timely's runtime/subgraphs when it processed and/or sent/received progress messages. Additionally, some work done outside operators by the program, e.g. the reading, preprocessing or generating input, could also be considered useful work. However, since this part of a program is not automatically instrumented, it is treated as an *unknown* activity and is not considered to be useful work by default. An easy remedy to this is to put any reading/preprocessing/generation of input into a designated operator. Obviously, useful work is defined with a certain (relatively coarse) granularity. For instance, it would be very hard to determine whether every single code instruction executed by an operator performed useful work; however, it is easy to determine whether the whole operator performed any useful work.

In practice, accurately detecting the exact points in time when a worker is waiting as per Definition 5.1 is still a difficult task which requires intimate knowledge of the program itself. To bypass this challenge, our algorithm makes a number of assumptions about the behavior of Timely Dataflow programs and uses a heuristic approach to determine the causal links between events. All of these assumptions and heuristics are described in detail in the following sections.

5.1.1 Assumptions about Operators

The following assumptions about operator behavior need to hold true for the wait-state analysis and the critical path analysis overall to produce correct results. The assumptions are deemed to be reasonable given Timely's programming model, however, in practice developers themselves need to make sure not to violate them:

Operators do not defer the processing of data messages. If an operator reads a data message from one of its input queues, it will process the message during the same execution of the operator, not store it and process it when the operator is scheduled again at a later point in time. This assumption is necessary to establish the causality between a message arrival and any activity that an operator might show (e.g. the production of an output message).

If the operator would defer the processing of the input message and process it at a later time, it could not be determined whether it performed useful work at that later time, if it did not consume any additional message(s) and did not produce any either. If the operator did produce a message as a result of the deferred processing, it must be assumed that it generated that message on its own (or based on external input it read).

Note that this assumption only applies to cases in which the message is the sole cause of the activity. It does does not exclude the possibility of an operator storing the message (or modifying its local state based on the message in another way) and deferring its processing until *another* message or notification arrives. This behavior poses no problem, as in this case, the second message/notification can be seen as the direct cause of any useful work being done.

This assumption cannot be guaranteed by Timely's runtime, however, it is not unreasonable to expect operators to behave this way, given that deferring the processing of messages would only harm the overall performance of the computation in almost every case imaginable.

Operators do not defer their reaction to new progress information. This assumption is very similar to the previous one. Whenever a subgraph pushes new progress information onto the operator, which is reflected by a `PushProgressEvent` in the trace (see Section 4.2), it does not defer its reaction to the resulting notifications to a later time, when it is being scheduled again. This assumption is needed to establish the causality between the arrival of new progress information for an operator and any useful work an operator might perform as a result.

Again, this assumption does not exclude the possibility that an operator modifies its local state based on newly available progress information and then defers any further processing until a new message or more progress information arrives.

Operators only interact with operators on different workers through the mechanisms provided by Timely Dataflow. Operators can only interact with operators on other workers through data messages and indirectly through Timely's progress tracking. This assumption might seem a bit obvious, however it is important to note that all interactions need to be captured by the existing instrumentation for the critical path algorithm to work.

Operators running on the same worker do not share any state. Even if operators on the same workers interact, they need to do so by sending data messages to each other. Again, this might seem obvious given Timely Dataflow's programming model. Yet, Timely's runtime does not enforce this rule. In fact, it is very easy (and one might say tempting) to share state between operators on the same worker to avoid the overhead of exchanging data messages.

The fundamental problem with sharing state is that the interaction between the operators is not observable in the trace data. Just like when an operator deferred the processing of a message, the cause of an operator's activity when reacting to a change in a shared state is hidden and cannot be determined, as the state-sharing is not recorded by any instrumentation.

There is a remedy which still allows some operators on the same worker to profit from the performance benefits of state-sharing, without affecting the critical path computation. If two operators share a state, whenever one of them modifies the state, it also needs to send an (empty) signal message to the other operator, indicating that the state has changed. The signal message needs to be sent during the same operator execution in which the state was modified. When the other operator is scheduled, it can react to the reception of the signal message by accessing the shared state. Because the operators are both running on the same worker and are executed strictly sequentially, when following this pattern there is no possibility that one of them reacts to a state modification before the arrival of the signal message indicating the change. Thus, the signal message, which will be visible in the trace, can be seen as the direct cause of any useful work performed by the receiving operator. Conceptually, one could also imagine that the signal message contains, and hence, transfers the entire shared state from one operator to the other. Looking at the situation this way, the two operators would still be acting in conformity with the actor model [23].

There already exist operators which make use of this pattern, an example being the `ArrangeByKey` operator in Differential Dataflow [4].

5.1.2 Causal Relationships between Events/Activities

The generality of Timely's programming model makes determining the exact causes of an operator's behavior a task which is impossible to fulfill accurately without additional knowledge about the implementation of said operator. For instance, let us assume that an operator has seen new progress information and has also received a message. Now, without having any additional information available, we cannot determine what percentage of the operator's execution time was spent processing the message, and what percentage (if any) was spent processing a potential progress notification. Any progress the operator makes could be the result of either the message or the progress information, or a combination of both. Nevertheless, we can still conclude that the arrival of the data message was at least partially responsible for the operator's progress, since we know the message was consumed. Thus, the message is deemed to be a *direct cause* of the operator's activity, even though it might only be partially responsible. Data- or progress messages which were already read/processed earlier could potentially be *indirect causes*. However, the worker can still be waiting for the direct cause of some activity after an event which denotes an indirect cause occurred. Therefore, we must not consider indirect causes of activity when identifying waiting phases.

As mentioned before, we consider two types of useful work (which advance the computation): operator activity or progress tracking activity. Note that "activity" as used in this section does not denote *program activities*, but useful work being performed. Progress tracking activity is meant to denote any computation performed by Timely's progress tracking implementation, e.g. the reading and processing of a progress message. The rest of this section describes the heuristics used to determine all *direct causes* of any useful work being performed, which are needed to detect whether or not a worker is in a waiting state at a given time, according to Definition 5.1.

Causes of Operator Activity

The following events are deemed to be direct causes of an operator showing activity (i.e. making progress). Note that multiple direct causes can apply at the same time:

The arrival of a data message destined to the operator. The arrival of a data message in an operator's input queue is guaranteed to be at least a partial cause of the destination operator's activity during the particular operator execution during which the message was read from the queue. If the message arrived before the operator was scheduled, it is always treated as a direct cause. If it arrived during the operator's execution, but was read from the queue only when the operator was scheduled again at a later time, it is treated as the cause of any activity during the later scheduling. If the message arrived after the operator already started running but was read during the same execution however, it is not treated as a direct cause, as an earlier event might already have triggered the operator's activity. In case a message arrives just in time when an operator is running which would otherwise not have performed useful work, this heuristic is inaccurate. Note that such a scenario is highly unlikely to happen, since operators which do not perform useful work are usually only scheduled for a tiny period of time.

In other words, this means that as long as a data message buffered in an input queue of an operator, the worker cannot be in a waiting state according to Definition 5.1, as there must come a time in the future when said message is read and processed by the operator, which is useful work directly caused by the arrival of the message.

The processing of new progress messages or the pushing of new progress information onto any operator. Determining whether or not an operator

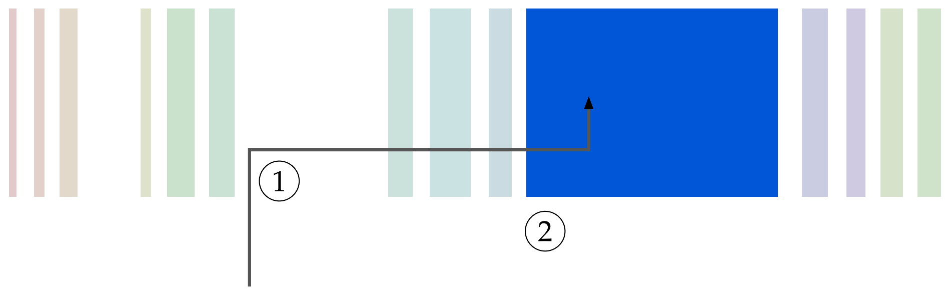


Figure 5.2: The arrival of a data message at (1) is treated as the direct cause of the operator activity beginning at (2) (remember that lighter colors indicate operators which have not shown activity). The message is read from the input queue during the same execution of the operator.

reacted to new progress information would require knowledge of the operator's implementation as well the content of each individual progress update. The latter could be recorded by appropriate instrumentation and the analysis could replay the whole progress tracking protocol, although this would likely substantially harm the performance of both the original computation (due to instrumentation overhead) as well as the analysis. The former however is unrealistic given the generality of Timely's programming model. Therefore, our heuristic assumes that any time an operator has shown activity after Timely has either read a new progress message or pushed progress information onto any operator, this activity is directly caused by the newly available progress information.

This heuristic is inaccurate in cases where progress information was received and an operator also received a data message, but did react only to the data message, not the progress information. It is also inaccurate in cases where progress information was received but an operator became active because it generated/read input data.

The availability of external input or the generation of input. If neither of the previous two causes are present to explain an operator's activity, our heuristic approach assumes that the operator in question must have read input from an external, unobserved system, or alternatively, generated input itself. As the precise cause cannot be observed in this case because of the lack of instrumentation, it is assumed that the triggering event (i.e. the new input data becoming available) happened exactly at the time the operator started running.

Causes of Progress Tracking Activity

Progress tracking activity means the processing of progress updates and the sending/receiving of progress messages. This work is performed by Timely's progress tracking implementation. The following events are deemed to be direct causes of progress tracking activity (again, multiple causes may apply simultaneously):

The arrival of progress messages. Just as the arrival of data messages for an operator, the arrival of progress messages for a particular subgraph causes the subgraph's progress tracking logic to perform useful work at the time the progress message is read from the queue. There is no ambiguity in this case.

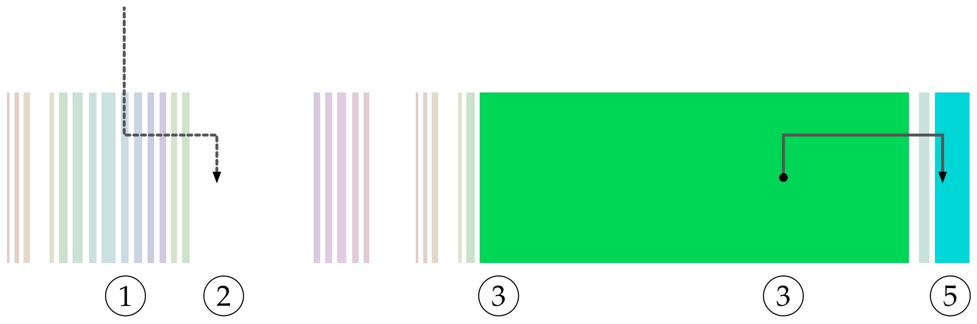


Figure 5.3: The arrival of a progress message at (1) causes progress tracking activity at (2). This progress tracking activity in turn causes operators to become active at (3) and (5). Furthermore, at (3) a message is sent, which is also a cause for operator activity at (5).

The reading a progress message from a queue. This is assumed to be the cause of any following `PushProgressEvents`, i.e. the application of progress information to an operator. The latest progress message which was read is assumed to be the cause for the next `PushProgressEvent`.

This heuristic is incorrect in two ways: first, if multiple progress messages were read, any of them (also a combination) could be the cause. Second, the cause of a particular `PushProgressEvent` could also be an outer subgraph which pushed new progress onto an inner subgraph in the previous step, as described in Section 4.1.2.

The first case does not affect the results of the wait-state analysis. The worker would not be in a waiting state no matter which exact progress message(s) were responsible for the `PushProgressEvent`, as until the last progress message was read from the queue there would be at least one progress message buffered. This negates the possibility of a waiting state during that time.

The second case can lead to incorrect results if a progress message was read immediately prior to a `PushProgressEvent`, but the event was caused by new progress information from the outer subgraph (delayed pushing of progress information, as can bee seen in Fig. 4.2). In this case, the progress message is erroneously assumed to be the causal factor, which could lead to a waiting state being detected (again erroneously) before the arrival of the progress message. This issue could be solved by using a more detailed model of the progress tracking logic and by the replaying of the content of progress updates, as described in Chapter 8.

The pushing of progress information onto operators. In addition to being a cause of operator activity (see previous section), these events are also assumed to be a direct cause of any future events of the same kind if no new progress messages were read in between. This heuristic only applies in the case of delayed `PushProgressEvents` as shown in Fig. 4.2, and does not cause any errors.

Active operators. An operator which made progress (showed activity) is assumed to be a direct cause for any progress messages being sent by the worker until the same operator is scheduled again. In other words, the worker cannot be in a waiting state after an operator was active until all progress messages which are assumed to be caused by the operator's activity have been sent.

5.1.3 Wait-State Analysis Algorithm

By combining the knowledge of Timely's runtime behavior, particularly its static scheduling algorithm, together with the assumptions about operators and the heuristics to determine causal relationships, we can now define an algorithm to detect any waiting phases during a worker's execution history.

The algorithm processes the output of the preprocessing step described in Section 4.3, which is a stream of ordered program activities. Grouped by worker, this stream is processed sequentially. For each worker program activity, the algorithm checks whether the worker was in a waiting state during the execution of the activity. By Definition 5.1, this can only be if the activity did not perform useful work and if no direct cause for future useful work has already been observed in the stream prior to the activity in question. The latter part is tested by applying the cause/effect heuristics described in Section 5.1.2. Once a waiting state is detected, its starting timestamp is noted. Then, any program activities that occur and do not terminate the waiting state are discarded. Finally, if an activity causes the waiting state to be terminated, a *waiting* or *waiting for input* activity from the noted starting point to the time the waiting state was terminated is inserted into the stream, replacing all the discarded activities.

The conditions which need to be checked for each activity timestamp in the stream can be summarized as follows:

- Is an operator active?
- Are any (progress-) messages buffered?
- Do operators become active in the future as a result of current or past progress tracking activity?
- Does the progress tracking logic become active in the future as a result of current or past activities?

If any of these conditions apply, the worker is not waiting at the checked time. The first condition is obvious: if an operator is active, it is performing useful work, hence the worker cannot be waiting. Checking this condition is straightforward.

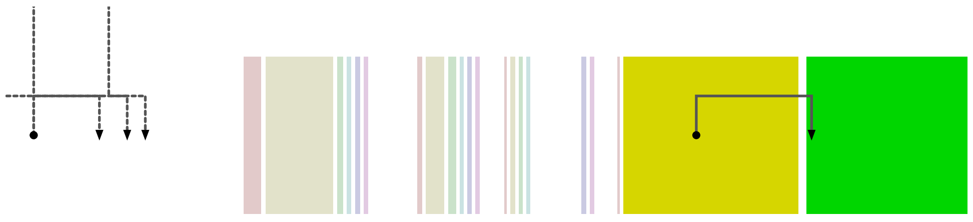
The second condition covers all the cause/effect heuristics in which the arrival of a (progress-) message is the direct cause of any progress tracking/-operator activity. If a cause/effect pair covers the checked timestamp, the worker cannot be waiting at that time according to our definition. From its arrival time until it is read, the message is buffered, hence the simple condition to check. Keep in mind that communication activities (progress- and data messages) occur in the stream at the time they arrived in the target buffer. Thus, when the algorithm sees a communication activity, it also knows when the corresponding (progress-) message will be read by the worker, since this can be computed from an extra attribute of the communication activity, the *receive buffer time*. Therefore, the algorithm knows exactly for which period of time (progress-) messages are buffered.

The third condition covers the cause/effect heuristic which says that operators can become active as a result of newly available progress information. As we assume that operators do not defer their reaction to progress information, the algorithm only has to look ahead in the stream until the time *every* operator has been executed again to check whether any operator has shown activity after new progress information was read/pushed onto it. If this is the case, the worker cannot have been waiting until *after* the operator in question was executed. The lookahead is implemented by keeping a sliding window over the next executions of every operator in the computation.

The last condition is similar, but is concerned with progress tracking activity instead of operator activity. By looking at the possible causes of progress tracking activity listed in Section 5.1.2, we first observe that any such cause occurs at maximum one step of the computation earlier than its effect. In order to check the last condition, we thus also only have to look ahead in the trace until we have seen the execution of every operator in the computation once. Therefore, the sliding window mentioned above is sufficient to check the last condition as well. The condition is checked by testing whether the current program activity is responsible for any future progress tracking activity in the sliding window by applying the heuristics from Section 5.1.2.

Insertion of Waiting Activities

As mentioned above, whenever the termination of a waiting state is detected, a corresponding activity is created and put in place of all the discarded activities which occurred during the waiting phase. The type of this activity is determined by the type of activity which terminated the waiting phase. As described in the formal model, specifically Section 3.2, there are two types of waiting activities: *waiting* and *waiting for input*. As per definition, the wait-state analysis inserts a *waiting* activity if the waiting phase was terminated by the arrival of a communication activity, i.e. a progress/data message.


(a) Trace before the wait-state analysis. The yellow operator seems to become active with no observable cause after a spinning phase.

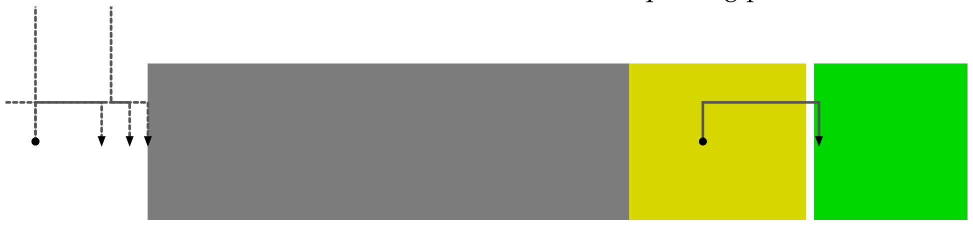

(b) A *waiting for input* activity replaces the spinning phase. The assumption is that the yellow operator must have read/generated input data.

Figure 5.4: The detection of a *waiting for input* activity.

On the other hand, it inserts a *waiting for input* activity if the waiting phase was terminated by an active operator, whose cause of activity could not be observed. In this case, the wait-state analysis simply assumes that the operator read external input from an unobserved source, and during the waiting phase this source was not ready to supply input yet, hence the waiting phase. This is illustrated in Fig. 5.4.

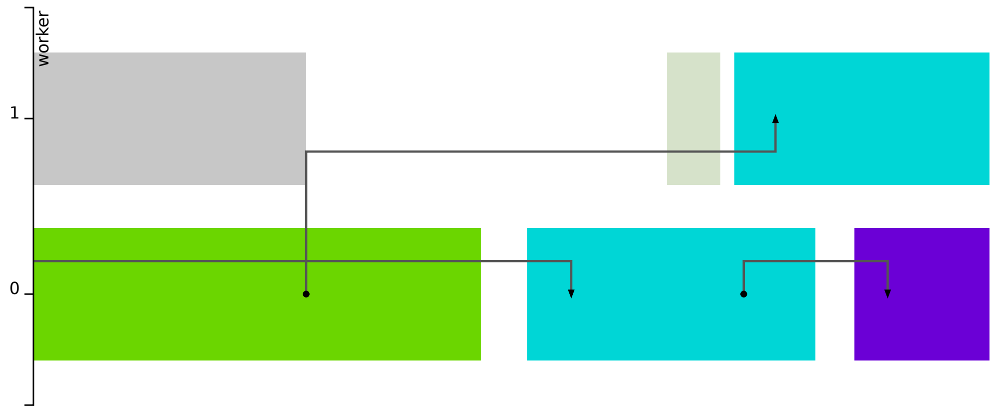
5.2 Activity Graph

After the preprocessing of the logs and the wait-state analysis, the resulting trace contains all the necessary information to form the activity graph as defined in Definition 3.3. However, to apply the critical path algorithm from the formal model, it is not actually necessary to explicitly transform the whole trace into an activity graph. Instead, the algorithm can be applied directly, with necessary changes to the trace being performed on the fly. Apart from reducing the overall computational complexity, this has the advantage that additional information, such as the receive buffer time of communication activities, is retained. It also allows for simple, easy to understand visualizations which do not have to be graph-based.

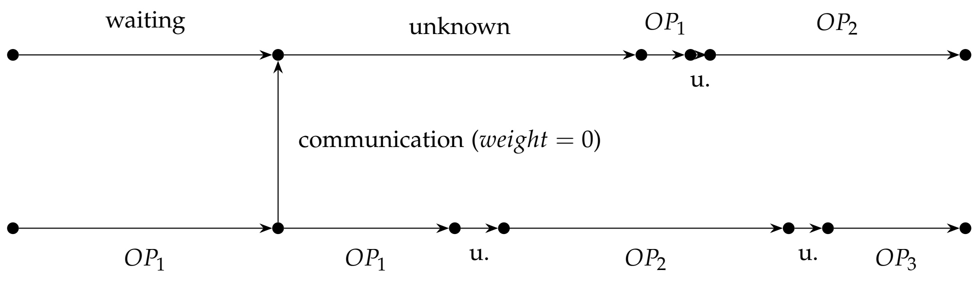
Figure 5.5 shows how an activity graph, in theory, would be constructed from an execution trace. The main difference between the activity graph view and trace view is that worker activities must be split at each event — such as at the arrival of a message — in the activity graph (otherwise it would not form a proper graph). Furthermore, worker-local messages do not need to be represented in the activity graph, since they are not considered to be communication activities in our formal model. Finally, *unknown* activities need to be explicitly represented such that each stream of worker activities forms a connected subgraph.

5.2.1 Trace Slicing

The trace can now be split into slices (snapshots) of a given duration. The slicing process itself is straightforward: any activities overlapping the slice boundaries are simply split into two parts. This is equivalent to applying the edge projection from Definition 3.4, and the resulting slices correspond to the activity graph snapshots in the formal model. One can of course also compute the critical path for the whole trace instead of slicing it into smaller snapshots, however, doing so is only possible if the entire trace is small enough to fit into the memory of the machine performing the critical path analysis.



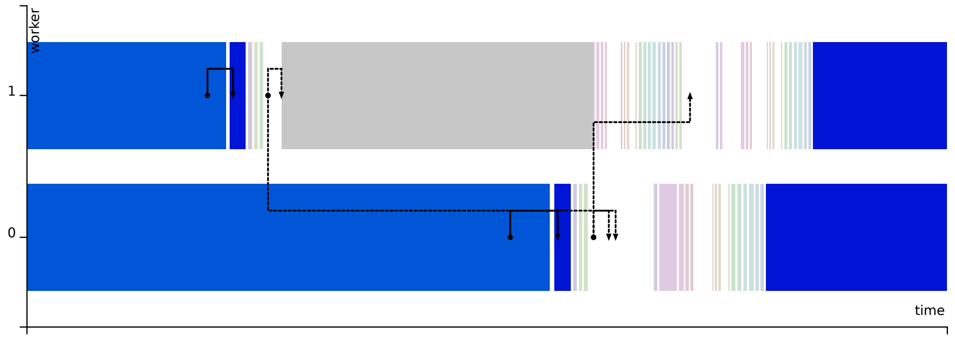
(a) Execution trace



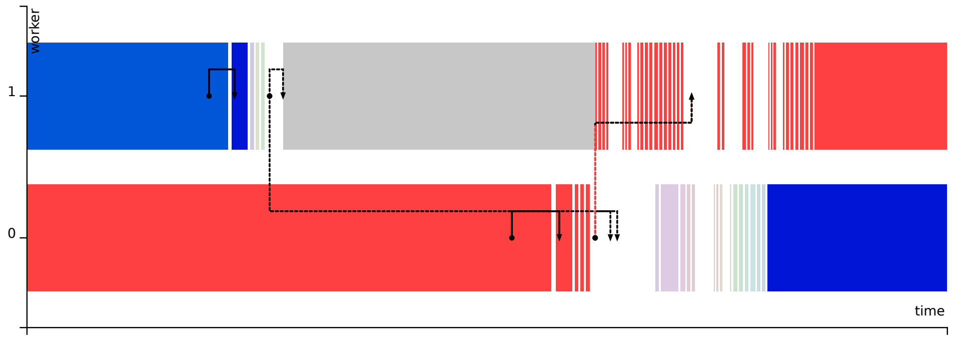
(b) Corresponding activity graph. The length of each edge corresponds to its weight, except for the communication activity. Note that the green data operation activity (OP_1), has been split at a message send event. Also note that worker-local messages and the buffering times of messages are not visible in this view.

Figure 5.5: Differences between trace view and activity graph view.

5.3 Critical Path Computation



(a) Trace excerpt after the wait-state analysis, before the critical path algorithm was executed.



(b) Trace excerpt showing the computed critical path (marked in red).

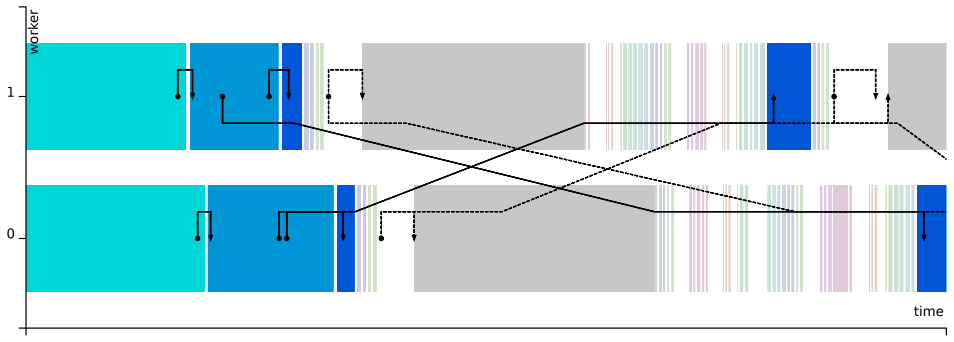
Figure 5.6: Illustration of the critical path computation for a small trace excerpt. The traces show the Differential Dataflow-based BFS implementation running on two worker threads in the same process.

In this section we discuss how the critical path algorithm from our model in Section 3.4 is applied to Timely Dataflow computations. Because the formal algorithm operates on an activity graph, which is not explicitly constructed in our implementation, the algorithm needs to be modified slightly. As discussed in Section 5.2, to construct a proper activity graph, the worker activities would need to be split at any communication events, worker-local messages would need to be removed and empty gaps between worker activities would need to be filled with *unknown* activities. It is clear where exactly unknown activities are in the trace, so they do not need to be added to the stream of activities even now — they can just be added by filling any gaps between activities of the critical path when actually needed, e.g.

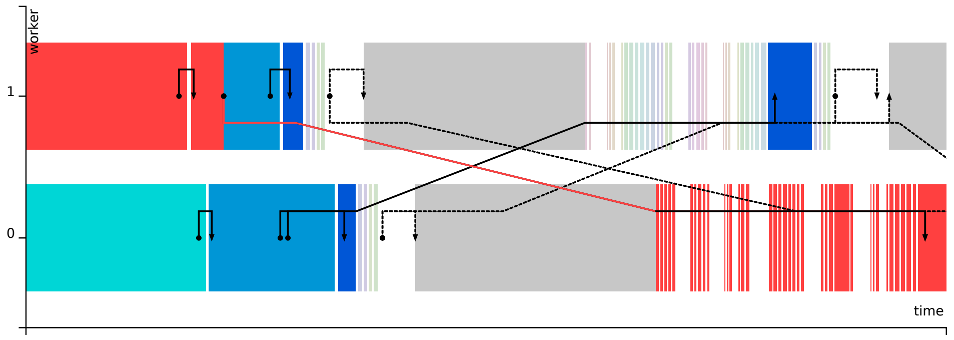
when computing a critical path profile per activity type. Worker-local messages are simply ignored entirely by the algorithm. Finally, worker activities only need to be split when they are just partially on the critical path. Otherwise, leaving them as single activities does not make a difference for our purposes. Thus, if the critical path follows along a communication activity whose start/end occurs during a worker activity, the worker activity is split on the fly by the algorithm and only the relevant part is added to the critical path. Any other worker activities are left untouched.

These slight modifications make the computation of the critical path more efficient, as less activities need to be represented. Inserting *unknown* activities would roughly double the number of worker activities in the trace given that our instrumentation does not cover every bit of code Timely executes, and there are tiny gaps between almost any two consecutively recorded worker activities. Furthermore, note that most of the example traces shown in this work were taken from computations with small input data sets (e.g. a BFS computation on a graph with a thousand nodes) and few active workers, in order to keep the illustrations simple and uncluttered. In more realistic computations with large input datasets and many workers, an operator can potentially send hundreds of messages per execution, which would require splitting the corresponding *data operation* activity into hundreds of parts, all containing similar information about the operator. Thus, these modifications were made mostly to make the analysis more efficient, to reach the goal of being able to find the critical path close to real-time.

Figure 5.6b shows an excerpt of the critical path (marked in red) of a Timely Dataflow computation. In this example, the algorithm started on worker one (remember that it processes the trace backwards), added all worker activities to the critical path until it encountered a waiting activity (gray), which can never be part of the critical path. At that point, as explained in our model, the algorithm's $E_{critical}$ set contains only the communication activity, a progress message in this case, which terminated the waiting state. Therefore, the computed critical path includes this communication activity. The algorithm then continues to add all worker activities on the source worker of said communication activity (worker zero) until the start of the trace. Note that worker-local messages are not marked in red, as they are ignored and not considered part of the critical path, as described above. Further note that the time a message/progress update spent in one of the receiving worker's buffers is not part of the corresponding communication activity's duration, as described in Section 4.3.2 and 4.1.3. Thus, this *receive buffer time* is also never part of the critical path.



(a) Trace excerpt after the wait-state analysis, before the critical path algorithm was executed.



(b) Trace excerpt showing the computed critical path (marked in red).

Figure 5.7: Critical path computation for a small trace excerpt. The traces show the Differential Dataflow-based BFS implementation running on two machines, with network communication.

A different example, shown in Fig. 5.7, shows the same computation but executed using two separate processes which are communicating over a network. This example demonstrates how two workers can be waiting at the same time while messages are being transferred over the network. Note also that the workers can be waiting during the time a message is still in a send buffer. This is no mistake; keep in mind that the messages are being sent by dedicated sender threads — the worker thread itself can thus already be waiting at the time a message is actually written to a network socket by the sender thread. Further note that this example also illustrates how activities are split during the critical path computation, as can be seen on worker one when the critical path follows along a communication activity. The worker activity during which the message in question was sent is split at the point in time when the message was put into the sender queue, as only the part of the activity prior to this event belongs to the critical path.

5.3.1 Preference for Worker Activities

One additional modification of the algorithm described in Section 3.4 we implemented is that only communication activities which terminated waiting activities are ever added to the critical path. In other words, if the algorithm encounters the arrival of a communication activity, but said communication activity arrived not during a time in which Timely was waiting (spinning), but for example during the time an operator did useful work, then it will never be on the critical path. The reason is that such a message spent some time in a receive buffer until it was actually read. Thus, a delayed message arrival would only decrease the time it spent in the buffer and not increase the overall runtime of the computation, unless the delay was longer than the message's receive buffer time. This observation, originally made by Schulz [36], means that any such communication activity should not be on the critical path.

Therefore, our algorithm preferably adds worker activities to the critical path instead of communication activities, unless there is no other choice (which happens when a waiting activity is encountered). This is visible in Fig. 5.7, where the algorithm ignores an arriving progress message on worker zero, and instead is following the worker activities until it encounters a data message which actually terminated a waiting state. Note that this approach never produces provably wrong results given the level of instrumentation we use, even in the unlikely case of a message with a zero receive buffer time, i.e. a message which was read immediately after its arrival. Such a case would be ambiguous, as the worker activity immediately following the message arrival could depend both on the message as well as the previous worker activity. However, in the absence of additional information, always following the worker activities by default in such cases is never incorrect.

Further note that because Timely does not block while waiting for messages to arrive and instead is spinning and polling all the message queues, even messages which terminated waiting states do have an above-zero receive buffer time, as they arrived in a queue some time before said queue was polled again. However, keep in mind that the communication activity's end point is at the time of its arrival, not the time it was read from the queue. Therefore, any remaining spinning the worker does until the message is read is part of the critical path, which is the most accurate way of modeling this behavior. If the duration of the remaining spinning phase would be reduced — e.g. using a scheduler which immediately schedules the operator which received the message — the overall runtime would improve, which is why it should on the critical path.

Chapter 6

Implementation

A critical path analysis prototype for Timely Dataflow computations was implemented as part of this work. Since much of the analysis is concerned with processing separate streams of instrumentation log events created by the workers of the reference computation, most of the workload is highly data-parallel. In order to make use of this inherent data-parallelism, the analysis itself was also implemented as a Timely Dataflow computation.

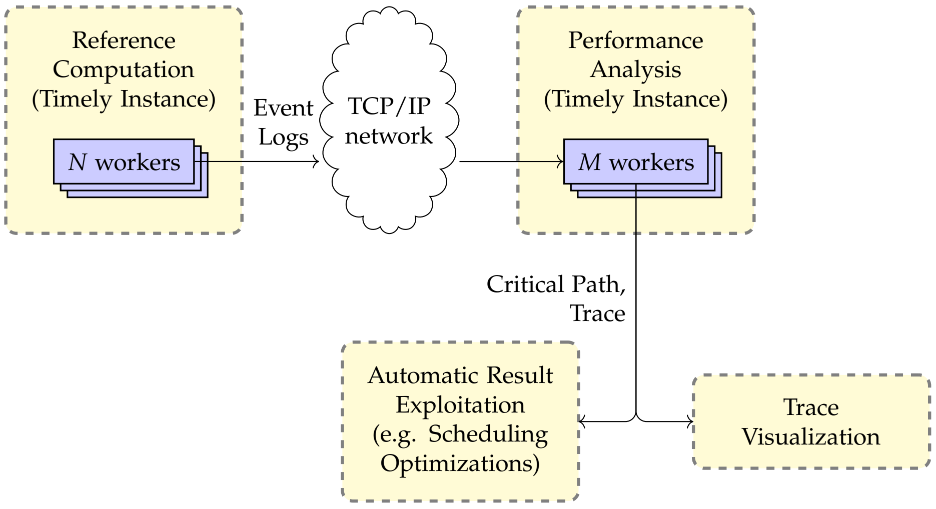

Reference Computation (Timely Instance)
N workers
Event Logs
TCP/IP network
Performance Analysis (Timely Instance)
M workers
Critical Path, Trace
Automatic Result Exploitation (e.g. Scheduling Optimizations)
Trace Visualization

Figure 6.1: System architecture. Note that the trace- and critical path data is currently stored to disk for the visualization.

Figure 6.1 shows the architecture of the complete system. The *N* workers of the reference computation feed the event logs created by the instrumentation directly into a distinct Timely Dataflow computation which runs the performance analysis live in (near) real-time. The performance analysis can run on a different machine than the reference computation and receive the logs over a network. As the parallelization of the performance analysis relies on the fact that event streams from different workers can be processed largely in parallel, it can make use of up to N workers as well ($M \leq N$). The results of the performance analysis for a particular trace slice can then be exploited directly, for example to optimize operator scheduling in the reference computation. Furthermore, the results can also be stored to disk for later (manual) analysis, e.g. using the visualization tool described in Section 6.2.

While the prototype is capable of doing the critical path analysis in real-time or near real-time (depending on the characteristics of the original computation), Timely Dataflow's logging facility is not yet capable of sending logs directly over the network, which is why the prototype needs to read the logs from disk[1]. The automatic exploitation of results is part of future work.

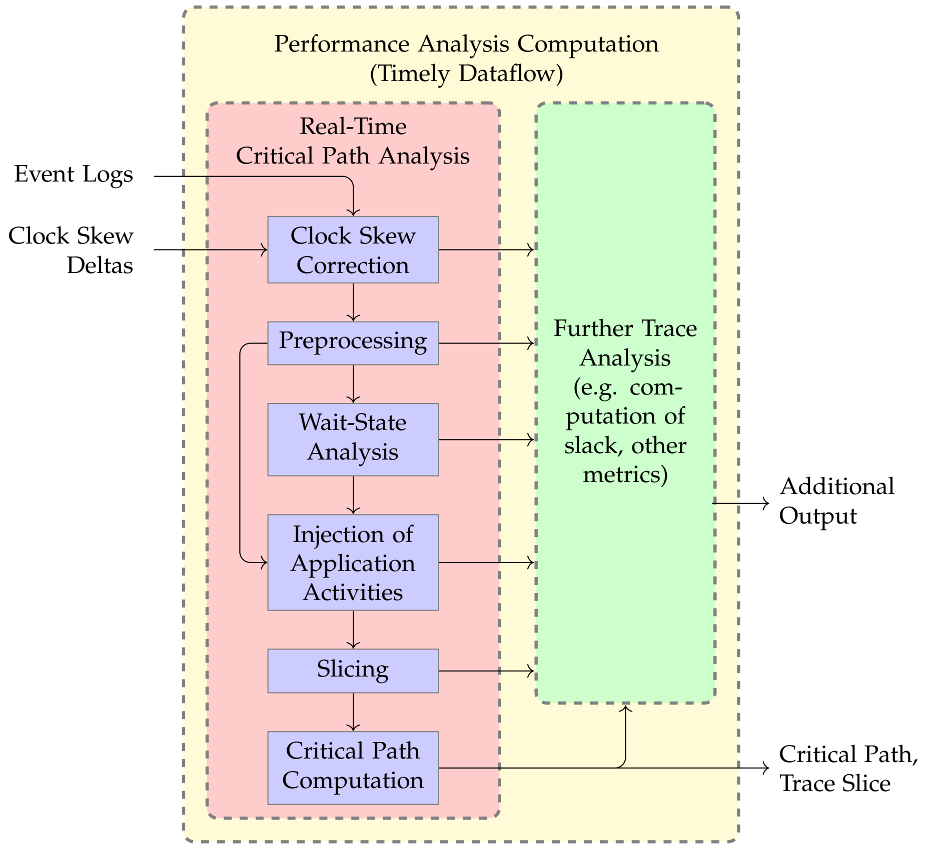


Figure 6.2: Simplified dataflow graph of the performance analysis computation.

[1]Of course it is possible to read/write the traces from/to network storage.

A coarse overview over the stages in the performance analysis can be seen in Fig. 6.2. The red section marks the critical path analysis which was implemented in this work. The green section indicates that any future additions to this work, like the computation of additional metrics based on the critical path or the trace in general, can be plugged in easily into the computation as a result of implementing the analysis as a Timely computation. Such downstream operators can make use of any of the intermediate results of the critical path analysis.

6.1 Stages of the Critical Path Analysis

Conceptually, the different stages of the computation are already described in Chapter 4 and Chapter 5. This section focuses purely on implementation details. Overall, the implementation is focused on performance, and therefore tries to minimize the amount of data which needs to be exchanged between workers. It also avoids sorting/re-ordering the log events (and later activities), instead making use of the fact that the workers of the reference computation are producing log events with monotonically increasing timestamps already. Most of the operators take care to not disturb this ordering on a per-worker basis.

The first stage corrects the *clock skew* of all the logs received. The assumption is that on each machine of the reference computation, only one Timely process is running (which is the best configuration to maximize performance). Therefore, the user has to supply the clock skew deltas for each process measured using the tool described in Section 4.3.1. The advantage of implementing clock skew correction as a Timely operator is that it can be easily distributed, even if multiple workers are receiving event logs and clock skew information. However, it also makes it difficult to associate log events to Timely epochs based on their timestamps, as the accurate timestamp of a log event is not known at the time it is supplied to the computation, but only at a later stage in the dataflow graph[2]. If the reference computation only consists of one process (with multiple worker threads), the clock skew correction stage is omitted.

The *preprocessing* stage handles the creation of program activities from the raw log events. This mostly involves a series of join-like operators for the different types of log events (see Section 4.2). At the end of this stage, the various streams of different types of program activities are merged, in order, into a single stream of program activities partitioned by worker ID. Grouped

[2]Note that putting each event into a separate Timely epoch would be infeasible anyway, as early tests have shown that this would lead to a prohibitively large increase in the cost of Timely's progress tracking. However, a more coarse-grained approach would still be possible.

by worker ID, each sub-stream consists of program activities sorted by their *end* timestamp. Communication activities are associated to the sub-stream based on the worker ID of their destination worker, hence they will appear in the destination worker's sub-stream at the time they were received.

The *wait-state analysis* operates on individual, per-worker sub-streams as described above, and is therefore completely parallelized. As the communication activities appear at the point they were received in the per-worker sub-stream, it can detect the termination of waiting states easily, without exchanging any data. As the wait-state analysis needs to buffer some activities to be able to look ahead in the stream, it introduces a slight delay. Also, waiting activities are only inserted when the wait-state analysis has seen the activity terminating it (e.g. a communication activity or an active operator), which can introduce an additional delay equal to the duration of the waiting activity.

Optionally, if the reference computation includes application-defined instrumentation, the application-defined activities, which are also created in the preprocessing stage, are injected into the stream of program activities after the wait-state analysis. As application activities can be used to provide more precise performance data, e.g. about different parts of an operator, they take precedence over any other worker activity type excluding waiting activities. This means that if a worker activity overlaps with an application activity, the part of the worker activity which is covered is replaced by the application activity. The same process is also applied for nested application activities; in this case, activities with a higher nesting level take precedence. The reason application activities are injected into the stream at this stage, and not upstream in the computation, is because the wait-state analysis relies on certain characteristics of the activity stream which are based on Timely's runtime behavior. For example, it needs the complete *data operation* activities denoting the operator executions. If application activities were injected earlier, it could be that such an operator execution is not observable in the stream anymore if an application activity happens to cover it.

The slicing stage does exactly what its name suggests: it splits the stream of activities to form slices, on which the critical path algorithm can then be run. This allows partial results to be provided in near real-time, while the reference computation is still running (remember that the critical path algorithm needs to start from the end of a trace slice, thus it can only be executed after the slice is available completely). It also computes the minimum timestamp in each slice and subtracts it from all timestamps in the slice. This stage is currently not parallelized in our prototype, and might be a bottleneck of the whole computation.

The final stage runs the critical path algorithm on an individual slice. As the backtracking-style critical path algorithm cannot be easily parallelized, the partitioning in this stage is done by slice number, i.e. each slice is assigned to a different worker. The computation of a critical path for a specific slice is therefore done by only one worker. However, keep in mind that the backtracking algorithm directly follows along the edges of the critical path, therefore the performance does only depend on the number of activities on the critical path (which is bounded by the maximum number of events for a worker), not the total number of activities in the trace. Thus, computationally the algorithm scales well as the number of workers are increased. The drawback is that the entire slice needs to fit into the memory of a single machine, which limits the maximum slice size for very large-scale computations.

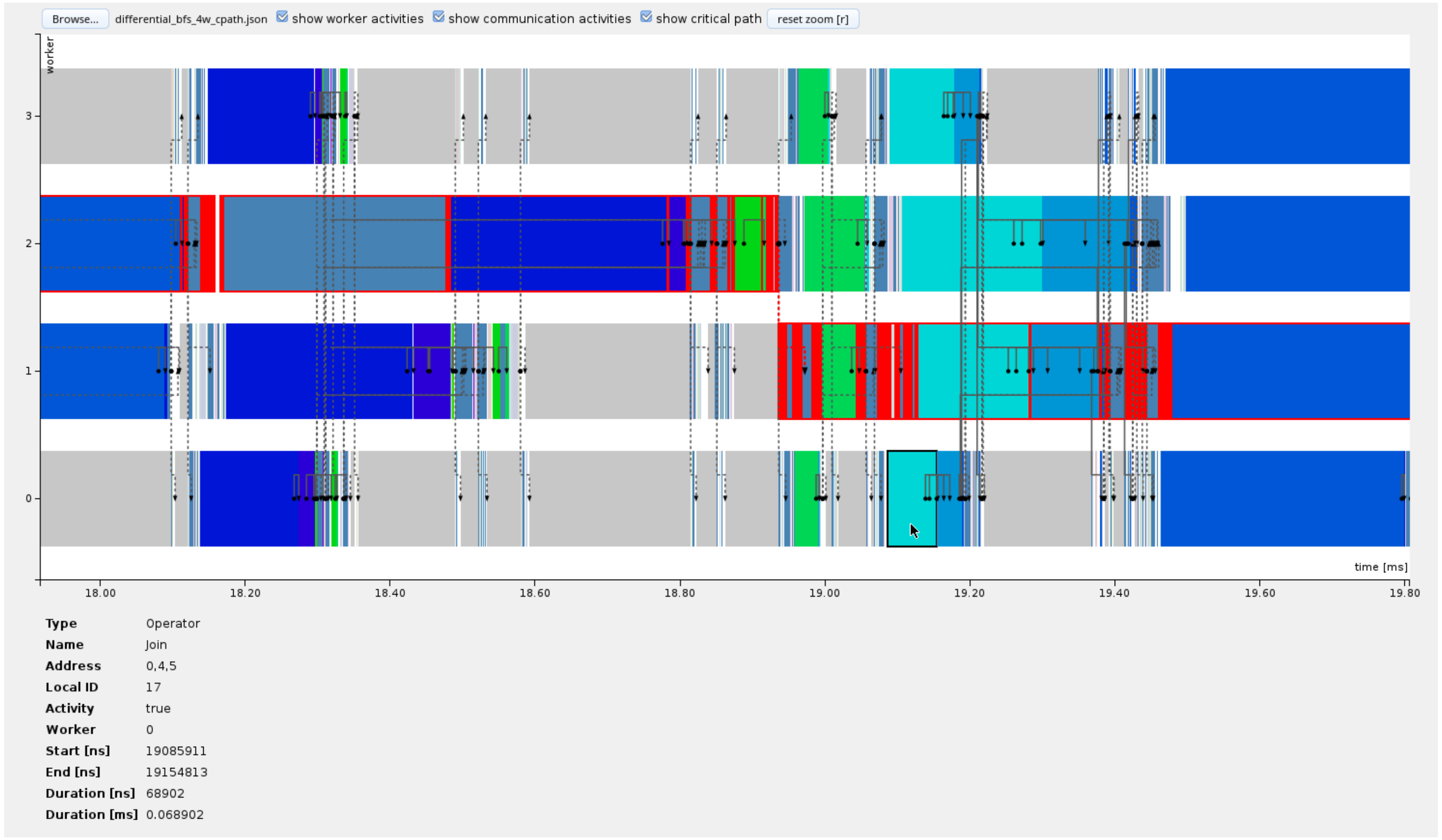
6.2 Visualization

To facilitate the analysis of the results of a critical path analysis, an interactive trace visualization tool was also developed as part of this work. It allows the inspection of both raw traces (which were not modified by the wait-state analysis) as well as the post-analysis traces which include waiting activities as well as the complete critical path. Furthermore, it can also be used to inspect the details of all activities in the trace, including extra attributes of certain activities, such as the send/receive buffer time of messages.

The web-based visualization was implemented using JavaScript and the Data-Driven Documents (D3.js) [3] library. The trace view is based on Scalable Vector Graphics (SVG), which adds the benefit of being able to export trace visualizations in a standardized vector graphics format. As SVG is natively supported by the browser itself, zooming in on the the trace is very efficient. The visualization tool is purely client-side, i.e. it does not require a server component to run.

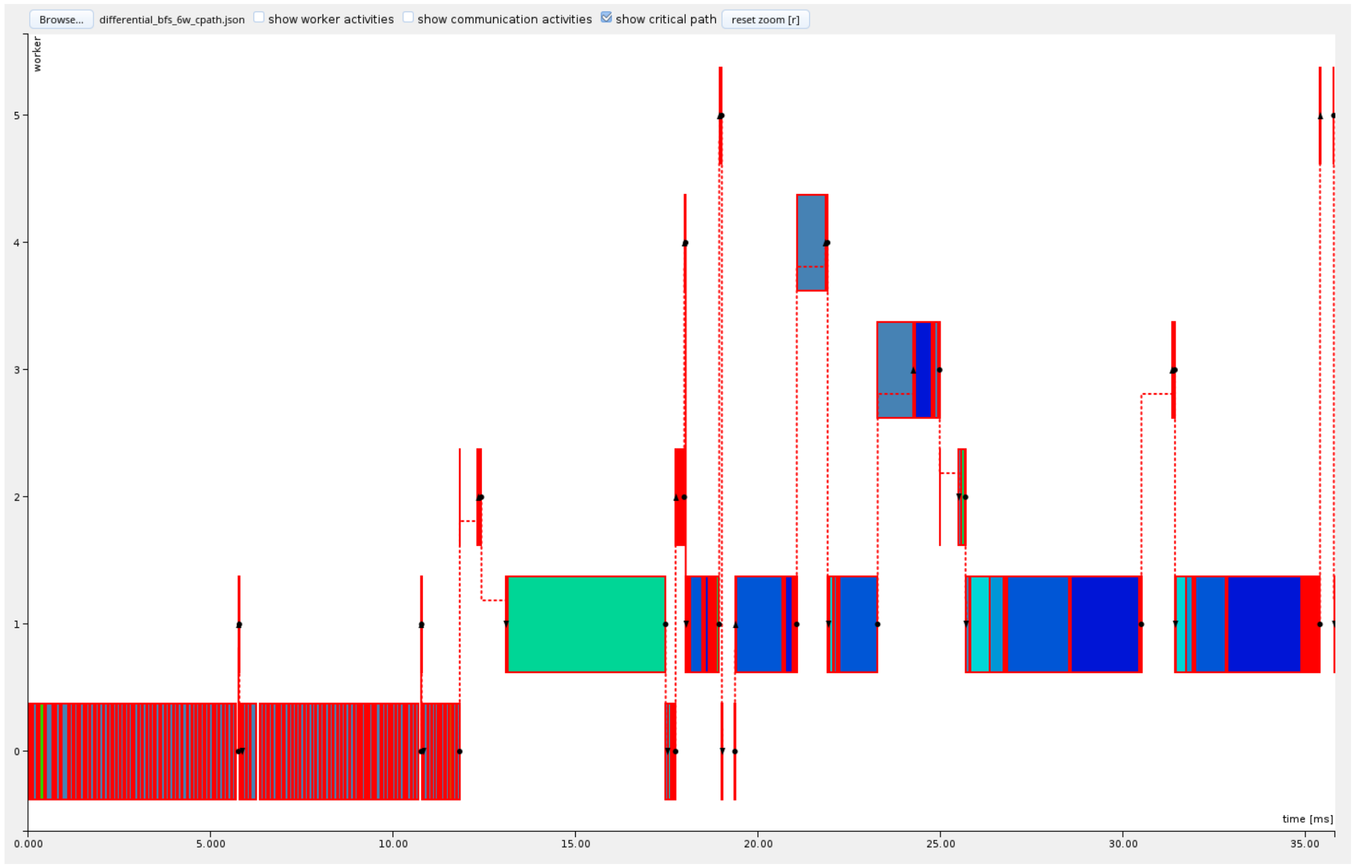
In order to visualize a trace, the Timely-based analysis program can be instructed to write the trace output to disk as one JSON file per slice, which can then be loaded into the visualization tool.

Screenshots of the visualization tool are depicted in Fig. 6.3 and Fig. 6.4.



Type	Operator
Name	Join
Address	0,4,5
Local ID	17
Activity	true
Worker	0
Start [ns]	19085911
End [ns]	19154813
Duration [ns]	68902
Duration [ms]	0.068902

Figure 6.3: Screenshot of the interactive trace visualization tool. The critical activities are marked with a red border.



Browse... differential_bfs_6w_cpath.json [ ] show worker activities [ ] show communication activities [✓] show critical path reset zoom [r]

Figure 6.4: View of the complete critical path of a Differential Dataflow-based BFS computation, as depicted by the visualization tool. Notice the graph generation phase at the beginning on worker zero (up to about 12ms).

Chapter 7

Evaluation

In this chapter, we evaluate both the performance of the critical path analysis itself as well as the usefulness of its results to find bottlenecks in Timely computations. All the experiments were run on a single machine with four octa-core 2.4GHz Intel Xeon E5-4640 processors (32 CPU cores in total) and 500GB of memory.

Section 7.1 discusses the overhead of Timely's instrumentation for two different kinds of computations. In Section 7.2, we analyze the performance and scalability of the critical path analysis itself and demonstrate that its throughput is large enough to be able to compute the critical path in real-time in certain configurations. Finally, we discuss how critical path analysis can be used to find the factors limiting the performance and scalability of a Differential Dataflow computation in Section 7.3.

7.1 Overhead of Instrumentation

We measured the overhead of the event logging for two reference computations with different characteristics. The first computation, the Differential Dataflow-based BFS implementation (see Fig. 2.3), is a fast-stepping computation whose dataflow graph consists of 18 operators. The operators are mostly "thin" operators, i.e. operators which do not keep running for a long time. The computation needs to be stepped often (many executions of each operator) for the computation to make progress, as the operators need to be able to exchange data with each other.

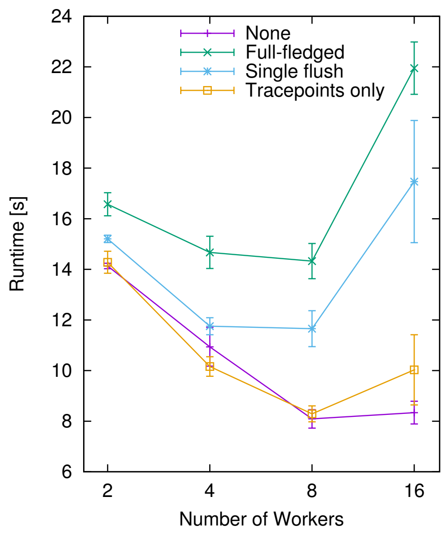
The second computation is the Timely-based BFS implementation (see Fig. 2.1). Computations based on pure Timely Dataflow tend to contain more "fat" operators with very specific purposes. In this case, the computation only consists of 5 operators. The computation does not need to be stepped as often, as most of the work is done by just a handful of operators (mostly by the "BFS" operator itself). Thus, it is a more slow-stepping computation.

Both computations were run on random graphs with 5 million nodes and 50 million edges. As a baseline, they were run with disabled instrumentation/logging. This run was compared to runs with full-fledged instrumentation using three different logging facility configurations. The first configuration uses the standard logging facility which flushes the log buffer to disk after every step of the computation. Additionally, we tested a configuration in which the log buffer is only flushed at the end of the computation (single flush). Finally, we tested a configuration in which the log data is kept in memory and never written to disk, to measure the overhead of only the tracepoints themselves. Obviously, the last two configurations are not feasible for long-running computations as they cause the memory to be filled up. However, the results give an indication of how much a smarter logging architecture could reduce the overhead.

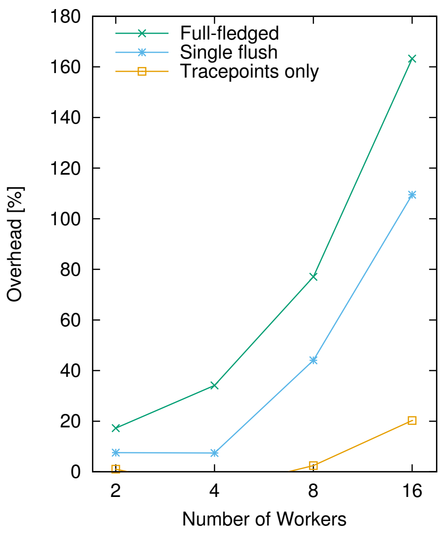
We expect the instrumentation overhead for the fast-stepping computation to be larger than for the slow-stepping one. The reason is that the fast-stepping computation generates more log events in a given period of time, as there are more operator executions. Moreover, as the dataflow graph contains more operators and more channels, we expect a higher number of messages being exchanged, which also adds overhead. Also, we expect that the log flush at every step will be delaying the fast-stepping computation by a larger degree, as it needs to perform more steps in order to make progress. Finally, we expect that the overhead will get larger with more workers in the computation, as more log events will be generated in a shorter time, and as result the disk to which the logs are written will become a bottleneck.

7.1.1 Results and Discussion

Figure 7.1 shows the results for the fast-stepping computation (Differential-BFS). Each data point shows a mean over 50 experiments. The error bars indicate the sample standard deviation. Clearly, the standard full-fledged instrumentation with a log flush after every step has the highest overhead. Comparatively, the single log flush at the end of the computation is more efficient. Note however that this method also has the drawback that more log events are generated in total, since workers which are spinning constantly generate log events, and the spinning rate (and thus the log event rate) is not limited by the disk throughput. The tracepoints only, without I/O, have a negligible (1-2.5%) overhead on 2-8 workers. On 16 workers, the overhead increases to 20%. One possible explanation for this increase could be that memory management becomes a bottleneck at this point, given that the log buffers are growing very quickly with so many workers generating log events.

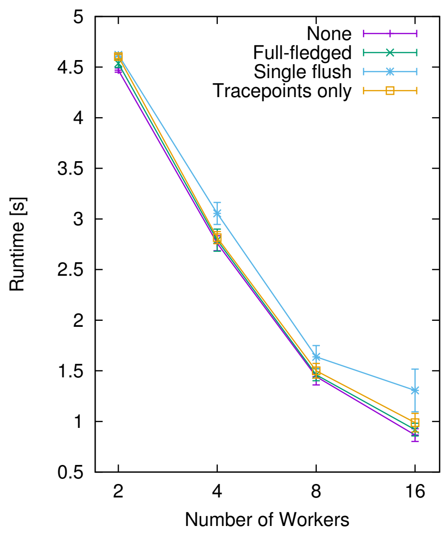


(a) Runtime of different instrumentation configurations. Error bars depict the sample standard deviation.

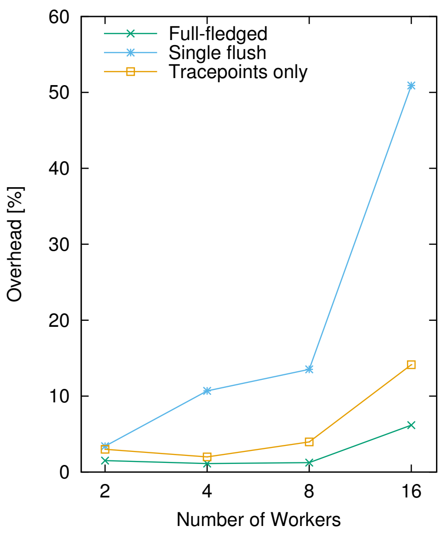


(b) Overhead compared to baseline computation without instrumentation/logging.

Figure 7.1: Instrumentation overhead for the fast-stepping computation (Differential-BFS).



(a) Runtime of different instrumentation configurations. Error bars depict the sample standard deviation.



(b) Overhead compared to baseline computation without instrumentation/logging.

Figure 7.2: Instrumentation overhead for the slow-stepping computation with "fat" operators (Timely-BFS).

Figure 7.2 shows the results for the slow-stepping computation (Timely-BFS). As expected, the overhead is generally much lower compared to the fast-stepping computation in all tested configurations. Interestingly, the standard configuration seems to result in a lower overhead than the single-flush or tracepoints-only configurations for this computation. The most likely explanation to this is that, as mentioned above, in these configurations workers generate vastly higher numbers of log events during spinning phases. Compared to the fast-spinning computation however, the delay caused by a log flush at each step has less impact on the total runtime because the computation is not stepped as often.

For suggestions on how to improve the instrumentation and Timely's logging facility, refer to Chapter 8.

7.2 Performance of the Critical Path Analysis

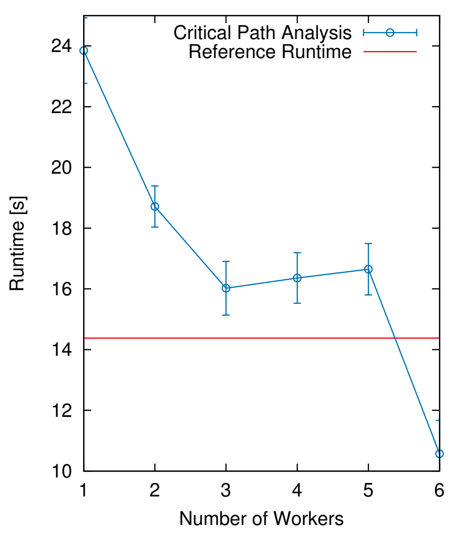
To test the performance of the critical path analysis itself, we used the same two programs — the fast-stepping Differential-BFS and the slow-stepping Timely-BFS — as reference computations. The analysis was configured such that it splits the trace into 1-second slices and computes the critical path for each slice.

We ran the Differential-BFS computation on 6 workers on a random graph with 5 million nodes and 50 million edges to generate an execution trace. We then measured the runtime of the critical path analysis in different worker configurations. The whole experiment was repeated 10 times.

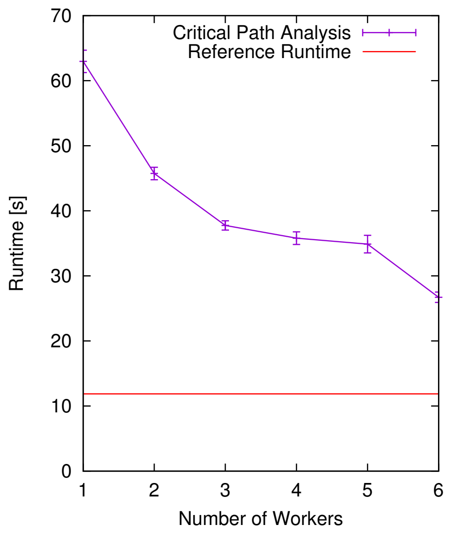
Figure 7.3 shows the resulting runtime on 1-6 workers. Error bars indicate the sample standard deviation. Fig. 7.3a shows the analysis of traces which were recorded using the standard logging infrastructure using regular disk flushes, whereas Fig. 7.3b shows the analysis of traces which were kept in memory and written to disk at the end of the computation. The red lines show the mean runtimes of the reference computations.

The data in Fig. 7.3a shows that the critical path analysis is running faster than the reference computation when using 6 workers. Thus, it is possible to perform it in real-time.

The the lack of improvement on 4-5 workers compared to 3 workers can be attributed to load imbalances occurring in these worker configurations. Remember that most of the parallelization comes from the fact that it is possible to process logs streams from different workers of the reference computation in parallel. As there are 6 workers running the reference computation, it is possible to distribute the load evenly onto 1, 2, 3 or 6 workers as these numbers are divisors of 6. As 4 and 5 are not divisors of 6, there is a load imbalance in those configurations, which explains the lack of improve-



(a) Analysis of disk-recorded traces.



(b) Analysis of memory-recorded traces.

Figure 7.3: Critical path analysis runtime for Differential-BFS traces. The red lines indicate the mean runtimes of the reference computations, which used 6 workers.

ment compared to the 3-worker configuration. For maximum efficiency, it is therefore advisable to always select the number of workers for the analysis computation to be a divisor of the number of workers in the reference computation.

On the other hand, when keeping the logs in memory during the reference computation to reduce overhead, the critical path analysis of the resulting trace is much slower, as Fig. 7.3b shows. Table 7.1 shows the average number of activities in the generated (unprocessed) traces. Clearly, the increase in runtime is caused by the much higher number of activities/log events

Activity Type	**Avg. Count, Disk-recorded Trace**	**Avg. Count, Memory-recorded Trace**
Active Operator	392,876	358,757
Inactive Operator	14,049,106	65,928,959
Message	166,613	166,613
Progress Message	842,286	834,730
Total	**15,450,881**	**67,289,059**

Table 7.1: Average number of activities before the wait-state analysis in the Differential-BFS traces.

generated when memory-based log recording is used. The higher number of total events is caused solely by the increase in inactive operator activities. As mentioned in Section 7.1, when the log event rate is not limited by disk throughput, more such activities are generated during the spinning phases. Note that the number of active operators and progress messages are even *lower* than in the disk-recorded traces, while the number of messages stays constant.

This result clearly indicates that reducing the number of inactive operator activities would lead to a substantial performance improvement. Approaches to do so are discussed in Chapter 8. Note also that the memory-based tracing approach gives an upper bound on the number of log events generated if the Timely's logging facility was implemented more efficiently. Thus, if the logging facility is to be optimized, reducing the number of inactive operator events should also be given a high priority.

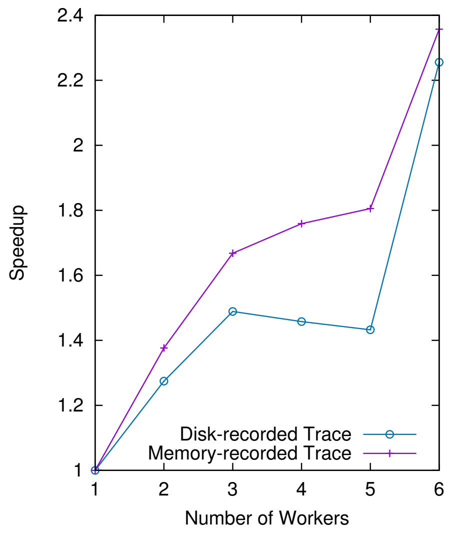
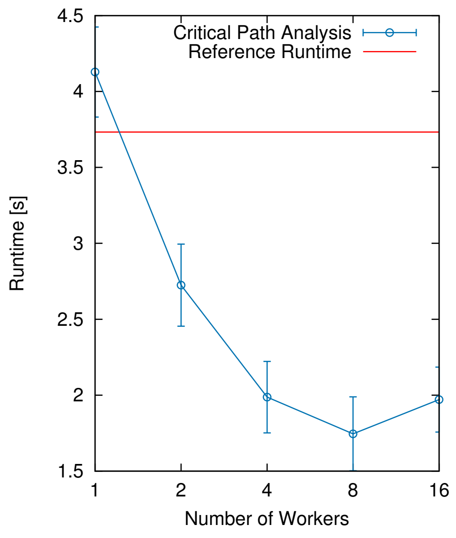


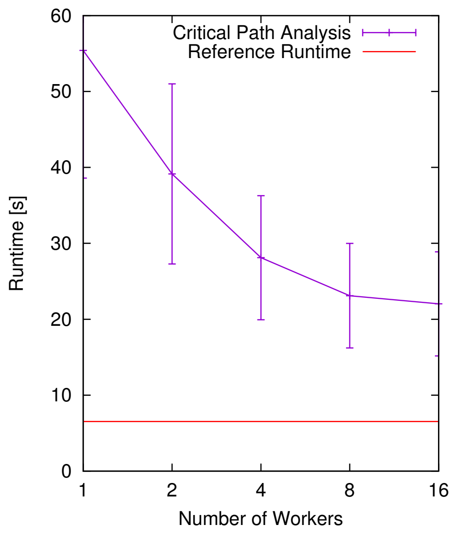
Figure 7.4: Critical path analysis speedup for Differential-BFS traces.

Figure 7.4 shows the speedup for the analysis of both types of traces. Again, the lack of improvement on 4-5 workers is due to the load imablance explained above.

Compared to the Differential-BFS implementation, the Timely-BFS implementation is considerably more efficient. Hence, we used larger graphs, consisting of 20 million nodes and 200 million edges, as inputs for the following experiments. Also, the reference computation was run on 16 workers instead of just 6.



(a) Analysis of disk-recorded traces.



(b) Analysis of memory-recorded traces.

Figure 7.5: Critical path analysis runtime for Timely-BFS traces. The red lines indicate the mean runtimes of the reference computations, which used 16 workers.

The results, shown in Fig. 7.5, follow a similar pattern. As the Timely-BFS implementation is slow-spinning, it generates less log events. Thus, the critical path analysis running on just two workers is already faster than the reference computation on 16 workers. However, Fig. 7.5b shows that just like for the Differential-BFS computation, the analysis of the memory-recorded traces is much slower. Again, this is explained by the increased number of log events which need to be processed, as shown in Table 7.2. While in this case, the number of active operators and progress messages increased as well, the absolute numbers are still very low. The increase

Activity Type	**Avg. Count, Disk-recorded Trace**	**Avg. Count, Memory-recorded Trace**
Active Operator	15,680	34,908
Inactive Operator	421,202	58,966,272
Message	587,689	587,689
Progress Message	232,205	540,018
Total	**1,256,776**	**60,128,887**

Table 7.2: Average number of activities before the wait-state analysis in the Timely-BFS traces.

in those activities is negligible compared to the almost 140-fold increase in inactive operator activities, which is clearly the reason for the runtime increase. Note that the high variance in the critical path analysis runtime is explained by the high variance in the number of log events generated by the Timely-BFS computation.

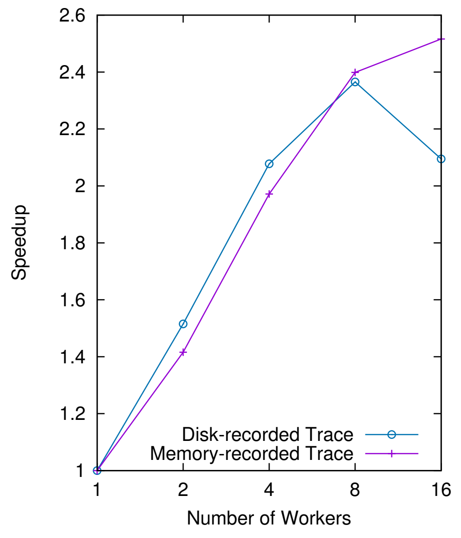


Figure 7.6: Critical path analysis speedup for Timely-BFS traces.

The speedup of the analysis is shown in Fig. 7.6. While the number of events has a large impact on the runtime, it is clear from the speedup plot that it does not affect the scalability, as the increased number of events are still processed in parallel. Given the relatively low number of events in the normal traces, the speedup decrease from 8 to 16 workers could be explained by the comparatively high overhead introduced by the added workers. Also, keep in mind that the machine on which these experiments were run has 4 octa-core processors. Hence, NUMA effects could play a role once more than 8 threads (and thus at least two physical processors) are used. As the analysis of the memory-recorded trace is dominated by the processing of inactive operator events/activities, a task which requires no data exchange between the workers, it is possible that it is less affected by such effects compared to the analysis of the disk-recorded trace.

It is important to note that stages that are placed after the wait-state analysis in the dataflow are largely unaffected by the increased number of inactive operator activities, as most of those are replaced by waiting activities during the wait-state analysis. This is of particular importance to the slicing stage, as it is currently not parallelized.

Note that for both the Differential-BFS and Timely-BFS computations, the speedup is not ideal. The most likely bottleneck is the reading of the input trace; as the traces are currently read from disk, the constant disk throughput puts a limit on the scalability. Another potential bottleneck could be the slicing stage.

7.3 Analyzing the Scalability of BFS

To evaluate the usefulness of critical path analysis for Timely Dataflow computations, we performed an analysis of the Differential Dataflow-based BFS computation to ascertain the factor(s) limiting its scalability.

For this purpose, we ran the BFS computation on a random graph with 10 million nodes and 100 million edges and computed the critical path over the whole generated trace. The experiment was run in 2-, 4-, 8- and 16-worker configurations, and for each configuration we performed 10 repetitions. The instrumentation data was kept in memory during the BFS computation in order to avoid the distortions introduced by regular log buffer flushes to disk.

For each configuration, we computed the critical path profiles, i.e. the total duration on the critical path for each activity type, averaged over the 10 repetitions. From these profiles, we then selected the top 5 contributors (for each configuration individually), and aggregated all other activities into a separate group ("other activities").

The BFS implementation includes the generation of the random graph at the beginning of the computation. Since this is done outside the scope of any operator, we instrumented this part using application-defined activities. Furthermore, instrumentation was also added to cover the most important parts of Timely's progress tracking logic. This was also done using application activities [1].

7.3.1 Results and Discussion

The resulting mean critical path profiles can be seen in Fig. 7.7. We can see immediately that the graph generation takes almost the same time in each configuration, and does therefore not scale at all. This can be explained by the fact that the whole graph is generated by worker zero alone at the beginning of the computation. The other workers only absorb the generated edges during that time, i.e. they are not performing a lot of work. Thus, the graph

[1]Even though the progress tracking logic is technically a part of the Timely system, it can be instrumented using application-defined activities. This is a quick and simple way of extending the existing instrumentation.

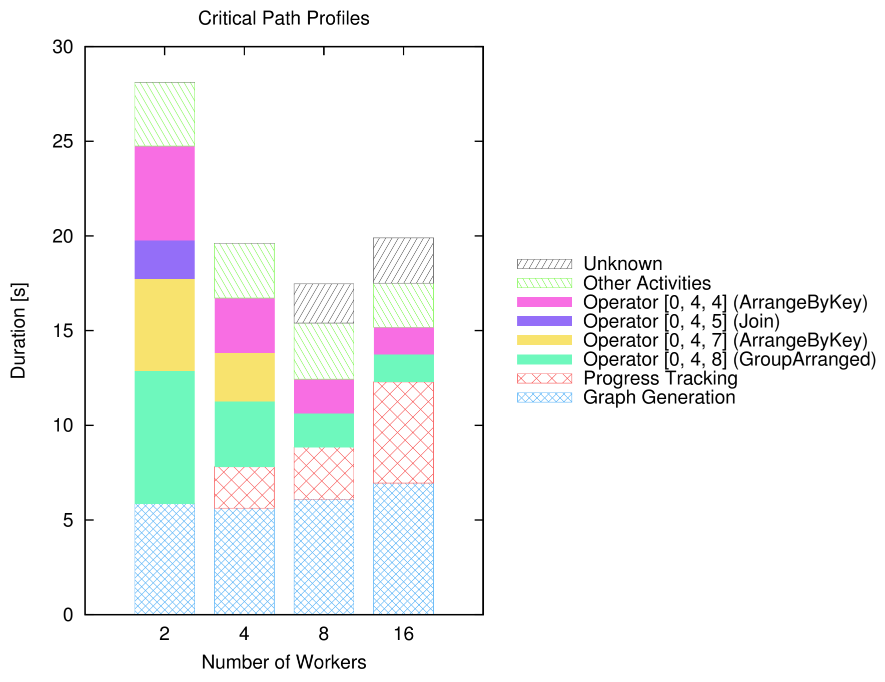


Figure 7.7: Critical path profiles showing the top five contributors in each configuration. The smaller contributors are grouped together in the "Other Activities" category. Note that the top five contributors change for the different configurations.

generation is usually almost entirely part of the critical path in this computation. This can be seen in Fig. 6.4, which shows the same computation but executed with a much smaller input graph size.

On the other hand, the plot shows that the critical path duration of the top contributing *operators* decreases as the number of workers is increased. For an operator/activity which scales perfectly, one would expect its duration to be approximately halved each time the number of workers is doubled. Clearly, the top contributing operators do scale, although not perfectly.

Apart from constant-duration graph generation, the limiting factor seems to be the progress tracking logic. Starting at four workers, it already belongs to the top five contributing activities. The duration progress tracking spent on the critical path increases as the number of workers increases. From 8 to 16 workers, the duration almost doubles, and as a result the whole computation has a longer runtime when using 16 workers than when using only 8. It is clear that progress tracking is the primary reason why the computation does not scale well. Note that small parts of the progress tracking implementation are still uninstrumented, and therefore part of the "Unknown" category. This could be the reason why the duration of unknown activities does also not decrease on 16 workers compared to 8.

7.3.2 Finding Activities to Optimize

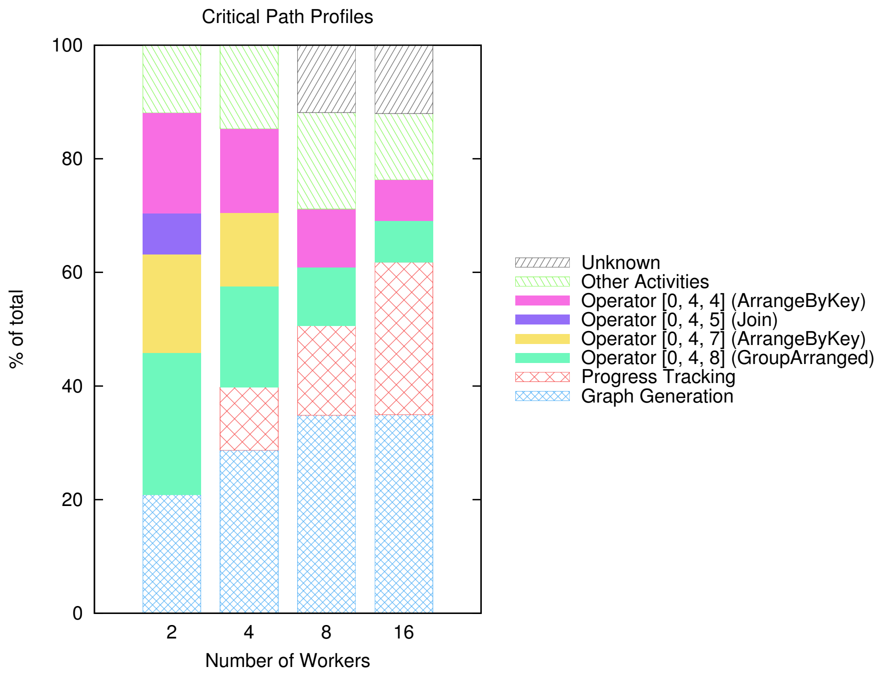


Figure 7.8: Critical path profiles showing the relative critical path contribution of the top five activity types.

Figure 7.8 shows the relative critical path contributions of each activity type instead of the absolute time. This plot is very useful to find activities to optimize. Assuming that the progress tracking cannot be optimized, the scalability is mostly limited by the graph generation. On 8 workers, it contributes to more than a third of the whole critical path and is thus by far the largest contributor. Hence, we can conclude that optimizing the graph generation by distributing it to multiple workers would substantially improve the performance and scalability of the computation, and should be given a high priority.

Chapter 8

Future Work

This chapter discusses areas on which future work efforts should be focused in order to further improve or extend the results of this work. Section 8.1 discusses how our model could be applied to other data-parallel systems. In Section 8.2, we provide suggestions for the extension of the performance analysis to provide more useful information than just the critical path. Section 8.3 discusses how a sampling method could be used to decrease the workload and logging overhead. Sections 8.4 and 8.5 discuss improvements to the instrumentation and the logging facility, respectively. In Section 8.6, ways to improve the accuracy of the current wait-state analysis (which is partially based on heuristics) are discussed. Finally, Section 8.7 discusses how the interactive trace visualization tool could be improved in the future.

8.1 Applying the Formal Model to Other Systems

The formal model described in Chapter 3 is intended to be generally applicable to modern data-parallel systems (and potentially other distributed systems). Therefore, a straightforward extension would be to instrument other systems, such as Spark [40], Flink [1, 21] or Storm [2, 37] accordingly and apply the methods discussed in this work to these systems.

Note that much of the actual performance analysis computation currently expects a Timely-specific log format. Furthermore, the wait-state analysis is specific to Timely Dataflow's runtime behavior, although many of the principles could also be applied to similar systems. Therefore, it would be advisable to develop a more generalized intermediate log format to facilitate an implementation for multiple systems. Moreover, a standardized interface for the wait-state analysis should be developed, such that implementations for different systems can be plugged in easily.

Keep in mind however that using an intermediate format for the logs might come with a performance penalty. However, this might be acceptable, especially if the real-time aspect of the analysis is not part of the use case.

8.2 Extending the Performance Analysis

As described in Chapter 6, the entire performance analysis is implemented as a Timely Dataflow computation. This makes it very easy to extend by adding additional downstream operators. Such operators could be used to compute useful metrics based on the trace or critical path data. An example of such a metric could be the slack of an activity, which is the time by which the activity's duration could be increased without it becoming part of the critical path, i.e. without causing an increase in the overall runtime of the computation. Another example would be the critical path imbalance indicator, as defined by Böhme et. al. [13, 12], which is a measure of the difference between an activity type's contribution to the critical path and the average time spent executing said activity type across all workers. Indubitably, new useful metrics based on the complete critical path could also be devised. These metrics could also be aggregated over multiple slices of the computation.

Furthermore, if the real-time aspect is not important for a particular use case, it might be beneficial to actually construct the proper activity graph. Doing so could simplify the computation of additional metrics. Also, Differential Dataflow's [4] higher-level interface could be used to implement additional analysis stages more quickly.

8.3 Applying a Sampling Method

To reduce the workload, a sampling method could be applied. Instead of computing the critical path for consecutive slices of the computation, one could sample a smaller number of slices, compute the critical path for those samples, and then aggregate the results.

In order to do so, the performance analysis could ditch a certain amount of the trace in the early stages. An even better solution would be if the instrumentation could be enabled/disabled during runtime, as this would also remove the overhead of logging from the reference computation while no sample is taken.

One difficulty would be to select the start- and end points of a sample, such that aggregate measures over multiple samples are meaningful. Ideally, the sample outline should be defined by the reference computation itself. This would make it possible to perform a critical path analysis of just the processing of individual queries submitted to a running computation, which would be useful to optimize query response time.

As an example in the context of Internet services, the Mystery Machine [17] is able to perform such an end-to-end query analysis and produces aggregated results over a large number of sampled traces.

8.4 Instrumentation Improvements

The most obvious instrumentation improvement is to add more of it. Currently, the default instrumentation is relatively coarse-grained. In the future, instrumentation could be added to cover all the activity types listed in Section 3.2. For example, neither buffer management nor serialization is currently covered with dedicated instrumentation.

8.4.1 Recording OS Scheduling Information

The current performance analysis does not receive any information about when a worker thread of the reference computation was preempted by the operating system's scheduler. If the instrumentation would include the tracing of OS scheduler events, the *idle* activity type (see Section 3.2) could be included in the analysis. Idle activities would not only improve the accuracy of the analysis, they could also enable the spotting of certain configuration problems. For example, if the computation is run with more (worker-) threads than CPU cores available on the machine, this could lead to massive latency issues. As a large percentage of the critical path would consist of idle activities, such problems could easily by spotted during the analysis. The tracing of the scheduler could be implemented on Linux using a tracing framework such as DTrace [15] or LTTng [18].

8.4.2 Reducing the Number of Schedule Events

As mentioned in Section 4.2, currently `ScheduleEvents` are recorded both for operators which performed useful work as well as for those which did not. As a consequence, a large amount of schedule events are recorded for operators which were running only briefly to poll their queues. These log events (as well as the resulting program activities) convey little useful information. If one excludes all such activities which will be replaced by waiting states, these inactive operator activities make up only a very small percentage of the overall runtime for computations with realistic input data sizes, as operators which are performing useful work run for an much longer time in comparison.

However, recording all schedule events adds a significant overhead, especially considering that they are also recorded when the worker is in a waiting phase, i.e. when it is spinning. During the spinning phases, such log events are produced at a much higher rate than during normal execution, but all of these events are later discarded by the wait-state analysis. Especially if multiple workers are spinning simultaneously, the recording of these events puts an enormous pressure on the disk. The resulting overhead in both the reference computation as well as the performance analysis is demonstrated by the experiments described in Section 7.1 and Section 7.2. It is therefore clear that reducing the number of schedule events for inactive operators has the potential to massively improve the performance of the analysis and to reduce the overhead of the event logging in the reference computation. Hence it should be given a high priority for future work.

One proposed solution would be to implement a dynamic scheduler for Timely, as described in Section 8.6.1. Operators which would not perform any work would then not be scheduled in the first place, hence eliminating all inactive operator schedule events. Alternatively, Timely's logging facility could filter out the needless events from the log buffers before flushing them. While this would add some computational cost, it would help to significantly reduce disk pressure.

Consider that the wait-state analysis in its current form makes use of inactive operator activities as it needs to keep executions of all operators, active or inactive, in its sliding window. However, for this purpose, the inactive operator activities are not essential. An alternative solution would be to record a single log event at each step of the computation, and use that event as a reference to define the boundaries of the sliding window.

8.4.3 Removing Sequence Numbers from Messages

Currently, both data- and progress messages contain sequence numbers. These sequence numbers are needed to correlate progress- or message events from the event logs. Each message sent over a communication channel, i.e. over a TCP connection, contains an additional sequence number. Since these channels are all ordered, a minor improvement to reduce network overhead would be to not include the sequence numbers in the message itself, but rather maintain them on both endpoints using simple counters.

8.4.4 Dynamic Instrumentation

When one is only interested in analyzing specific parts of the computation, switching to a more flexible dynamic instrumentation method would be beneficial in the future. Dynamic instrumentation would also facilitate the temporary insertion of more fine-grained tracepoints for certain parts of the program while searching for the precise root cause of a performance issue. These are some of the benefits a modern dynamic tracing framework, such as Pivot Tracing [27], Fay [20] or DTrace [15], would offer over the current, static instrumentation. The current solution also has the drawback of not being able to record any system calls. Thus, I/O activities are currently not registered. By inserting tracepoints at read/write system calls for example, such activities could be detected. This kind of tracing could easily be implemented using already existing tracing frameworks like the ones mentioned above.

An interesting further possibility would be to develop a cost model that can be used to adaptively enable/disable tracepoints in a dynamic fashion, such that both the instrumentation overhead as well as the performance of the critical path analysis is kept within acceptable (user-defined) bounds. The cost model could be based on measurements of the log event rate (which may vary over time), the memory- and disk bandwidth, and potentially other factors.

8.5 Logging Facility Improvements

Assuming the logging facility used to record all trace events is not replaced by an external, dynamic instrumentation framework in the future, it could be improved in several ways. Currently, it is tightly integrated with Timely, which makes it difficult to add instrumentation to libraries which do not depend on the Timely library (called a “crate” in Rust [5]). In the future, it would be beneficial to create a dedicated logging library, which could then be used by both Timely as well as other components. An example of such a component would be Timely’s communication library, which already required a separate logging facility to allow communication events to be recorded, or the serialization library which is currently uninstrumented.

Moreover, the logging facility currently flushes the events from all the log buffers to disk at every step of the computation. This is done synchronously, which means the computation is delayed until the data is written to disk. Also, the instrumentation of Timely’s communication library flushes events to disk immediately. Ideally, any I/O operations should be taken *off the critical path*, such that they have no effect on the total runtime of the computation. This could be achieved in different ways. For example, an asynchronous I/O API could be used, such that the write calls would not block, and therefore not delay the computation as much. A different solution would be to store the events in a ring buffer and hand over the responsibility of writing the events to disk to a separate I/O thread. Logging calls would only block in case the buffer is full. To avoid lock contention, one could use separate buffers for all worker threads.

8.5.1 Integrating Clock Alignment

The logging facility itself should take periodic measurements of clock skew/-drift, preferably during a time in which the network is idle. It could then use the measurements to align the clocks on different machines, and hence produce event timestamps which do not need to be corrected by the performance analysis anymore. The increased accuracy provided by periodic measurements would make it possible to more precisely measure the duration of communication activities. See also Section 4.3.1 for more information about clock alignment.

8.6 Improving the Accuracy of the Wait-State Analysis

The wait-state analysis currently relies on a number of heuristics and assumptions. Although reasonable, these cannot always be correct. The following sections describe ways in which the accuracy of the wait-state analysis could be improved.

8.6.1 Replacing Timely's Static Scheduler

Instead of Timely's current static scheduling method, a dynamic scheduler could be implemented which would be aware of any outstanding work an operator has (e.g. by keeping track of input queue sizes and outstanding progress information). In this case, the scheduler itself would already know when a worker is in a waiting state, and would not need to spin the operators. Using appropriate instrumentation, the start- and end timestamps of waiting states could be captured directly, eliminating the need for a sophisticated trace-based wait-state analysis.

On the other hand, a proper dynamic scheduler adds a lot of complexity in comparison to Timely's current scheduling logic, and it is unclear if typical Timely computations themselves would even benefit from this approach.

8.6.2 Programming Model Modifications

In order to remove some of the uncertainty when determining causal relationships between events, Timely's programming model could be extended slightly. For example, an operator's logic could be split into separate callbacks for message- and progress notification processing. This would allow one to more accurately determine the causes of an operator's activity. Furthermore, special markers could be applied to operators which are allowed to read input from an external system, to allow one to more accurately determine whether an operator read external input or reacted to an earlier progress update.

However, any such extensions are also restrictions of Timely's very general and flexible programming model. Thus, one has to weigh that generality against the improved accuracy of the performance analysis.

8.6.3 More Precise Modeling of Progress Tracking

Finally, an improvement which does not require a modification of Timely's current behavior would be to more accurately model Timely's progress tracking. For this purpose, the contents of progress messages, i.e. each contained progress update, need to be logged. Then, the whole progress tracking protocol could be re-played by the analysis. Knowledge of the contents of progress updates as well as the behavior of the progress tracking logic would allow one to draw more accurate conclusions about the causal relationships between new progress messages, the progress updates which are pushed onto operators, and operator activity.

For example, a `PushProgress` event could be caused by a newly read progress message or by progress updates which were pushed onto the subgraph by an outer subgraph, or both. Re-playing the progress tracking protocol would reveal the exact cause(s).

If this approach is pursued in the future, it would be advisable to extract Timely's current implementation of the progress tracking protocol and move it to a dedicated library, which could then be loaded by both Timely as well as the performance analysis computation.

8.7 Improving the Visualization

The visualization tool described in Section 6.2 could be improved to display more information. For example, it could include the critical path profiles. A server-based component could also help to provide useful functionality such as computing aggregates over the trace/critical path data of multiple slices, management of stored traces and more. Furthermore, for large traces, the performance of the visualization could be improved by not loading the entire trace, but instead loading only the parts which are visible to the user (the view would need to be zoomed in to a level with acceptable performance by default). If the user scrolls to either side, the relevant parts of the trace could be loaded into memory on the fly (with some caching/pre-loading of adjacent sections).

Chapter 9

Conclusion

In this work, we have successfully applied critical path analysis to Timely Dataflow, a modern, distributed, data-parallel stream processing engine. We have shown that our refined performance model offers viable methods to find the causes of performance bottlenecks in computations. The trace-based approach only requires a small amount of instrumentation in the target system and the analysis can be performed efficiently. We have formalized our methodology such that it can be applied to other data-parallel systems.

Moreover, we have introduced an algorithm to identify waiting phases in a worker thread's execution based solely on the execution traces produced by the instrumentation. While this algorithm is specific to Timely Dataflow's runtime behavior, we believe that it is possible to adapt it to other, comparable systems.

We have further shown that it is possible to perform the entire performance analysis, including the identification of waiting phases and the critical path algorithm, as a data-parallel computation itself by implementing it using Timely Dataflow. Our prototype system is capable of performing critical path analysis efficiently and in a scalable manner. We have demonstrated that the entire analysis can even be performed in real-time or close to real-time in many configurations.

Thus, we offer both the methods as well as a system to identify application- or system components that limit the performance and scalability of data-parallel computations in real-time.

Bibliography

[1] Apache Flink: Scalable batch and stream data processing. `https://flink.apache.org/`. Retrieved: August 2016.

[2] Apache Storm. `http://storm.apache.org/`. Retrieved: August 2016.

[3] Data-Driven Documents. `https://d3js.org/`. Retrieved: August 2016.

[4] Differential Dataflow. `https://github.com/frankmcsherry/differential-dataflow`. Retrieved: August 2016.

[5] The Rust programming language. `https://www.rust-lang.org/`. Retrieved: August 2016.

[6] Timely Dataflow. `https://github.com/frankmcsherry/timely-dataflow`. Retrieved: August 2016.

[7] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, July 2008.

[8] Martín Abadi, Frank McSherry, and Gordon D Plotkin. Foundations of differential dataflow. In *International Conference on Foundations of Software Science and Computation Structures*, pages 71–83, 2015.

[9] Cedell Alexander, Donna Reese, and James C. Harden. Near-critical path analysis of program activity graphs. In *International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, 1994.

[10] Cedell A. Alexander, Donna S. Reese, James C. Harden, and Ron B. Brightwell. Near-critical path analysis: A tool for parallel program optimization. In *Southern Symposium on Computing*, 1998.

[11] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 127–138, 2000.

[12] David Böhme. *Characterizing Load and Communication Imbalance in Parallel Applications*. PhD thesis, Aachen University, Germany, June 2013.

[13] David Böhme, Bronis R. de Supinski, Markus Geimer, Martin Schulz, and Felix Wolf. Scalable critical-path based performance analysis. In *IEEE International Parallel and Distributed Processing Symposium*, 2012.

[14] Magnus Broberg, Lars Lundberg, and Håkan Grahn. Performance optimization using extended critical path analysis in multithreaded programs on multiprocessors. *Journal of Parallel and Distributed Computing*, 61(1):115–136, 2001.

[15] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, 2004.

[16] Jian Chen and Russell M. Clapp. Critical-path candidates: scalable performance modeling for MPI workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[17] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end performance analysis of large-scale internet services. In *OSDI*, 2014.

[18] Mathieu Desnoyers and Michel R Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[19] Isaac Dooley and Laxmikant V. Kalé. Detecting and using critical paths at runtime in message driven parallel programs. In *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[20] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems*, 30(4):13:1–13:35, 2012.

[21] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11):1268–1279, July 2012.

[22] Brian Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 74–85, 2001.

[23] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, 1973.

[24] Jeffrey K. Hollingsworth. An online computation of critical path profiling. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1996.

[25] Jeffrey K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, 1998.

[26] James E. Kelley, Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Eastern Joint IRE-AIEE-ACM Computer Conference*, 1959.

[27] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *Symposium on Operating Systems Principles*, 2015.

[28] Frank McSherry. Tracking progress in timely dataflow. `https://github.com/frankmcsherry/blog/blob/master/posts/2015-12-19.md`. Retrieved: August 2016.

[29] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6th Conference on Innovative Data Systems Research*, CIDR, January 2013.

[30] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, Apr 1990.

[31] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518. ACM, 2010.

[32] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, 2013.

[33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307. USENIX Association, 2015.

[34] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Online computation of critical paths for multithreaded languages. In *IPDPS Workshops on Parallel and Distributed Processing*, 2000.

[35] Ali G. Saidi, Nathan L. Binkert, Steven K. Reinhardt, and Trevor N. Mudge. Full-system critical path analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2008.

[36] M. Schulz. Extracting critical path graphs from MPI applications. *IEEE International Conference on Cluster Computing*, 2005.

[37] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[38] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth C. Goldstein. Global critical path: A tool for system-level timing analysis. In *DAC*, 2007.

[39] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *IEEE International Conference on Distributed Computing Systems*, 1988.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Real-Time Performance Analysis of a Modern Data-Parallel Stream Processing Engine

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Sager	Ralf

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, September 8, 2016	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.