Rescaling of Timely Dataflow

Lorenzo Selvatici

Aug 2019

1 Introduction

This document summarizes the work done to support dynamic rescaling of timely dataflow[1].

In a nutshell, timely runs distributed dataflow computation in a cluster. The shape of the cluster is fixed when first initializing the computation: how many timely processes (likely spread across several machines) and how many worker threads per timely process. With this project, we allow the addition of new worker processes to the cluster.

In long running jobs with unpredictable workloads, it is likely that the initial configuration will not be the ideal one. As such, there is the need to scale-out by adding more workers (and scale-in, by removing worker processes).

2 Timely Model

* each worker has a copy of the entire dataflow * async * progress tracking

3 Rescaling the computation

We now go into details of how the rescaling is actually implemented. Firstly, a high-level overview of the key points that need to be dealt with is given. Secondly, we present the design options for the initialization of the progress tracking state for the new worker. Lastly, we describe the (rather smooth) integration with Megaphone.

3.1 Communication Infrastructure

Rescaling the computation is possible only when running in cluster mode: multiple timely processes, each wrapping several worker threads, are established connections to each other and exchange both data and progress updates.

[1] https://github.com/LorenzSelv/timely-dataflow/tree/rescaling-p2p

In this setting, "adding more workers" means adding another timely process with the same number of worker threads of every other timely process in the cluster.

Communication among workers happens in two instances:

- `exchange` operator
- progress updates broadcast

In both cases, communication is enabled by *allocating a channel* which contains an endpoint to every other worker for both sending and receiving (`Pusher` and `Puller` traits, respectively).

Channel allocations happen while building the dataflow within the `worker::dataflow` function call.

If a new worker process joins the cluster, we need to *back-fill* these previously allocated channels. We do this by storing a closure for each past allocation, so that when the new worker initiates the connection we can invoke these closures which add the supplied pushers to the list of pushers forming the channel.

Each channel is associated with a specific data type. Thus, we cannot simply store a map of `(channel id => channel handle`, as collections can be generic but also need to be homogeneous. One might work around this by using trait objects, but then the channel would be associated with the trait object itself rather than the concrete type.

Moreover, fast-forwarding a bit, having a closure turned out to be very handy when implementing the bootstrap protocol to initialize the new worker progress tracker: when adding the new pusher to the list, we also push a *bootstrap message* that informs the new worker about the next progress-update sequence number it will receive from that direct connection.

When running in cluster mode, each worker process spawns an extra-thread that will accept incoming connections from workers joining the cluster. Upon accepting the connection, the thread will spawn a pair of `send` and `recv` network threads, perform some bookkeeping and finally inform the worker threads about the new timely process that is trying to join the cluster. This is achieved by sending a `RescaleMessage` on a shared `mpsc` channel.

Worker threads need to explicitly check for rescale messages via the `worker::rescale` function call. This is done in the very beginning of the `worker::step` function.

With the exception of the *bootstrap* protocol that we will talk about in a later section, this is all from the perspective of worker processes already in the cluster.

3.2 New Worker Initialization

Initialization of the new worker boils down to two things:

- building the dataflow (possibly more than one)
- initialize the progress tracker

We will now talk about them in detail.

3.2.1 Building the dataflow

Since the binary for the new process is identical to the binary of the other processes, the construction of the dataflow comes for free. The single and very important difference is how we handle *capabilities*. In particular, generic operators such as `source` and `unary_frontier` supply a capability to the user-provided constructor, which can use it request notifications or store it to retain the ability of producing output.

Clearly, as the new worker can join at any time in the computation, we must not supply capabilities for timestamps (epochs) that have been closed already as that would allow the worker to produce output "in the past".

Ideally, one would like to have a capability consistent with the frontier of that operator at the time of joining the cluster.

Unfortunately, there are some complications in trying to do this:

- As already mentioned, capabilities are passed to the constructor of the generic operators while building the dataflow. However, at least with the current implementation of the bootstrap protocol, we need to build the dataflow *before* actually performing the protocol. But then we do not know the operator frontier and thus also the right capability to supply.

- If we do supply a capability for a certain timestamp 't', we need to ensure that no other worker will ever consider that timestamp "closed" before the new worker discard the capability for that timestamp (e.g. by dropping or downgrading it).

For the first item, the issue could be worked-around by sending a map to the new worker which, for each operator, specifies the frontier of that operator.

Then, while building the dataflow, the new worker would look up in the map the frontiers and supply appropriate capabilities.

An alternative solution, and the one currently implemented, is to *not* supply capabilities at all. This approach does bring some limitations for the new worker:

- Operators can produce output only in response to some input (which comes with the associated capability for that timestamp). As such, `source` operators are useless for new worker: they do not hold any capability and cannot produce any output.

- Operators cannot request notifications for future times unless they use capabilities that came with input data.

There is an asymmetry between workers which hold capabilities (initial workers present in the cluster) and are capable of injecting new data in the dataflow (via the `source` operators) and workers that do not hold capabilities as they joined the cluster at a later time. So, while we can add worker at runtime, there are some special workers. Looking from a fault-tolerant perspective, where we would like to allow arbitrary worker to crash without compromising the execution, this could be a showstopper.

For the second item, the bootstrap worker (see later section about the bootstrapping protocol) should emit a `(t, +1)`[2] pointstamp (progress update), to ensure that the timestamp `t` will not be closed until the corresponding `(t, -1)` pointstamp is emitted. Such `(t, -1)` will be emitted by the new worker upon downgrading the capability.

3.2.2 Initializing the Progress Tracker

The progress tracking protocol is arguably the most important component at the core of timely computational model. As such we need to ensure that new workers have an up-to-date and correct view of the progress state.

When building the dataflow, each operator is supplied by default with capabilities for timestamp `0` (or the default analogous for different timestamp types). Some operators do not require capabilities and simply discard them. Other operators use them to request notification, produce output, etc. These operations on capabilities are associated with the emission of progress updates, made of a pair `(pointstamp, delta)`. These progress updates are broadcasted by every worker to all other workers via the established TCP connections between pair of processes.

There are two alternatives to initialize the progress state of the new worker:

- Reconstruct a consistent view by all the frontiers of the operator, which have changed over time as a consequence of progress updates
- Record and accumulate every progress update that has been sent so that we can apply them again

We implemented the second option, as it is easier to reason about and hard to get wrong. Also, as progress updates tend to cancel each other out (`+1` and `-1` pairs), we *expect* such accumulation to remain fairly small, no matter how long the computation has been run for. We should add proper instrumentation to the code to verify this claim.

In the next two sections we present the two alternatives for implementing the above idea: `pubsub`-based approach and `peer-to-peer`-based approach, both of them have been implemented to some extent, but `peer-to-peer` turned out to be a better option.

3.3 Pub-Sub Approach

One possible design would be to swap the all-to-all communication pattern for progress updates with a single `pubsub` system. Such system would have established connection to all workers. Each worker would send its progress updates that will be broadcasted on its behalf. Each worker then reads progress updates from every other worker.

Progress updates from each worker could be appended to a queue and when every other worker currently in the cluster has read the progress updates up to a certain

[2] it should actually be `num_worker_threads` instead of `+1`, as every new worker thread should get a capability

point, updates before that point can be safely compacted (most term would cancel out).

When a new worker joins the cluster, it would "subscribe" to every other worker queue, initialize its progress tracking state using the compacted updates and finally start reading new updates from the append-only queue of updates.

A possible implementation of such queue mechanism has been implemented[3].

3.3.1 Discussion

While this sounds like a clean and simple idea, there are a few complications:

- To perform updates compaction, the `pubsub` system needs to be aware of timestamp data types in the dataflow. Since these timestamps depend on the dataflow that has been constructed, the `pubsub` system needs to build the same dataflow, unless we implement some weird export mechanism. Moreover, the `pubsub` system needs to use timely data types, which would require a circular dependency between the two otherwise-unrelated crates.

- More on a philosophical note, adding this external system, which also represents a single point of failure, goes a bit against the peer-to-peer spirit of timely.

From performances point of view, there would be definitely some overhead associated with the `pubsub` system itself. Among other things, now there are two hop that a message needs to go through before reaching its destination. On the other hand, one could easily optimize the amount of progress messages sent around: you could have a single subscriber per process, so that the same progress messages are not duplicated because of multiple internal worker threads. This might also solve the communication volume bottleneck that some experiments show to prevent timely from scaling-out after a certain limit[4].

3.4 Peer-to-Peer Approach

An alternative design would be to have the new worker initializes its progress tracking state by selecting some *bootstrap server* and perform some *bootstrap protocol*.

In particular, the bootstrap server is simply another worker that is selected to help the new worker initializes its progress tracking state.

The rescale message received by the worker thread contains a flag signaling if the worker thread was selected as the bootstrap server, and in that case it would initiate the bootstrap protocol.

[3] https://github.com/LorenzSelv/pubsub

[4] mostly referring to the results in the Noria paper, but we should perform some proper experiments to investigate this further

3.4.1 Bootstrap Protocol

Before describing the protocol, we presents some changes needed to make it possible. To begin with, the server side of the protocol is performed in the `Worker::rescale`[5] and `TcpAllocator::rescale`[6] functions; the client side of the protocol is performed in the `bootstrap_worker_client`[7] function.

Somewhat similarly to the previous approach, each `Progcaster`[8] keeps an accumulation of all progress updates as a `ChangeBatch`. We define the progress state as this accumulation plus the information about the last sequence number included in the state for each worker.

The bootstrap server, while performing the protocol does not perform other work. Since it needs to use non-thread-safe data structures, it was not possible to spawn a separate thread to perform the protocol.

The bootstrap protocol (also depicted in figure 1) consists of the following steps:

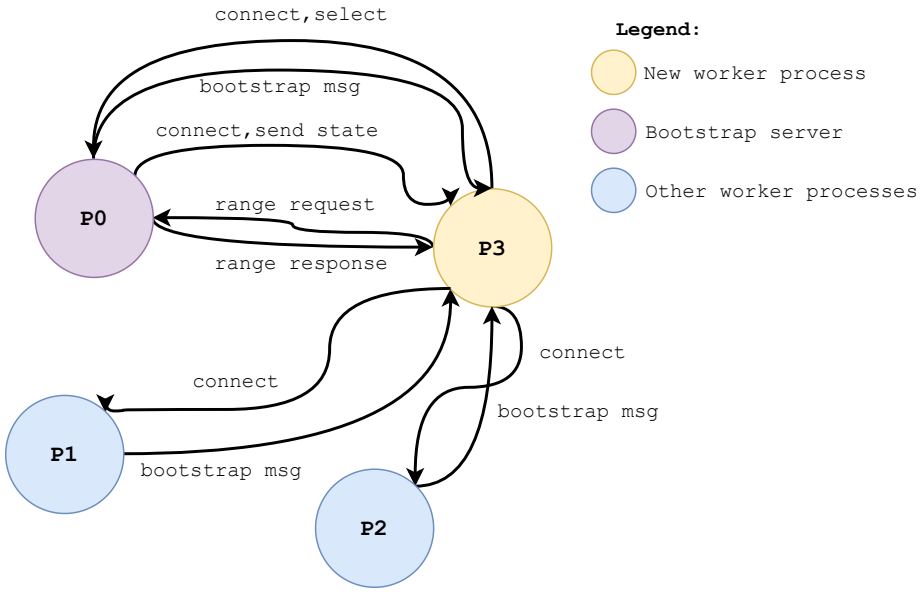

Figure 1: Bootstrap protocol, timely processes are shown as circles

We will refer to the new worker joining the cluster as "bootstrap client".

1. `Bootstrap client waits for an incoming connection at some arbitrary bootstrap address.`

2. `Bootstrap server initiates the connection to that same address.`

[5] `timely/src/worker.rs`
[6] `communication/src/allocator/zero_copy/allocator.rs`
[7] `communication/src/rescaling/bootstrap.rs`
[8] the entity responsible for broadcasting and receiving progress updates for a specific scope

3. `Bootstrap server sends the progress tracking state to the new worker and start recording following progress updates.` The progress tracking state has been defined above. Recording messages simply means appending them to a list (different for each worker) so that we can access them at a later time. The lists are cleared at the end of the protocol.

4. `Bootstrap server listens for incoming progress-updates-range requests.`

5. `Bootstrap client inspects direct TCP connections with other workers, where it will finds the bootstrap message, containing the next progress-update sequence number it will read from that channel.` The bootstrap message is the first message that is sent over the newly-established TCP connection.

6. `Bootstrap client computes the missing progress updates ranges by comparing the sequence numbers, in the state and the ones in the bootstrap messages.` Since the new worker has the guarantee that it will see all progress updates with sequence number greater or equal to the one in the bootstrap message, filling the missing gaps between the received progress state and the such sequence numbers guarantees that it will have seen all progress updates.

7. `Bootstrap client sends progress-updates-range requests to the bootstrap server.`

8. `Bootstrap server sends progress-updates-range responses to the bootstrap client.` A progress-updates-range request has the format `(scope_id, worker_index, start/end sequence number)` which uniquely identifies some updates range. If the bootstrap server cannot fulfill the request as it has not seen all requested messages yet, it will pull more progress updates from the channels with the other workers. Eventually, all required progress will be received also by the bootstrap server which will be then able to fulfill the request.

9. `Bootstrap client terminates the bootstrapping protocol by closing the connection.`

3.5 Megaphone Integration

* not much, just number of peers changing over time

4 API changes affecting user code

* capability is now an option * need to call worker::bootstrap

5 Evaluation

* steady-state overhead compared to timely master (changes slowed down something?) - look up diff for any overhead change - progress state update

* plot latency vs time (epochs), two lines with expected behaviour: 1) timely/master can't keep-up with input rate of sentences (linearly increasing) 2) timely/rescaling can't keep-up as well, and it's slightly higher due to some overhead? rescale operation to spawn one/two more workers =¿ spike in latency but the lower

* size of the progress state over time, as a function of: - number of workers - workload type

* breakdown of timing in the protocol

6 Limitations and Future Work

* no removal of workers * no capabilities at all for new workers * formal verification of the protocol