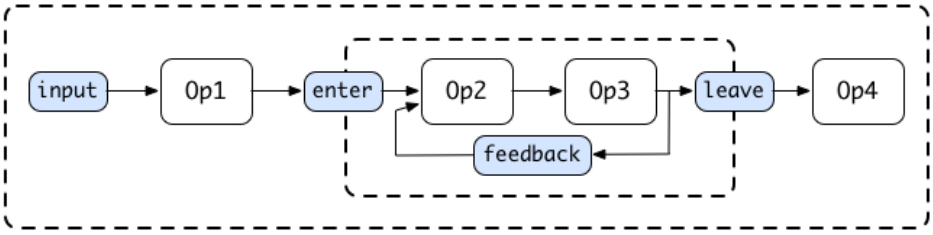
Progress Tracking

Created	@September 11, 2021 5:15 PM
Tags	

Why Progress Tracking?

Timely dataflow supports parallel execution of messages of different rounds of loops and different epochs.



We assign timestamp to messages.

We would like the operator to know when it has received all messages prior to a timestamp `t`. This is called notifications. To enable notifications, we need progress tracking.

Concepts

Capabilities

An output port needs to hold some capabilities in order to send messages. When an operator receives an input with timestamp t it also receives a capability to send messages with that timestamp. The operator can send messages with timestamp `t` only if it holds a capability for a timestamp `t' <= t`. If the capability with the least timestamp hold by the output port is `t'`, it indicates this port cannot send messages with timestamp less than `t'`.

Messages

Each message on the fly bears a timestamp. This timestamp is also modified by the operators it passes.

SharedProgress

`SharedProgress`: to record the progress information and it is shared between the child (e.g., operator) and the parent (e.g., the parent scope that the operator resides in)

- **frontiers**: Each subgraph / operator gets a frontier status update of each of its input port `x` (the input to the operator, or the inputs from the outside world to the subgraph (that is the output ports of Op 0 in the subgraph) from `SharedProgress`. The frontier update is recorded in `ChangeBatch<T>`. It maintains the timestamps `t` that some precursory locations (operator + ports)'s outstanding messages and capabilities could-result-in `t`, i.e., when a precursory location sends a message with `t'` that goes through the graph, the timestamp will get increased to `t` when it arrives at `x`. The parent scope reports the child's frontier update in `SharedProgress`, the counts indicate **the number of locations (NOT the number of messages / pointstamps)** added / decreased that could cause `t` at `x`.
- **consumeds**: In `Shared Progress`, there is a `ChangeBatch<T>` for each input port to record the number of messages consumed.

- **internals**: One `ChangeBatch<T>` for each output port to record the changes to the internal capabilities this port holds. An output port could drop capability `t` if it will not send any more messages with timestamp `t' <= t`.
- **produceds**: One `ChangeBatch<T>` for each output port to record the number of messages the output port produced.

Subgraph-Level Reachability Tracker

Output Summarizer

To answer the could-result-in questions, we need to track for a location `x'` and a timestamp `t'`, what is the least timestamp `t` (at `x`) that `t'` would increase to when it walks along the possible paths from `x' -> x`? To do this, we first use the function `summarize_outputs` to get a PathSummary with the least timestamp change from every location (port) in the subgraph (scope) to every subgraph output (i.e., the input ports of operator 0).

This function starts from subgraph outputs, then backtrace the subgraph to find the least PathSummary for a location `x` to an subgraph output. If a smaller PathSummary for `x` if found, then this `x` is added to the queue to further backtrace from it.

Per-Port Information

- **pointstamps**: For each input/ouput port `x`, we maintain its occurrence counts of its pointstamps (messages / capabilities AT location `x`). This counts the events happening at `x`, it is not propagated. We counts the number of events for each timestamp `t` at location `x`. We also maintain the current minimal timestamp `t` at `x`.
- **implications**: The computed **frontier** after propagating the graph. It only counts **the number of locations** `x'` with a timestamp `t'` that when goes through `x'->x'`, leads to `t`. We also maintain the minimal of such timestamp `t`. It indicates that this operator will no longer see any messages with timestamp less than `t`, and it cannot send any message with timestamp less than `t`. We manually update the pointstamps, and the implications are computed.

Since we only require timestamps and path summaries to be partial ordered, for `x'->x`, there might be multiple "minimum" path summaries where they are incomparable to each other. We use `Antichain` to store path summaries, which maintains a set of "smallest elements", i.e., a set of incomparable elements no greater than any other element (not in the set).

We use `MutableAntichain` to store `pointstamps` and `implications`. `MutableAntichain` supports counts for the elements, it will return a set of "smallest elements with positive counts". It maintains the entire count update history, and maintains a frontier, which is such set of "smallest elements with positive counts".

Propagate along the Subgraph

1. `propagate_all` will first merge and drain the pointstamps changes we made to each location `x` (node + port). The pointstamps changes will be written to the port information storage from the `ChangeBatch` buffer.
2. For location `x`, we call the `update_iter` of the port information storage's `pointstamps`. `update_iter` is a method of `MutableAntichain`, it returns the changes to the frontier (changed timestamp `t` & change is +1 (added to the frontier) / -1 (removed)). Then the changed timestamp `t` is propagated to the scope outputs. We then update the changes into output capabilities (a `ChangeBatch` for each scope output). The output capabilities indicate the number of locations that has timestamps `t'` could lead to events with timestamps `t` at `x`, and `t` is the smallest timestamp that the current outstanding messages and capabilities could-result-in. It indicates the scope outputs will only generate messages with timestamps greater or equal to `t`.
3. Starting from the locations whose distinct pointstamp changes (that is, the `MutableChain`'s frontier of `pointstamps` change), we propagate the frontier change along the subgraph. With a location `x` and a timestamp `x` which is added or removed from the `pointstamps`'s `MutableChain`'s frontier, we add (`x, t, diff (+1/-1)`) to a priority queue (sorting according to timestamp). We extract the tuple with minimum timestamp at each time. We update `x`'s `implications` with change `diff` to timestamp `t`, i.e., the pointstamp `(x, t)` itself is added to `x`'s frontier. We get the changes to `implications`'s `MutableChain`'s frontier (diff +1/-1). We push the changes to the port's frontier in `self.pushed_changes`.

a. If `x` is an input port, propagate along the operator (path summaries) to the output ports `x'`. For each `x'`, we get a timestamp `t'` that `t` results in. We add `(x', t', diff)` to the priority queue to further propagate the frontier changes along the subgraph. Note that `diff` is +1 or -1, indicating whether this timestamp `t` is added or removed from `x`'s frontier.

```
for (time, diff) in changes {
                            let nodes = &self.nodes[location.node][port_index];
                            for (output_port, summaries) in nodes.iter().enumerate()  {
                                let source = Location { node: location.node, port: Port::Source(output_port) };
                                for summary in summaries.elements().iter() {
                                    if let Some(new_time) = summary.results_in(&time) {
                                        self.worklist.push(Reverse((new_time, source, diff)));
                                    }
                                }
                            }
                            self.pushed_changes.update((location, time), diff);
                        }
```

b. If `x` is an output port, just further propagate the frontier changes along the input ports `x` is connected to, without changing to the timestamp.

```
for (time, diff) in changes {
                            for new_target in self.edges[location.node][port_index].iter() {
                                self.worklist.push(Reverse((
                                    time.clone(),
                                    Location::from(*new_target),
                                    diff,
                                )));
                            }
                            self.pushed_changes.update((location, time), diff);
                        }
```

4. `pushed_changes` stores the frontier changes to every location in the subgraph.

`is_global()`

```
pub fn is_global(&self, time: &T) -> bool {
        // if returns true, then only a single outstanding event (pointstamp) that could-result-in (t, *this)
        let dominated = self.implications.frontier().iter().any(|t| t.less_than(time));
        // "precursor count"
        let redundant = self.implications.count_for(time) > 1;
        !dominated && !redundant
    }
```

When we call `x.is_global(t)` on a location `x` with timestamp `t`, a `true` return value indicates that the (singular) pointstamp `(x,t)` holds at location `x` is in the frontier (currently blocking the progress). Or we can say that if we remove a pointstamp `(x,t)` (decrease occurrence count by 1), then the frontier of `x` will change.

Subgraph

A Subgraph serves as a computation scope, where it contains many children operators. It has a special placeholder operator indexed by 0 (Op 0), to serve as the scope's bridge to the outside world. The parent scope inputs data to the subgraph (sub-scope) through Op 0, and the subgraph outputs through Op 0 to the parent scope.

In the inside of the subgraph, we have the parent scope's inputs coming from Op 0's output ports, and we generate outputs to the parent scope through Op 0's input ports. We would establish a `PerOperatorState` to hold the states for each child. `PerOperatorState` holds the child operator, its internal summary and `SharedProgress`.

- `SharedProgress` is created by the child operator and shared with the parent scope.

- The parent scope puts frontier update to `ShareProgress.frontiers`.
- The child scope updates the rest (`consumeds, internals, produceds`).

Frontier Propagation

For each operator, `SharedProgress`:

- **frontiers**: related to the input ports, it contains the smallest outstanding timestamp and the number of possible locations that could-result-in this timestamp at a future point.
- **internals**: related to the output ports, it indicates the output port's capabilities. An output port needs to hold capability `t` to in order to send messages with `t'>=t`. The capabilities (internals) can be inferred from frontiers via propagation, and also it can be obtained when the operator consumes messages).
- **consumeds**: related to the input ports, the number of messages consumed. No propagation.
- **produceds**: the number of messages the output ports produced. No propagation. It will increase the pointstamp occurrence counts of all the input ports this output port connects to.

Normally, when we call the reachability tracker's `update` methods to update the pointstamps for a location `x`. When we call `update`, we are talking about an actual event with timestamp `t`, it is not some propagated information. When `x` is an input port, we update pointstamps when messages are sent to `x` (increase pointstamps) or consumed by `x` (decrease pointstamps). When `x` is an output port, we update pointstamps when its capabilities change.

Schedule

When the subgraph (itself is just an operator looking from outside) is scheduled:

1. Take the frontier it obtained from the parent scope (with timestamp type `TOuter`, stored in the subgraph's `SharedProgress`) into the subgraph scope by calling `accept_frontier()`. The frontier change to the subgraph will be recored as the pointstamps change in Op 0's output ports (which take inputs from the parent). **Note that this is the only case we update pointstamps based on propagated information (frontier change) instead of actual messages events / output port capabilities changes that happen in place.** This is just used to ease the pain of computing the frontier propagation along the subgraph.

   ```
   /// Move frontier changes from parent into progress statements.
       fn accept_frontier(&mut self) {
           for (port, changes) in self.shared_progress.borrow_mut().frontiers.iter_mut().enumerate() {
               let source = Source::new(0, port);
               for (time, value) in changes.drain() {
                   self.pointstamp_tracker.update_source(
                       source,
                       TInner::to_inner(time),
                       value
                   );
               }
           }
       }
   ```

2. We take the number of incoming inputs recorded in `self.input_messages`, we deliver the messages to the input ports connected to the Op 0's output ports (sources) by (first locally) add the pointstamps counts of the target input ports. When these input ports consume the messages, the pointstamps would decrease.

   ```
   fn harvest_inputs(&mut self) {
           for input in 0 .. self.inputs {
               let source = Location::new_source(0, input);
               let mut borrowed = self.input_messages[input].borrow_mut();
               for (time, delta) in borrowed.drain() {
                   for target in &self.children[0].edges[input] {
   ```

```
                    self.local_pointstamp.update((Location::from(*target), time.clone()), delta);
                }
                // this pointstamp change at Op 0 will not actually be updated into the tracker.
                // it is just used to fill in the subgraph's SharedProgress
                // to tell the parent scope the number of messages the subgraph consumed.
                self.local_pointstamp.update((source, time), -delta);
            }
        }
    }
```

3. Receive pointstamps changes from all peers (including itself) via `self.progcaster.recv(&mut self.final_pointstamp);`. `self.final_pointstamp` is just updated with newly received changes.
4. Call the tracker's `update` method on the received pointstamps changes to actually update the pointstamps change.
    a. Not for Op 0, for Op 0, the changes in `final_pointstamp` just indicates the number of inputs received from the parent scope / outputs send to the parent scope. We use the information to update `SharedProgress`.

```
for ((location, timestamp), delta) in self.final_pointstamp.drain() {

            // Child 0 corresponds to the parent scope and has special handling.
            if location.node == 0 {
                match location.port {
                    // [XXX] Report child 0's capabilities as consumed messages.
                    //       Note the re-negation of delta, to make counts positive.
                    Port::Source(scope_input) => {
                        self.shared_progress
                            .borrow_mut()
                            .consumeds[scope_input]
                            .update(timestamp.to_outer(), -delta);
                    },
                    // [YYY] Report child 0's input messages as produced messages.
                    //       Do not otherwise record, as we will not see subtractions,
                    //       and we do not want to present their implications upward.
                    Port::Target(scope_output) => {
                        self.shared_progress
                            .borrow_mut()
                            .produceds[scope_output]
                            .update(timestamp.to_outer(), delta);
                    },
                }
            }
            else {
                self.pointstamp_tracker.update(location, timestamp, delta);
            }
        }
```

5. Use the tracker to propagate the changes to obtain the frontiers.
6. The computed frontier changes to all the children (intermediate operators) is updated into the children's `SharedProgress` (this is shared between the children and the subgraph, bearing the timestamp of type `TInner`). The children now understands its frontier. *Note that frontier is something similar to precursors in the original Naiad paper. And note that we call `drain()` on the updates.*
7. The updated frontiers of the scope outputs (input ports of Op 0) serves as the child's output capabilities and `internals` in the subgraph's `SharedProgres` to its parent scope. It indicates the progress. With a timestamp `t` it means the subgraph will not output messages with timestamp `t' < t`.
8. Enqueue and schedule child operators.
9. Send local progress updates.
10. If final pointstamps is not empty, we must re-schedule current subgraph at a later point.

Extract Progress

We can extract the progress updates from child's `SharedProgress` (with timestamp type `TInner`) to the subgraph's pointstamps counts.

```
/// Extracts shared progress information and converts to pointstamp changes.
    fn extract_progress(&mut self, pointstamps: &mut ChangeBatch<(Location, T)>, temp_active: &mut BinaryHeap<Reverse<usize>>) {

        let shared_progress = &mut *self.shared_progress.borrow_mut();

        // consumes the shared progress
        // Migrate consumeds, internals, produceds into progress statements.
        for (input, consumed) in shared_progress.consumeds.iter_mut().enumerate() {
            let target = Location::new_target(self.index, input);
            // Note that we call drain() to drain the updates
            for (time, delta) in consumed.drain() {
                // consumes messages
                // decrease the occurrence counts for the input port that receives the messages
                pointstamps.update((target, time), -delta);
            }
        }
        for (output, internal) in shared_progress.internals.iter_mut().enumerate() {
            let source = Location::new_source(self.index, output);
            for (time, delta) in internal.drain() {
                // changes to capabilities, update the pointstamps for the output port
                pointstamps.update((source, time.clone()), delta);
            }
        }
        for (output, produced) in shared_progress.produceds.iter_mut().enumerate() {
            for (time, delta) in produced.drain() {
                for target in &self.edges[output] {
                    // send messages
                    // increase the occurrence counts for all the input ports this output port connects to
                    pointstamps.update((Location::from(*target), time.clone()), delta);
                    temp_active.push(Reverse(target.node));
                }
            }
        }
    }
```