dataflow/channels

Created	@September 12, 2021 2:14 AM
Tags	

Messages

Messages in timely dataflow's channels are defined as:

```
pub struct Message<T, D> {
    /// The timestamp associated with the message.
    pub time: T,
    /// The data in the message.
    pub data: Vec<D>,
    /// The source worker.
    pub from: usize,
    /// A sequence number for this worker-to-worker stream.
    pub seq: usize,
}
```

It contains both a timestamp and the actual data (a vector of records). It also contains the source worker ID, and the sequence number (seqno).

Messages implement an associated function `push_at()`, which takes a buffer and a timestamp, and a pusher. It constructs the message to push to the pusher. In the progress, the buffer is consumed and is replaced by an empty vector.

Tee

```
pub struct Tee<T: 'static, D: 'static> {
    buffer: Vec<D>,
    // we use dyn trait so that we can push to pushers of different types
    // e.g., thread-local pushers & inter-process pushers
    shared: Rc<RefCell<Vec<Box<dyn Push<Bundle<T, D>>>>>>,
}
```

`Tee` is a struct that we can designate multiple pushers of different types (e.g., thread-local pushers and inter-process pushers). When we push a message to `Tee`, it will further push the message to all the pushers it shares.

TeeHelper

We also have a `TeeHelper` to allow us to easily install new pushers to a `Tee`:

```
pub struct TeeHelper<T, D> {
    shared: Rc<RefCell<Vec<Box<dyn Push<Bundle<T, D>>>>>>
}

impl<T, D> TeeHelper<T, D> {
    /// Adds a new `Push` implementor to the list of recipients shared with a `Stream`.
    pub fn add_pusher<P: Push<Bundle<T, D>>+'static>(&self, pusher: P) {
        self.shared.borrow_mut().push(Box::new(pusher));
    }
}
```

Counter Pusher and Puller

The counter pusher and puller wrap a pusher or a puller, and reports the number of messages pushed / pulled to a shared `ChangeBatch`:

```
pub struct Counter<T: Ord, D, P: Push<Bundle<T, D>>> {
    pushee: P,
```

```
    produced: Rc<RefCell<ChangeBatch<T>>>,
    phantom: ::std::marker::PhantomData<D>,
}

pub struct Counter<T: Ord+Clone+'static, D, P: Pull<Bundle<T, D>>> {
    pullable: P,
    // thread-local shared count map
    consumed: Rc<RefCell<ChangeBatch<T>>>,
    phantom: ::std::marker::PhantomData<D>,
}
```

Parallelization Contracts

We have two types of parallelization contracts to let us define how data records are moved between the workers on a dataflow edge.

Pipeline

It just pushes the message from the worker to the worker itself (thread-local).

Note `connect` returns a pusher and puller. It wraps the pushers and puller allocated by the worker (allocator).

```
impl<T: 'static, D: 'static> ParallelizationContract<T, D> for Pipeline {
    type Pusher = LogPusher<T, D, ThreadPusher<Bundle<T, D>>>;
    type Puller = LogPuller<T, D, ThreadPuller<Bundle<T, D>>>;
    fn connect<A: AsWorker>(self, allocator: &mut A, identifier: usize, address: &[usize], logging: Option<Logger>) -> (Self::Pusher, Self:
        // allocate thread-local channel for the pipeline
        // the address specifies a path to an operator that should be
        // scheduled in response to the receipt of records on the channel.
        // identifier is the ID of the channel
        let (pusher, puller) = allocator.pipeline::<Message<T, D>>(identifier, address);
        // thread local channel, so the source and target are all allocator.index()
        (LogPusher::new(pusher, allocator.index(), allocator.index(), identifier, logging.clone()),
         LogPuller::new(puller, allocator.index(), identifier, logging.clone()))
    }
}
```

Exchange

In this contract, the current worker can send records to all the workers (peers) in the system, including itself. The worker a record is sent to depends on a provided hash function.

```
// Exchange uses a `Box<Pushable>` because it cannot know what type of pushable will return from the allocator.
impl<T: Eq+Data+Clone, D: Data+Clone, F: FnMut(&D)->u64+'static> ParallelizationContract<T, D> for Exchange<D, F> {
    // TODO: The closure in the type prevents us from naming it.
    //       Could specialize `ExchangePusher` to a time-free version.
    type Pusher = Box<dyn Push<Bundle<T, D>>>;
    type Puller = Box<dyn Pull<Bundle<T, D>>>;
    fn connect<A: AsWorker>(mut self, allocator: &mut A, identifier: usize, address: &[usize], logging: Option<Logger>) -> (Self::Pusher, S
        // senders is a vector of length #workers, it contains a pusher to each worker
        let (senders, receiver) = allocator.allocate::<Message<T, D>>(identifier, address);
        // wraps each sender
        // into_iter() consumes the vector (container)
        let senders = senders.into_iter().enumerate().map(|(i,x)| LogPusher::new(x, allocator.index(), i, identifier, logging.clone())).col
        // use ExchangePusher to wrap the senders with the hash function to dynamically distribute the data among the workers
        (Box::new(ExchangePusher::new(senders, move |_, d| (self.hash_func)(d))), Box::new(LogPuller::new(receiver, allocator.index(), iden
    }
}
```

Here, `Exchange` is a wrapper that wraps the pushers to all the workers. It then uses the hash function to determine which pusher to push to, when we call `push()` on `Exchange`.

Buffer

A buffered pusher wrapper.

```
/// Buffers data sent at the same time, for efficient communication.
///
/// The `Buffer` type should be used by calling `session` with a time, which checks whether
/// data must be flushed and creates a `Session` object which allows sending at the given time.
pub struct Buffer<T, D, P: Push<Bundle<T, D>>> {
    time: Option<T>,  // the currently open time, if it is open
    // it is an Option since the buffer might not be open.
    buffer: Vec<D>,   // a buffer for records, to send at self.time
    pusher: P,
}
```

We can allocates a `AutoflushSession` from it by providing it a capability. The capability is automatically dropped to inform the system about the capability change when the instance of `AutoflushSession` is dropped.

```
/// Allocates a new `AutoflushSession` which flushes itself on drop.
    pub fn autoflush_session(&mut self, cap: Capability<T>) -> AutoflushSession<T, D, P> where T: Timestamp {
        if let Some(true) = self.time.as_ref().map(|x| x != cap.time()) { self.flush(); }
        self.time = Some(cap.time().clone());
        // AutoflushSession holds the capability
        // so that when AutoflushSession is dropped
        // _capability is also dropped
        // _capability will inform the system it has relinquished its ability to send data at timestamp t
        // through updating `internal` when it is dropped
        AutoflushSession {
            buffer: self,
            _capability: cap,
        }
    }
```