Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Spring Term 2019 Dr. Vasiliki Kalavri

Data Stream Processing and Analytics

Assignment 2

In this exercise session, you will learn:
- How to setup Timely Dataflow on your laptop/desktop
- How to setup a development environment for Timely
- How to write and execute Timely Dataflow programs

You are going to need a UNIX-based setup to follow this assignement. If you are a Windows user, you are advised to use Windows subsystem for Linux (WSL), Cygwin, or a Linux virtual machine to run Timely in a UNIX environment.

1. Setup up Timely Dataflow and run an example

Timely Dataflow is a system for writing and executing distributed streaming computations. Its compuattion model is described in the Naiad paper and one of its distinctive features is that it supports dataflows with cycles (even nested ones!)

In what follows, you will learn the basic structure of a dataflow, how to provide inputs and then extend the graph with operators to observe the outputs. Following on this, you will feed data in logical epochs, distribute across multiple workers and use probes to monitor the progress of the computation. Before embarking on the tasks, we advise you to consult the documentation which introduces core concepts:

- Tutorial Book: http://timelydataflow.github.io/timely-dataflow/
- API Documentation: https://docs.rs/timely

Setup

To setup and run Timely, you need to have a working Rust installation. Go to https://www.rust-lang.org/tools/install and follow the instrustions to install Rust on your machine. You can check if everything went fine by opening a terminal and running `rustc --version`.

The output should look something like this:

```
rustc 1.32.0 (9fda7c223 2019-01-16)
```

Create a new folder for your project by running:

```
cargo new timely-playground --bin
```

You should now have a folder structure which looks as follows:

```
timely-playground/
├── Cargo.toml
├── src
│   └── main.rs
```

Open the `Cargo.toml` file and add Timely dataflow as a dependency to your project:

```
[package]
name = "timely-playground"
version = "0.1.0"

[dependencies]
timely = "0.8"
```

Run an example

We can now write our first -very simple- Timely program. Open the file `main.rs` and enter the following lines:

```
extern crate timely;
use timely::dataflow::operators::{Inspect, ToStream};

fn main() {
    // Demo: wraps an iterator into a simple streaming pipeline
    timely::example(|scope| {
        (0..10).to_stream(scope)
            .inspect(|x| println!("seen: {:?}", x));
    });
}
```

This program uses the *external* crate “timely” and all it does is create and print a stream of the integers 0 to 10. Run it by issuing the command:

```
cargo run --release
```

in the project home directory. You should see the following output:

```
seen: 0
```

```
seen: 1
seen: 2
seen: 3
seen: 4
seen: 5
seen: 6
seen: 7
seen: 8
seen: 9
```

Taking a step back, you will notice a few details within the code:

- We first create a lazy iterator with all values in the range [0, 10).
- We feed these inputs in by wrapping the iterator into a Timely `Stream` object. The argument being passed is the root scope which represents our dataflow graph.
- We inspect the data flowing through by using the fluent API to instantiate an operator. The argument being passed is an anonymous function where we can place logic that is is invoked on every tuple. For simplicity, we just print the output to the console in this example.

2. Multiple workers and data parallelism

Let's do something a bit more interesting. The following code generates and pushes data to the dataflow in several *rounds* and instructs the Timely workers to do some processing after each round:

```
extern crate timely;

use timely::dataflow::{InputHandle, ProbeHandle};
use timely::dataflow::operators::{Input, Inspect, Probe};

fn main() {
    // 1) Instantiate a computation pipeline by chaining operators
    timely::execute_from_args(std::env::args(), |worker| {

        let index = worker.index();
         // create opaque handles to feed input and monitor progress
        let mut input = InputHandle::new();
        let mut probe = ProbeHandle::new();

        worker.dataflow(|scope| {
            scope.input_from(&mut input)
           .inspect(move |(round, num)| println!("round: #{}\tnum: {}\
tworker: {}", round, num, index))
              .probe_with(&mut probe);
        });

        // 2) Push data into the dataflow and allow computation to run
        for round in 0..10 {
            for j in 0..round + 1 {
                 input.send((round, j));
```

```
                }
                // advance input and instruct the workers to do work
               input.advance_to(round + 1);
               while probe.less_than(input.time()) {
                    worker.step();
               }
          }
     }).unwrap();
}
```

Let us look at what is happening here.

First, `input.send()` moves data to a queue from which operators receive their inputs. The `InputHandle` will take care of moving the data to the appropriate workers and operators. Notice however that it is the same worker thread that generates the data and executes the dataflow code. That's exactly why after every round we call `worker.step()`.

Second, calling `input.advance_to()` informs the system that there will be no future input sent with an earlier timestamp than the specified one. The reasons this is necessary will hopefully become clear during the semester when we study the concepts of time and progress in dataflows. All you need to know at this point is that you can retrieve the current time of an input with `input.time()` and monitor the dataflow's progress with the `ProbeHandle`. Thus, all the `probe.less_than(input.time())` call does is check whether the computation has *caught up* with the input.

Tasks

#1: Extend the above program to work with more than one workers. To split the data and work among parallel workers, make the following changes:

- Have only worker 0 generate input data (hint: use the already defined `index`)
- Use Timely's `exchange` operator to partition the data across workers. Make sure to have all workers receive data during all rounds instead of diving the data per round. Check the Timely Dataflow documentation for an example.
- You can now execute the program using the `-w` option. For instance, the command `cargo run -release -- -w2` will execute the program with 2 workers.

3. You're finally ready to count words

No data processing tutorial is complete without a word count task. We will write a Timely Dataflow program that reads text from files, splits each line into words, and counts how many times each word occurs.

Reading text from files

You can use the following code to have parallel workers read text from files and push it to the dataflow line-by-line:

```
let path = format!("/path/to/input-{}.txt", index);
let file = File::open(path).expect("Input data not found in CWD");
let buffered = BufReader::new(file);
// send input line-by-line
let mut total_lines = 0;
for line in buffered.lines() {
    input.send(line.unwrap());
    total_lines = total_lines + 1;
}
// advance input and process
input.advance_to(total_lines + 1);
while probe.less_than(input.time()) {
    worker.step();
}
```

This code assumes that you have split your input text in as many files as available workers and you have named the files `input-N.txt`, where `N` is the worker number, so that worker 0 reads from `input-0.txt`, worker 1 reads from `input-1.txt`, etc. You can find plenty of text online for testing, as for instance, the lyrics of Yellow Submarine.

Splitting lines into words and counting them

The following Timely code uses a `flat_map` operator to split every text line into words and output pairs of (word, 1). It then feeds this data into an `aggregate` operator which counts the occurrences of each word. Notice how we did not have to use exchange to partition the words as the `aggregate` operator re-routes tuples and performs the grouping for us:

```
worker.dataflow(|scope| {
      scope.input_from(&mut input)
            .flat_map(|text: String|
                  text.split_whitespace()
                        .map(move |word| (word.to_owned(), 1))
                        .collect::<Vec<_>>()
            )
            .aggregate(
                  // fold: combines new data with existing state
                  |_key, val, agg| { *agg += val; },
                  // emit: produce output from state
                  |key, agg: i64| (key, agg),
                  // hash: route data according to a key
                  |key| hash_str(key)
            )
         .inspect(move |(word, count)| println!("worker: #{}\tword: {}\tcount: {}", index, word, count))
                  .probe_with(&mut probe);
});
```

Tasks

#1: Put the code together into a Timely Dataflow program and run wordcount for the lyrics of your favorite song.

#2: How is this wordcount different than the one we saw in Assignment 1? Can you change the program to read text, send inputs, and perform computation continuosuly (e.g. every 5 lines of text) instead of all at once?