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 – Sample Solution

2. Multiple workers and data parallelism

Tasks

#1: To effectively make use of data parallelism, we need to re-route tuples and achieve a uniform balance among workers. The `Exchange` operator is parameterized by an arbitrary hash function which receives a single element of the stream and returns the index of the worker it should be sent to. The required changes look as follows:

```
--- src/bin/part2.rs
+++ src/bin/part2.rs
@@ -1,7 +1,9 @@
 extern crate timely;

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

 fn main() {
     // initializes and runs a timely dataflow.
@@ -14,15 +16,18 @@
         worker.dataflow(|scope| {
             scope
                 .input_from(&mut input)
+                // re-partition across workers based on counter value
+                .exchange(|(_round, num)| *num)
                 .inspect(move |(round, num)| println!("round: #{}\
tnum: {}\tworker: {}", round, num, index))
                 .probe_with(&mut probe);
         });
```

```diff
          // introduce data and watch!
          for round in 0..10 {
-             for j in 0..round + 1 {
-                input.send((round, j));
-            }
+             // have only the first worker generate input data
+             if index == 0 {
+                 for j in 0..round + 1 {
+                     input.send((round, j));
+                 }
+             }
              input.advance_to(round + 1);
              while probe.less_than(input.time()) {
                  worker.step();
```

After compiling and running, you will now see output of the following form (e.g. with two workers):

```
  round: #0       num: 0  worker: 0
  round: #1       num: 0  worker: 0
  round: #1       num: 1  worker: 1
  round: #2       num: 0  worker: 0
  round: #2       num: 2  worker: 0
  round: #2       num: 1  worker: 1
                          ~~~~~~~~~
                                  ^
```

If you look back at original code from the handout, data was produced and consumed on the same worker meaning that you could expect to see arbitrary numbers across the logs of any worker. Take note of the highlighted column and how partitioning allows us to now say with certainty that all even numbers reside on worker 0 and all odd numbers on worker 1.

The partitioner we wrote above is terse and only required changing a single line of code but let's take a moment to unravel the details. Given that we instantiate a new operator, it's worth looking at the corresponding trait definition and, in particular, the types used for the the arguments:

```rust
pub trait Exchange<T, D: ExchangeData> {
    fn exchange(&self, route: impl Fn(&D)->u64+'static) -> Self;
}
```

By looking at the implementors of this trait we notice that this interface is applicable to any Timely stream, provided the data is serializable (i.e. implements `ExchangeData`). Further, this operation is applied to an input stream (namely the receiving type: `&self`) and produces a partitioned output stream of the *same* type (as seen in the return value: `Self`). The second argument (`route`) makes use of the so-called `impl Trait` syntax and should be read as being generic over any function that receives a read-only reference to a data element and returns a non-negative number. (Don't pay too much attention to the lifetime annotation (`'static`); it's purpose is to specify that the closure doesn't have any references to the current stack.) This idea of dispatching to some worker index modulo total number of workers fits with the intent of a partitioner. Within our code snippet, we have a stream of pairs with `(round, number)` and so we use pattern matching to destructure the individual parts and use the second element as our routing key. As another point of comparison, the simplest partitioner which ignores the data and always routes to a fixed target can be written as follows: `stream.exchange(|_| 0)`.

In this toy example you could use any hash function but once you start working with real datasets – which often have inherent skew (e.g. think of celebrities in social networks) – partitioning becomes even more essential to balance the load well and keep workers from idling.

3. You're finally ready to count words

Tasks
#1: For the word count task, we provided you with several snippets which can be be assembled into a full program like this:

```
/// More complete example of a 'word count' which maintains state and
performs aggregation

extern crate timely;

use std::collections::hash_map::DefaultHasher;
use std::fs::File;
use std::hash::{Hash, Hasher};
use std::io::{BufRead, BufReader};

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

fn hash<T: Hash>(obj: T) -> u64 {
    let mut hasher = DefaultHasher::new();
    obj.hash(&mut hasher);
    hasher.finish()
}

fn main() {
    timely::execute_from_args(std::env::args(), |worker| {
        let index = worker.index();
        let mut input = InputHandle::new();
        let mut probe = ProbeHandle::new();

        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(key),
                )
                .inspect(move |(word, count)| println!("worker: #{}\
tword: {}\tcount: {}", index, word, count))
                .probe_with(&mut probe);
        });

        // workers each read data from a separate file
        let path = format!("br-{}.txt", index);
        let file = File::open(path).expect("Input data not found in
current directory");
        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();
        }
    }).unwrap();
}
```

#2: The sub-question about how to perform computation continuously (i.e. in batches of 5 lines) caused a fair bit of confusion because we still have a (bounded) text file even though we'd like to treat the input as unbounded. The main idea here was to compare and contrast the effect of processing everything all-at-once (batch) versus progressively (streaming). By breaking the input into chunks we can't lump everything into a single epoch but need to feed some inputs and let the computation run until all these have been fully processed. Below you'll see a rough sketch of what was meant:

```
--- src/bin/part3.rs
+++ src/bin/part3.rs
@@ -43,28 +43,38 @@
         // workers each read data from a separate file
         let path = format!("br-{}.txt", index);
         let file = File::open(path).expect("Input data not found in current
directory");
```

```
          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();
+         let mut lines = buffered.lines().peekable();
+         let mut epoch = 0;
+         // trickle lines in small batches
+         'outer: loop {
+             for _ in 0..5 {
+                 match lines.next() {
+                     Some(line) => input.send(line.unwrap()),
+                     None => break 'outer,
+                 }
+             }
+             // advance input and process
+             epoch = epoch + 1;
+             input.advance_to(epoch);
+             while probe.less_than(input.time()) {
+                 worker.step();
+             }
          }
      }).unwrap();
 }
```

Having now solved the same task with both Flink and Timely, we wanted you to reflect on how the two programs differ syntactically and explain these in terms of differing choices in the underlying system architecture and execution model. To give a few examples consider how ...

- **Dispersed vs. monolithic:** Flink programs are thin clients which assemble the dataflow graph and hand it over to a remote job manager but aren't involved in runtime execution and consequently don't process data tuples, deal with scheduling of tasks, re-scaling actions or fault tolerance. By contrast, a Timely dataflow acts much more like a `TaskManager` in that all these components are bundled together and performed locally.
- **Job submission:** Both systems adopt the same idea of fluent-style API to lazily construct the operator DAG but how execution proceeds thereafter is quite different. In Flink, we have one blocking action in the form of `env.execute()` which hands over control to the system; in Timely, by contrast, the roles are reversed and it is our responsibility to periodically call `worker.step()`. In the first, we leave most freedom to the system to choose (or even adapt) its execution strategy whereas the latter is prescriptive. This has broader repercussions, for

instance it’s common to receive inputs over the network in streaming analytics, and so when using blocking I/O in Timely we’d need to carefully coordinate use of background threads to allow both computation and communication to proceed freely.

- **Receiving input:** With Timely, you most likely interleaved the process of reading text from a file and feeding it into the streaming topology because, as hinted at just before, we drive the computation loop and can intermingle other actions. With Flink, on the other hand, you will recall how we attached special source and sink operator to ingest and collect data (such as the Wikipedia edits connector). The difference may seem negligible but in former case data is being fed into a pipeline whereas in the latter data is generated inline. As a remark, this isn’t a fundamental difference and the pattern of having a source/sink is also possible in Timely though somewhat less common.
- **Task graph:** Yet another point of contrast is that in Timely all workers are alike and so all workers run replicated copies of the entire dataflow graph, whereas Flink’s execution model is task-based and runs distinct stages across the cluster. On a related note, Flink has an explicit in-memory representation of the operator graph and is therefore able to perform optimizations such as fusing operators into chains to reduce intermediate buffers or re-ordering operators to take advantage of sorted data or reduce re-partitioning. Timely is more of a domain-specific language and relies much more heavily on Rust as the host language and its toolchain, for instance, aggressive inlining is essential for the compiler to peer through layered abstractions (e.g. iterators) and generate efficient code. Another example is how Flink is aware of the data schema and partitioning strategy because of the additional information conveyed by an invocation like `.keyBy("someField")`. Timely’s lower-level API allows us to flexibly re-route tuples among workers as we’d like but the system is unaware of the actual data types at runtime because those were expressed via generics and lost during the compilation process.
- **Memory management:** Relying on garbage collection vs. ownership/scoping is arguably more a language-level design choice but this has practical consequences. One example is how the discipline of unique ownership in Rust means that the reference do not form cycles and how this makes serialization simple and fast[1].

1 This blog post details the gory implementation of ‘abomonation’ -- the serialization library used in Timely: http://www.frankmcsherry.org/serialization/2015/05/04/unsafe-at-any-speed.html