The eau2 Big Data System

Introduction

The eau2 system provides an interface for applications to interact with large datasets distributed over a network.

Calculations and permutations of the data are performed in parallel across the nodes on the network.

Architecture

There are three levels to the eau2 system:

1. Applications

The Application layer is user code running on each node which determines the cluster's functionality.

These Applications can define a wide range of functionality, for example counting all the words in a large dataset.

2. KV-Store

The Key-Value store is an abstraction layer on top of the networking layer. In addition to providing the application layer with access to dataframes, it is also used by network nodes to get other data types such as columns from other nodes.

This layer supports methods to operate on a local store, as well as retrieving data from other stores on the network.

3. Networking and Concurrency Control

The network layer has one registrar node and many other worker nodes. The registrar node's purpose is to tell worker nodes about each other, so that they can send data directly between one another.

Each worker node runs an Application. This can do the same thing for all nodes, or can delegate different tasks for different nodes, by switching based on the `this_node()` value.

Implementation

libeau2

Most of the codebase lives in the `libeau2` directory, which is built as a static library. Application code then links to this library and includes the header files in the various `include/` directories.

A quick note about the include directory: only .hpp files should be included in application code. The .tpp files are included at the end of the headers. It is done this way so that the template definitions are shared among translation units.

`libeau2` is further divided into the `database`, `network`, and `serial` directories (and `tests` of course).

/database

The `database` directory contains code for storing and indexing data locally - so it includes classes like `DataFrame` and `Column`, for storing large amounts of arbitrary data, and the `KVStore`/`Key` classes for assigning keys to those `DataFrames`.

Several common typedefs are defined here:

alias	type
`ExtString`	std::shared_ptrstd::string
`DFPtr`	std::shared_ptr
`ColPtr<T>`	std::shared_ptr<Column>
`ColIPtr`	std::shared_ptr
`BStreamIter`	std::vector<uint8_t>::iterator

What follows is a description of the main `database` and ancillary classes.

`ColumnInterface/Column<T>`: A collection of the same type of data.

Interface and generic implementation for columns of data. Any datatype `T` is supported for these classes. Columns are essentially wrappers for a vector of `Chunks` and some serialization methods (these methods will be explained in more detail in the **serial** section of this report).

`Chunk<T>`: a fixed-size collection of data.

Chunks are more-or-less uniformly scattered throughout the network. A single `Column` does not usually have all of its `Chunks` locally stored.

`DataFrame`: A set of data columns.

Although `Column<T>` allows for any templated type, `DataFrames` only support `int`-, `bool`-, `ExtString`-, and `double`-type columns.

Provides the static methods `fromArray(arr)` and `fromScalar(scalar)` for creating single-column and single-element dataframes, e.g. for lists of data or signals/wait variables, respectively.

`Row`: non-owning row of data in a DataFrame

`Rower`: Visitor for an entire row. Can be used with the DataFrame's visit method to permute entire DataFrames.

`Fielder`: Visitor for a single element. Can be used with the Row's visit method to permute entire Rows.

`Schema`: Representation of datatypes in the dataframe.

`KVStore`: the key-value store.

This is the class which the application layer uses to interact with the cluster. Each application gets its own KVStore, but they are linked together over the network so that methods like `waitAndGet` will ask the network for the key's location and transfer the data accordingly, opaque to the application user.

Supports the methods `insert(k, v)`, `waitAndGet(k)`, `fetch(k, timeout)`, and `push(k, v)`. The `insert` and `waitAndGet` methods operate on the local data store. The store utilizes the network layer in order to send/receieve values to/from other stores on the network.

`Key`: the key for KVStores

Just a key and a home node. Used by `KVStore`.

/network

This directory contains classes related to networking. *(these classes are not fully fleshed out)*

`KVNet`: abstract class / interface for a network implementation

This class is basically just a send and receieve method. This could be implemented, for example, by a multi-threaded node system, a multi-node network-based system, or many other implementations.

`KVNetTCP`: Particular implementation of `KVNet` which uses TCP sockets.

This class is what the final eau2 project uses.

`Registrar`: Single-node TCP listening socket for registering nodes

This class is not fully implemented, however, it would represent the master node for the system - dealing with registrations from worker nodes and telling them about each other.

`tcpUtils.hpp`: Adds some wrappers around POSIX sockets.

These methods are added to the `TCP::` namespace. For example, adds a method to sequentially send all elements of a byte vector to a socket.

/serial

This directory contains classes related to serialization and messaging.

`Message`: Message interface class.

Mostly just a container for a type, source, sink, ID, and to/from bytestream methods.

`Serial`: Namespace with a bunch of `canSerialize` templated methods.

This clas is used mainly to determine if certain datatypes can be serialized or not.

`Serializer`: Wrapper for a bytestream with some helper methods.

Allows for adding columns or DataFrames to the bytestream utilizing the `Payload` class.

`Payload`: Creates payloads for messages with automatic serialization.

Passing various datatypes into the Payload's constructor will convert those datatypes into bytestreams which can then be sent on the KVNet.

`Ack, Get, Kill, Directory, Put, etc.`: Particular implementations of Message.

These are used by KVNets to send/receive message to/from the network.

libsorer

This directory is a slightly modified version of https://github.ccs.neu.edu/euhlmann/CS4500-A1-part1.

The modifications allowed their CwC implementation of a SoR parser to be used in our CMake project. The header files were split into .h and .cpp files, and the `cassert`-based tests were converted to use `gtest`. All of their classes were wrapped in a `ne::` namespace to avoid conflicts. Otherwise, it remains the same.

We also added a static `DataFrame::fromColumnSet` method which takes the output of euhlmann's parser (which uses their own Column types) and converts it to our DataFrame. This is wasteful, as we ideally would load the SoRer directly into our own Column types.

Use Cases

Current Demo - Summarizer

For a simple example, an application could split nodes into 'producers' which generate new data and place it in a dataframe, 'counters' which take that data and sum it, and `summarizers`, which wait for the count to be completed and print the final result.

The code for this example can be found in the `demo/` directory.

Open Questions

- Should reimplement euhlmann's SoRer using our Column classes so we can read a .sor file directly into a DataFrame, as opposed to copying their columns.
- The network layer sometimes has trouble shutting down

Status

- Codebase was restructed for a proper C++ project.
- Parsing .sor files into DataFrames works. The test for this was disabled because the net code will not die.
- DataFrame methods work in isolation
- The TCP network is implemented.
- The M3 demo works.
- KVStores work as expected
- Serialization is working for pretty much everything
- Testing needs to be improved:
  - We are missing unit tests for some classes.
- There's a bit of inconsistency in the codebase regarding style and ownership.