CS3AC16: Cloud Computing 2016

Coursework: Passenger Airline Flights - Deyan Dyankov (24885531), March 2017

Introduction

The purpose of this assignment is to develop a non-MapReduce executable prototype. The objective is to develop the basic functional 'building-blocks' that will support the development objectives described below, in a way that mimics something of the operation of the MapReduce/Hadoop framework.

High level description

The MapReduce framework described in MapReduce: Simplified Data Processing on Large Clusters is a programming model for processing and generating large datasets. Users specify a mapper and a reducer function that are run in parallel, generally on top of a cluster of servers.
The current implementation for this assignment is designed to emulate the same behaviour but on a single server, using its CPU cores for parallel processing. Before the parallel processing begins, the input file is split into *n* parts based on number of available CPU cores. Once the file is split, the program spawns *n* processes and each process runs the user defined mapper function on its own part of the input file. The output of these processes is also saved in chunks and they are later reduced and combined to return a result. The framework allows the user to define their own mapper, reducer, combiner functions as well as necessary data types.

Programming Language

For this assignment v0.5 of the Julia programming language has been used. Julia is a high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments like MATLAB. It provides a sophisticated just-in-time compiler based on LLVM, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

Version control

The implementation of this assignment has been designed to work as a Julia package called `nmr` for "naive mapreduce" and is available on *github* under the following address `https://github.com/deyandyankov/nmr`
This means that once Julia is available on the user's computer, the installation of the package is as easy as running `julia` and executing the following: `Pkg.add("https://github.com/deyandyankov/nmr")`

Description of the NMR package

Julia packages usually (althought not necessarily) follow a similar structure which is easy to navigate. The following directory tree shows where code, tests, data and outputs reside.

```
!dev ~/repos/nmr> tree .
├── LICENSE.md
├── README.md
├── REQUIRE # required modules dependencies that are preinstalled when installing the package
├── src # directory with actual package source code
|   ├── io.jl
|   ├── logging_config.jl
|   ├── nmr.jl # main point of the package
|   ├── phases.jl
|   ├── types.jl
|   └── udf # user defined functions and data types
|       ├── combiners.jl
|       ├── joiners.jl
|       ├── mappers.jl
|       ├── reducers.jl
|       └── types.jl
├── test # directory with tests
|   ├── data # directory with assignment data files
|   |   ├── AComp_Passenger_data.csv
|   |   ├── AComp_Passenger_data_no_error.csv
|   |   ├── AComp_Passenger_data_no_error_DateTime.csv
|   |   ├── README.md
|   |   └── Top30_airports_LatLong.csv
|   ├── output # output directory for the mapreduce jobs
|   |   └── README.md
|   ├── runtests.jl # main point for running tests
|   ├── split # this is where files are split and processed
|   |   └── README.md
|   ├── test_lineofsight.jl # Code for the Line of Sight objective
|   ├── test_listofflights.jl # Code for the List of Flights objective
|   ├── test_numberoffights.jl # Code for the Number of Flights objective
|   └── test_numpassengers.jl # Code for the Number of Passengers objective
└── versions.js
!dev ~/repos/nmr>
```

When the package is loaded, the code in `src/nmr.jl` is executed which in turn loads the rest of the code form the `src` and `src/udf` directories.
The assignment's objectives are designed and coded as tests and reside in the `test/test_*.jl` files respectively. Once `julia` is ran, the user can install the package and run the tests which effectively performs the map, reduce, join and combine operations necessary for the successful achievement of the objectives. The main point for the tests is the `test/runtests.jl` file which runs all other tests.

Tests workflow

Let's examine and discuss the contents of `test/runtests.jl` which will give us a better understanding of how the tests are executed and how the module is used. This file is the main test entrypoint and it runs all of the assignment's objectives.

```
using Base.Test
using FactCheck
using JSON
using nmr # load the nmr module

# add as many processes/workers as there are CPU cores
addprocs(Sys.CPU_CORES)

# go into the package's test directory
@time cd(joinpath(Pkg.dir("nmr"), "test")) do
  function run_test(testfile)
    info("Running test file $(testfile)")
    include(testfile)
  end
  # run all files that match "test_*.jl"
  testfiles = [f for f in readdir(".") if isfile(f) && startswith(f, "test_") && endswith(f, ".
jl")]
  map(run_test, testfiles)
end
```

The code above is simply a wrapper for the rest of the objectives that are implemented in `test_*.jl` files. There are, however, two main aspects of it:
- `using nmr` -- this loads the naive mapreduce framework
- `addprocs(Sys.CPU_CORES)`

On a server with `4` CPU cores, `Sys.CPU_CORES == 4`; Running `addprocs(Sys.CPU_CORES)` essentially spawns `4` more `julia` processes that are later used to run the user defined mapper and reducer functions that operate on parts of the input files. The main process ends up being a coordinator.
The `nmr` module is designed to work with single and multiple core processors. This means that if we do not execute the `addprocs()` function, our mapreduce jobs will simply run using a single worker. This is an important aspect of the module as it allows the user to run the same mapreduce jobs on different servers, with no parallelism or with hundreds of workers -- without changing their code.

Objective #1: Number of flights

The code below describes what the user code would look like. We will then look under the hood of the `nmr` module and discuss how the framework works, but let's first focus on the user code. Mind that before this chunk of code is executed, `runtests.jl` has already loaded the module and ran `addprocs(Sys.CPU_CORES)` which has added the necessary workers.

```
# contents of test_numberofflights.jl
nmr.runjob(nmr.NMR(1, ["AComp_Passenger_data.csv"], nmr.mapper_parserecordacomp, "acomp.csv"))
nmr.runjob(nmr.NMR(1, ["acomp.csv"], nmr.reducer_numberofflights, "acomp_flights_reduced.csv"))
c = nmr.runcombiner("acomp_flights_reduced.csv", nmr.combiner_parsejson)

@test typeof(c) == Vector{String}
@test length(c) == 65
@test JSON.parse(c[1])[1] == "AMS"
@test JSON.parse(c[1])[2] == 3
```

Let's go over the code line by line. The first line:

`nmr.runjob(nmr.NMR(1, ["AComp_Passenger_data.csv"], nmr.mapper_parserecordacomp, "acomp.csv"))`

This creates an instance of `nmr.NMR` which is a composite type and passes it to `nmr.runjob()`. The NMR composite type is comprised of job id (1), input files ("AComp_Passenger_data.csv"), a mapper/reducer (nmr.mapper_parserecordacomp) and an output file (acomp.csv).

Executing this line spawns a mapreduce job with id=1 that runs the `nmr.mapper_parserecordacomp` on top of `AComp_Passenger_data.csv` and stores the results in `acomp.csv`.
Once the mapping has been done and results stored in `acomp.csv`, the user can execute the reducer `nmr.reducer_numberofflights` on top of `acom.csv` (mapper's output) and respectively store its output in `acomp_flights_reduced.csv`. However, both `acomp.csv` and `acomp_flights_reduced.csv` might be split based on number of CPU cores. Therefore we need to run:
`c = nmr.runcombiner("acomp_flights_reduced.csv", nmr.combiner_parsejson)`
which combines the results from all reducers and in the current scenario returns an array of `JSON` objects with the output.
At this point we are ready to do some sanity checking using the `@test` macro.
The user is then free to store the contents of `c` in a file which is essentially the output of the mapreduce job, combined.

Module workflow

The NMR module's main function is `runjob`, accepting a `NMR` data structure which has the input data, the user function that processes the data and an output file. The code for the `runjob` function is really straightforward and simple:

```
function runjob(j)
  info("Running job: $j")
  split_raw_data(j)
  create_job_area(j)
  phase_run(j)
  ctx = copy(default_ctx)
  return true
end
```

Splitting the input files and creating the job area

The `split_raw_data()` function is used to split the input data file into *n* parts that are later going to be processed by the mappers defined by the user. The code for this function is in the `io.jl` file. It works by first opening the input file for reading, then opening *n* number of files to write the *n* parts into.
The `create_job_area()` function simply creates output directories for each worker if they do not exist. For example if the user has requested `4` workers, then four directories will be created within `test/output`.
The code for the `split_raw_data()` function is provided below:

```
function split_raw_data(j)
  # operate on each input file as they may be more than one
  for filename in j.inputs
    wrkrs = workers() # how many workers are we dealing with?
    input_filename = joinpath(datadir, filename)
    !isfile(input_filename) && return true # not in data/ but already in split/
    info("Splitting $input_filename using $(length(wrkrs)) workers")
    output_filenames = Array(String, length(wrkrs))
    for i in 1:length(wrkrs)
      output_dir = joinpath(splitdir, string(wrkrs[i]))
      !isdir(output_dir) && mkdir(output_dir)
      output_filenames[i] = joinpath(output_dir, filename)
    end
    # open the files for writing (1 file per core)
    handles = [open(f, "w") for f in output_filenames]
    handle = 1
    for ln in readlines(input_filename)
      current_output_file = output_filenames[handle]
      # println("current handle: $(handle) and current output filename $(output_filenames[handl
e])")
      current_handle = handles[handle]
      write(current_handle, ln) # write line into current handle
      handle += 1 # get next handle
      if handle > length(handles)
        handle = 1 # cycle back to first handle once handles have been exhausted
      end
    end
    map(close, handles) # close all output handles
  end
  return true
end
```

Running the job

The `phase_run()` function is where the actual execution is done. The function is defined in the `phases.jl` file. Let's review this whole file and go over its functinons:

```
function phase_run(j)
  info("initiating run phase")
  r = [@spawnat w runfun(j) for w in workers()]
  map(wait, r)
  info("run phase finished")
end
```

Here, `workers()` returns an array of integers `2, 3, 4, 5` in case we are dealing with four workers. The code above spawns `runfun(j)` on each worker and waits for each worker to finish their execution. The `runfun()` function itself is then executed on each worker and each worker operates on the specified part of the input file. For example `runfun()` on worker `2` will operate on the `2` part of the input file.

Below is the code for the `runfun()` function. It is very simple and it expects that `j.inputs` is an array of one or two input filenames. If the input file is only one, `runfun()` branches into `runfunsinglearg()` or if the input files are two, it goes to `runfunmultiplearg()`.

```
function runfun(j)
  info("runfun worker $(myid()) initiated")
  length(j.inputs) > 2 && throw(ArgumentError("Cannot operate on more than two inputs."))
  length(j.inputs) == 1 && runfunsinglearg(j)
  length(j.inputs) == 2 && runfunmultiplearg(j)
  info("runfun worker $(myid()) finished")
end
```

Given that the input file is only one, it is processed line by line and the user function is applied on every line. It is important to keep in mind that each worker runs its own instance of `runfun()` and therefore its own instance of `runfunsinglearg()` that operates on this worker's chunk of the file. This is how the parallelisation is done.

The function is relatively simple. It opens the input part of the file, reads it line by line, applies the user function on each line and if a result is returned it is being written to an output file. If a mapper encounters an exception, a warning is issued but processing continues.

```
function runfunsinglearg(j)
  info("runfunsinglearg $(myid()) initiated")
  input_filename = j.inputs[1]
  inputfile = joinpath(splitdir, string(myid()), input_filename)
  io_input = open(inputfile)
  io_output = write_sink(j)
  for (linenum, line) in enumerate(eachline(io_input))
    try
      v = j.fun(line)
      v == "" && continue
      write(io_output, json(v) * "\n")
    catch e
      if typeof(e) in [MapperException, UDFException]
        warn("[JOB $(j.jobid)] mapper encountered an exception of type $(typeof(e)) on line $(l
inenum): $(e.msg)")
      else
        rethrow(e)
      end
    end
  end

  if length(ctx) > 0
    for (k, v) in ctx
      write(io_output, json((k, v)) * "\n")
    end
  end
  close(io_input)
  close(io_output)
  info("runfunsinglearg $(myid()) finished")
end
```

If instead the user has provided two input files, they are processed using the `runfunmultiplearg()` function in a nested-loops fashion. This means that if the user provides input files `A` and `B` each of which has `40` lines, the user function will be applied `40 * 40 = 1600` times in total. This is useful for implementing joins and linking two input files as used in Objective #4.

```
function runfunmultiplearg(j)
  info("runfunmultiplearg $(myid()) initiated")
  input_filenamex = j.inputs[1]
  input_filenamey = j.inputs[2]
  io_output = write_sink(j)
  inputfilex = joinpath(splitdir, string(myid()), input_filenamex)
  for worker in workers()
    inputfiley = joinpath(splitdir, string(worker), input_filenamey)
    io_inputy = open(inputfiley)
    io_inputx = open(inputfilex)
    for (lineynum, liney) in enumerate(eachline(io_inputy))
      for (linexnum, linex) in enumerate(eachline(io_inputx))
        try
          v = j.fun(linex, liney)
          v == "" && continue
          write(io_output, json(v) * "\n")
        catch e
          if typeof(e) in [MapperException, UDFException]
            warn("[JOB $(j.jobid)] mapper encountered an exception of type $(typeof(e)) on line
 $(linenum): $(e.msg)")
          else
            rethrow(e)
          end
        end
      end
    end
    close(io_inputy)
    close(io_inputx)
  end
  if length(ctx) > 0
    for (k, v) in ctx
      write(io_output, json((k, v)) * "\n")
    end
  end
  close(io_output)
  info("runfunmultiplearg $(myid()) finished")
end
```

The code showcased above is the backbone of the framework. The functions and data types that will be discussed in the next sections are user defined and they contain the business logic that implements the objectives.

Data types, error handling and user defined functions

For achieving the objections, several entities have been defined. They are split into mappers, reducers, combiners and data types and are defined in the `src/udf` directory respectively.

Data Types

Since the rows in the `AComp_Passenger_data.csv` file all follow the same structure, a single data type has been defined to hold a record of type passenger data:

```
type UDFRAcomp
  passengerid::String
  flightid::String
  originairport::String
  dstairport::String
  departuretime::DateTime
  arrivaltime::DateTime
  totalflighttime::Int16
end
```

Similarly to C/C++, Julia allows us to create composite data types. This particular data type called `UDFRAcomp` has all the attributes that are present in a passenger data record. When reading the `AComp_Passenger_data.csv` file, every line is parsed into a `UDFRAcomp` record.

Error handling

Several other data types are used when parsing the data. An important side effect of those data types is that an error is thrown on instantiation in case the data fails to meet certain constraints. Let's take for example the `UDFAirportName` data type as defined below:

```
function UDFAirportName{T<:AbstractString}(value::T)
  length(value) < 3 && throw(UDFException("Airport Name must be longer than 3 characters: $valu
e"))
  length(value) > 20 && throw(UDFException("Airport Name must be shorter than 20 characters: $v
alue"))
  value
end
```

When the user tries to instantiate a `UDFAirportName` using:

```
f = UDFAirportName("AB")
```

an error is thrown letting the user know that this particular airport name is invalid.
Those errors are handled by the framework and only a warning is issued. Processing of the mapreduce job does not stop in case a type fails to instantiate.

User defined functions

Several mappers, reducers and combiners are predefined for the user's convenience.
The main mapper that is in use is the following:

```
function mapper_parserecordacomp(x::AbstractString)
  line = chomp(x)
  separator = ","
  isempty(line) && throw(MapperException("line is empty"))
  s = split(line, separator)
  length(s) < 6 && throw(MapperException("malformed line has less than six elements when split
by ,"))

  departuretime = UDFDepartureTime(s[5])
  totalflighttime = UDFTotalFlightTime(s[6])
  arrivaltime = departuretime + Dates.Minute(totalflighttime)
  UDFRAcomp(
    UDFPassengerId(s[1]),
    UDFFlightId(s[2]),
    UDFAirportCode(s[3]),
    UDFAirportCode(s[4]),
    departuretime,
    arrivaltime,
    totalflighttime
  )
end
```

When processing a line from the `AComp_Passenger_data.csv` file, this mapper returns a structure of type `UDFRAcomp` if the record is successfully parsed and all its attributes are valid.

An example reducer is provided below:

```
function reducer_numberofflights(s)
  j = JSON.parse(s)
  fromairport = j["originairport"]
  ctx[fromairport] = get(ctx, fromairport, 0) + 1
  return ""
end
```

Instead of returning anything, this reducer is using a `ctx` dictionary to store counters against airports - a classic wordcount example.

Objectives

The assignment's objectives are briefly described here.

Objective 1

The code for the first objective is in `test/test_numberofflights.jl` ; This involves a mapper, reducer and a standard combiner function. Output is available in `objectives/numberofflights.json`

Objective 2

The code for the second objective is in `test/test_listofflights.jl` ; This involves only a mapper and a standard combiner function. Output is available in `objectives/acompdata.json` .

Objective 3

The code for the third objective is in `test/test_numpassengers.jl` ; This involves a mapper, reducer and a standard combiner function. Output is available ina `objectives/numpassengers.json` .

Objective 4

The code for the third objective is in `test/test_lineofsight.jl` . Output is available ina `objectives/lineofsight.json` .
The heavy lifting is done here which is implementing joins. The join method is available in `src/udf/joiners.jl` and it uses two input files as described in `Module Workflow` section of the document. However, nautical miles per flight and passengers were not calculated due to lack of time on student's part. The prerequisites are achieved: flights' origin and destination airports are extracted using the join method and from then on it is a matter of fetching the coordinates of the respective airports coordinates and calculating the distance between them.

Self-appraisal

The assignment has been very interesting and a great opportunity to create a prototype of a mapreduce system. With some more work and networking code, it can be extended to multiple servers.
The code was very carefully designed and a lot of time was spent on it. On the contrary, this particular document was created in the last minute and it can definitely be improved.
For a single server, this mapreduce framework would work nicely and it can process large amounts of data that do not fit into memory - because of the way mappers and reducers are implemented. The more cores a server's cpu has, the faster the processing becomes and the less the overhead of synchronisation and combination is.
One significant drawback of the framework is that at the moment mappers, reducers and datatypes need to be defined within the module. Instead, the module should enable the users to define those functions outside the module and pass them ot it. This is achievable using the `@everywhere` macro in julia which would execute code on all workers. Due to time constraints, this was not implemented.