SpaceCoMP Programming Model

This document describes the design of the SpaceCoMP Space Collect MapReduce Processing Model from an application programmer's perspective.

Overview

The MapReduce model has been widely adopted as a primitive for parallel computing of data-intensive workloads in compute clusters. Mappers process data in parallel and Reducers consolidate, sort and aggregate the output from mappers. The MapReduce primitive is popular as it shields the application programmers from decisions and complexities around scheduling, orchestration, reliability and data locality. It also works equally well with simple one-off batch jobs as well as complex streaming data processors in complex hierarchical DAG workflows.

The idea with SpaceCoMP is that in space, or more precisely in ISL-connected mega-constellations of satellites in LEO orbits, sensor data is generated in a distributed pattern as well. The data production may for instance be correlated with a projection of an area of interest on Earth to observe something like a developing forest fire. Due to orbital dynamics, the satellites that cover this area of interest (AOI) changes constantly but predictably. When orchestrating sensor data collection and processing nodes, this dynamics has to be taken into account to avoid excessive data shuffling. As the satellites move in their orbits the orchestration and scheduling decisions have to be re-evaluated. Communication in ISL mesh networks follow a hop-by-hop pattern best described by Manhattan distances. Thus distances between nodes both in terms of hops and physical distances of each link for each hop has to be considered. Routing decisions also need to consider that there are orbital cross-over events where some links are unreliable. Another fundamental difference to traditional MapReduce processing in ground data centers is that scheduling decisions need to be taken in a decentralized manner as many concurrent jobs may be injected from any satellite in a constellation that is visible from a point on the ground.

The fundamental principles of SpaceCoMP are:

- Collectors are scheduled based on predicted current AOI projection (and load)
- Mappers are scheduled based on cost of transmitting and processing data from collectors
- Reducer is scheduled based on proximity to mappers (and load)

Application Programmers may provide custom or use generic Collector, Mapper and Reducer processors described below.

To simplify the discussion we assume here that these components are implemented in Python and that the payloads being passed follows a JSON format (with the exception of binary data which is streamed separately).

Collector

The collectors implement a collect method with a JSON payload input parameter

```
def collect(payload):
```

A payload meta_data field contains meta_data that travels with the job through all phases including:

data_id	Unique index across all collectors
data_size	Total number of collectors
job_start	Timestamp when job was submitted
jobid	Globally Unique Job Identifier
max_collect	How many collect tasks should be buffered before streaming to mapper
job_data	Application specific data

When a collector has collected a data record it is emitted with a yield statement, meaing it is ready to be processed by a mapper:

```
yield data
```

The data object may by any serializable valid JSON.

If binary data is collected, e.g. an image needs to be sent to mappers for processing the collector should return the following structure:

```
yield {"value": data, "_COMP_FILE_": {"name": fname, "stream": stream}}
```

Where fname is the filename and the stream is the binary file object stream. The binary stream can simply be an open file or be created from a bytearray with e.g:

```
stream = io.BytesIO(byte_arr)
```

The basic design is that the collector can focus on expensive IO processing and maybe some initial pre-processing of sensor data before streaming records to a mapper for more elaborate processing. That way the collection and pre-processing can be parallelized across data records even within the same collect/map pair.

Mapper

The mappers implement a run_map method with a JSON payload input parameter

```
def run_map(payload):
```

This payload parameter has the same meta_data field as described above for the collector as well as:

data	The record the collector produced
files	Dictionary with keys being filenames of binary data set in the collector. The value is a stream that can be read from just like a local file.
end_collect	Boolean set to indicate whether this is the last record produced by the collector
collected_index	The ordering number of record collected
collector	The satellite that collected the data

Just like with the collector both basic JSON and binary streams can be emitted for reduce processing with a yield statement.

Reducer

The reducer implement a reduce method with a JSON payload array input parameter

```
def reduce(payloads):
```

A payload is one map record and there may be many map records for each mapper. The last record from a mapper is marked with an end_map flag similar to the end_collect flag described above. The data received for each payload mimics the mapper structure:

data	The record the mapper produced
files	Dictionary with keys being filenames of binary data set in the collector. The value is a

	stream that can be read from just like a local file.
end_map	Boolean set to indicate whether this is the last record produced by the mapper
mapped_index	The ordering number of record mapped
mapper	The satellite that mapped the data

The reducer can produce JSON data or binary stream output the same way as the collectors and mappers but instead of yielding or streaming records it simply returns the result, e.g.:

```
return {"value": data, "_COMP_FILE_": {"name": fname, "stream": stream}}
```

Next, to illustrate how the processors above can be used to implement different applications we provide a “hello world” example (Word Count) and a couple of examples related to Earth Observation.

Word Count

To illustrate the simplest possible SpaceCoMP application we implemented the classical word count example. A mapper reads a chunk of a file and streams it line by line to the mapper that counts the words and streams the word counts to a reducer that produces the final word counts across all mappers.

See:
https://github.com/LeoDOS-Project/leopymr/blob/main/docker/collectors/doccollector.py
https://github.com/LeoDOS-Project/leopymr/blob/main/docker/mappers/wordcountmapper.py
https://github.com/LeoDOS-Project/leopymr/blob/main/docker/reducers/sumreducer.py

SAR Example

Synthetic Aperture Radar (SAR) Images are popular in satellite observation applications as the pictures produced are less susceptible to atmospheric distortion and cloud obstruction.

The images are however large in size (up to 7GB for a single image) and some denoising is necessary.

In our example a custom mapper denoises SAR TIFF images using a deep (neural network) despeckling tool, and the denoised images are then sent to a custom mapper that does object detection with computer vision. The CV output (count of objects detected) is then sent to a

simple reducer that just takes the counts from all mappers and aggregates them to produce the final results.

More info at:
https://github.com/LeoDOS-Project/leopymr/blob/main/usecases/sar/README.md

MISR Example

Multi-Image(Frame) Super Resolution is a popular image processing algorithm used in Earth Observation use cases to combine many lower resolution images taken by satellites into a single higher resolution image.

In our example a custom collector reads a burst of images in dng format and converts them to png before streaming them to a custom mapper that implements the algorithm used in Android to create higher resolution images from a burst. The mappers then forwards the combined image to the reducer that takes all the images from the mapper to produce the final merged image output using the same algorithm.

More info at:
https://github.com/LeoDOS-Project/leopymr/blob/main/usecases/misr/README.md