[CPARK] Software Design Document

Group members: Jiakai Xu (ax2155), Xintong Zhan (xz3165), Shichen Xu (sx2314)

Section 1 - Project Description

In the current era of big data and big models, where more and more tasks rely on parallel batch processing of given data, thus, a light-weighted, distributed (or parallel) computing framework for C plus plus that offers a fast and general-purpose large data processing solution is crucial in our life. We got the project name from Apache Spark, and we aim to build a C plus plus version of a large data processing framework, which would behave similarly to Spark with parallel processing (similar to the cluster structure in Spark), immutable data (similar to RDD in Spark), and lazy evaluation (similar to transformations in Spark).

In other words, we want to design a convenient, general, and high-level parallel computing framework that automatically splits complex computing problems into sub-tasks and then intelligently assigns, executes, and combines the sub-tasks over parallel threads (or even distributed machines). Users benefit from the acceleration, scalability, and fault tolerance of parallel/distributed computing without the pain of dealing with any low-level details other than the problem itself. Lazy evaluation will be applied during the execution to reduce the usage of memory and computing resources.

Section 2 - Data Model Design

The CPARK library uses a Resilient Distributed Datasets (RDD) data model. An RDD is a read-only, partitioned collection of records. It is a memory abstraction that distributes a set of lazily evaluated data into several sub-partitions, allowing the sub-partitions to be actually computed by different threads, and letting programmers perform efficient parallel computing in a more convenient and less error-prone manner.

More formally, an RDD is an input_range that contains several splits. A split is a lazily evaluated collection of records. The records (or elements) contained in a split can be of any type. RDDs can be created through the operations by either (1) reading data from any existing views (2) generated by the custom functions defined by users (3) applying transformations to existing RDDs.

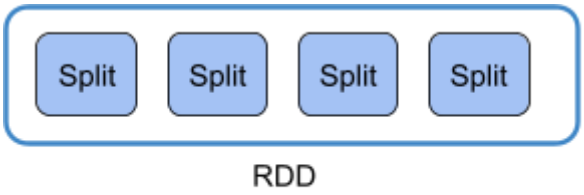
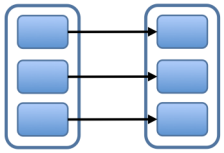


Figure: an RDD is a range of Splits

Transformations of RDDs means conceptually applying some operations to the elements in an existing RDD, and creating a new RDD that (conceptually) contains the transformed data.



Common transformations including filter, transform, flatMap, union, zip, etc. CPARK library provides an aboundant set of transformations.

The transformations of RDDs can be applied and chained for any number of times or from any RDDs. An existing RDD can create any number of different new RDDs by various transformations, and several existing RDDs can create a new RDD by transformations such as union or zip. Thus, a typical workflow in CPARK would form a directed acyclic graph (DAG), where the vertices are the RDDs (and the transformed data inside it), and the edges are transformation operations and dependency relationships between the RDDs.

Then the RDDs are created and the DAG is formed, the actual computation task is not started. It will wait until some actions are called by the users to start the actual evaluation, and only the part of data needed for the action will be evaluated.

This lazy evaluation feature of CPARK empowers the ability to do global optimization and refinements for the actual computing tasks. The CPARK library can decide the order of computation and whether to cache the intermediate results in a global scope.

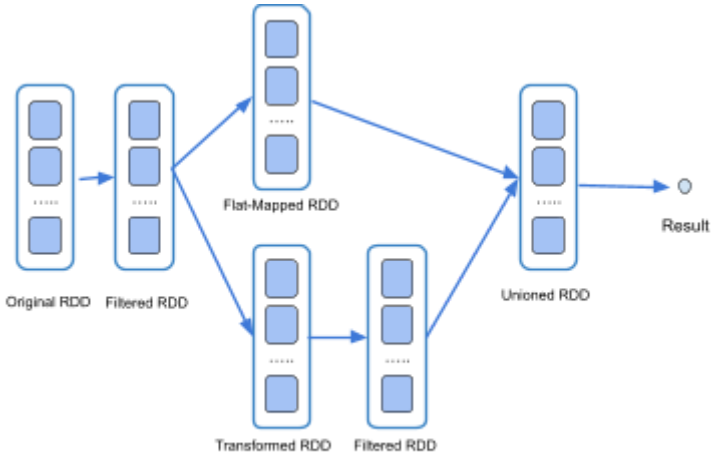


Figure: A task graph formed by RDDs and Transformations

Section 3 - User Interface Design

The CPARK library provides a generic programming styled interface, and an abundant set of concepts that make the user program more readable and clear. It is completely compile-time polymorphism. The CPARK library supports and works well with the C++ standard library, and follows its interface style.

3.1 RDD Interface Design

As described in Section 2, an RDD is a range of splits. More precisely, every RDD in CPARK library is a view of splits with random access ability. Users can apply most of the helper functions in standard ranges library to an RDD.

There are generally two categories of RDD in CPARK library: the creation RDD and the transformed RDD. A creation RDD is those who are created directly and do not rely on other RDDs. These RDDs can serve as the original vertex of a DAG task graph. Examples are PlainRDD (which reads data from any view that is not an RDD) and GeneratorRDD (which creates elements by the functions specified by users). The other category of RDD is transformed RDD, which is created by applying some operations to one or several existing RDDs. These RDDs can serve as the intermediate vertices or final vertices of the task DAG graph.

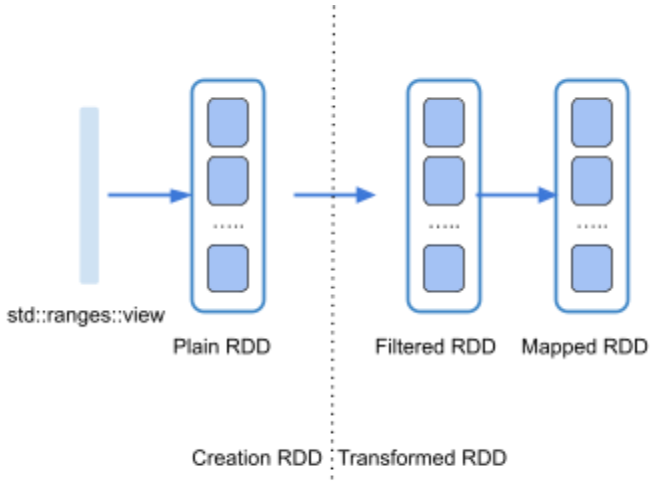


Figure: Difference Between Creation RDD and Transformed RDD

The RDDs (and all other classes or components in CPARK library) uses compile-time static polymorphism (instead of runtime polymorphism) to achieve better performance and better type safety. Typically RDDs created by different transformation types, by different transformation functions, or from different types of previous RDDs, will have different types. For example, the type of a transformed RDD will carry the following information:

(1) what type of transformation it is (filter, map, flatMap, or others)

(2) The type of the transformation function (like the filter function for filter transformation). It can be of any callable object type that fits the type requirements (e.g. a function, a function pointer, a lambda expression, a class with operator(), etc.).

(3) The (exact) type of the previous RDD.

There is a common RDD concept which fits different types of RDDs. Specifying the full type of an RDD can be long and clumsy sometimes, so idiomatically users should use auto or concept::RDD auto to declare a variable for an RDD.

As previously mentioned, an RDD is a view of Splits. It has public interfaces begin() const and end() const that returns a pair of random access iterator and its sentinel. Typically, when a user is given an RDD, he or she can do the following three things: (1) Create a new transformed new RDD from this

existing RDD, (2) Call an action upon this RDD to do some kind of actual computing and get the final result (3) Get the splits from this RDD and use the splits to implement the user's own parallel computing patterns or tasks.

An RDD is a view, so it will have constant time copy and move operations. A new copied RDD will conceptually have the same elements as the previous one.

3.2 Transformation Interface Design

A transformation turns an existing RDD to another new RDD containing the (lazily evaluated) new elements, with the elements in the previous RDD unchanged.

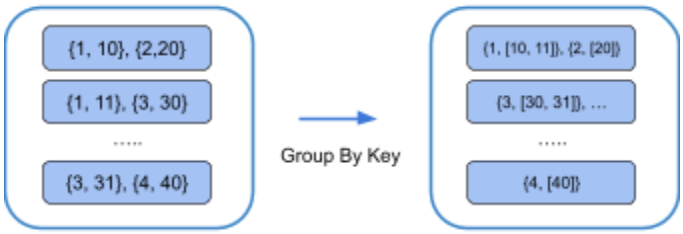
Users can create a transformed RDD by either directly calling the constructor of the corresponding RDD type, or using a pipelined operator | with a transformation calculator. This ensembles what users do with the range adaptors in the standard ranges library. Users will have a smooth and painless experience with CPARK if they are already familiar with C++ standard ranges library.

Most of the commonly used transformation operations are provided in CPARK library, including map, flatMap, filter, sample, groupByKey, union, zip, and so many other ones.

To specify a transformation function for a transformation operation (for example, the function that filters elements by receiving an element as a parameter and returning a boolean value in filter operation), users can pass in any callable objects of any possible types as long as it fits the requirements for the argument type and return value type. This callable object can be a function, a function pointer, a lambda expression, a std::function object, or a class object that overrides the operator().

Note that there are generally two categories of transformations: The first category is transformations who apply to elements or records as one single type; The second category is those transformations who only make sense for elements or records of key-value types (i.e. Types that are pair-like, with a first and a second member that gets its key and value). Examples for the first category include map, filter, flatMap. Examples for the second category include groupByKey, crossProduct.

RDD and Transformation with Elements of Int Type



RDD and Transformation with Elements of pair<int, int> type

3.3 Actions Interface Design

An action starts the actual computation task to a lazily evaluated RDD and gets the final result based on the parallel policy chosen by the user. It marks the end of the CPARK task graph. Before an action is invoked, the CPARK task graph is resilient and has low CPU cost. After an action is invoked, the CPU-intensive tasks will start.

Currently, the actions include reduce, count, and collect.

Users can invoke an action by either directly calling the constructor of the corresponding action, or using a pipelined operator | with a helper action calculator.

3.4 Split Interface Design

A split is conceptually a sub-partition of an RDD, and it is actually implemented as a view of elements or records. Users can apply all of the functions and operations in the standard ranges library that fits a view to a split.

Except for the features of a view of elements, splits also have some dependency relationships with each other.

Precisely speaking, if the evaluation of the elements in a split would need the values of the elements from another split, we say a split is DEPENDENT on the other split. A transformation of RDD must cause the splits in the new RDD to be dependent on some of the splits in the previous RDD. For example, in a map transformation, to calculate the values of the elements in a new split, we must know the values of the elements in the corresponding previous split. Only then can we apply the map function to those elements and create the new elements.

Technically speaking, it is not necessary to expose the split interface to the users. The users would already have the ability to do the parallel tasks given the RDD interfaces and action interfaces. However, we want to enable the users to define their own parallel computing pattern beyond the transformations and actions provided in CPARK library (though we think they are already enough for most of the tasks), so we choose to expose the splits as a public interface to the users.

3.5 Configuration

Users can specify their own configurations for the parallel computing tasks by setting the corresponding fields in the Config class. Configurable items including the parallel policy (single-thread or multi-thread), the level of concurrency, the debug name, the error logger, and others.

It is the user's responsibility to make sure the configuration object does not go out of lifecycle before the computing task is finished,

3.6 Execution Context

An execution context (or environment) for a set of CPARK tasks to run. It contains the information needed to evaluate the Rdd-s and run the CPARK tasks, including the id information of Rdd-s and Splits, the cache information, the thread synchronization information, and the scheduler information. Each Rdd and Split will be included in one and only one execution context.

Generally the user does not need to know much about the execution context. The only thing they should do with execution context is, the users are responsible for creating the execution context, make sure the RDDs in the same task graph will have the same execution context, and make sure the execution context does not go out of lifecycle before the task is done.

Users only need to explicitly pass execution context to the creation RDDs. Note that a task graph may have several RDDs, so users should be careful to pass the same execution context to those creation RDDs, and make sure the execution context is valid during the computation.

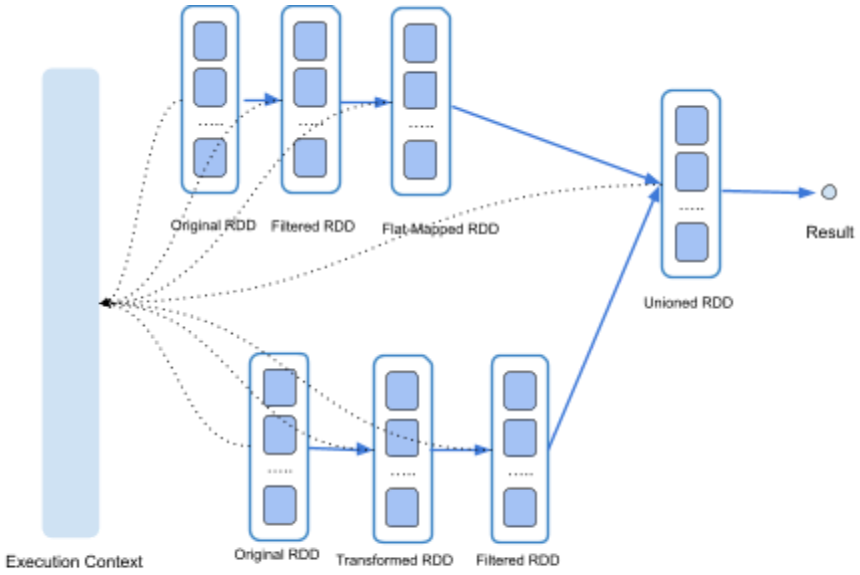


Figure: All RDDs in a task graph rely on one single execution context

Section 4 - Implementation Design

4.1 Global View

The diagram below shows a rough overview of the components that make up the whole CPARK implementation.

From a high point of view, a complete CPARK task graph is made up of the RDDs (as vertices), the transformations (as edges), the action, and the execution context. Each RDD consists of its splits, and each split contains the elements and some meta information. The execution context contains the meta information and the runtime information for the whole task graph.

The split contains the following information: (1) elements, which are not the actual values of the elements, but only a pair of iterators that represents the range of the real elements. Only when iterating and dereferencing the iterator, will the real values of the elements be computed. (2) Split id. Each split will have one unique ID within the scope of the execution context. The id will be assigned to the split when it is constructed in a thread-safe manner. The ID is mainly used for indicating the dependency relationship between the splits and handling the cache logic. (3) The cache logic. The cache logic indicates how this split will be cached, the type of the cache, and when should the split read value from

the cache. (4) The computation logic. It indicates how the values of this split should be computed from the values of its dependency split(s). (5) The context ptr. Each split will also have a point pointing to the execution context.

The execution context has the following components and responsibilities: (1) The configuration, which specifies the parallel task number, the debug name of this task graph, and other configurable options. (2) The split ID and RDD ID information. The execution context keeps track of the ID of all the splits and RDDs in the graph. (3) The split dependency information. The execution context knows about the whole dependency information of every split, so it can do some global optimization based on this information. (4) The cache. The cache stores the cached value of the splits whom the execution context decides to cache. (5) The thread synchronization information. It coordinates the threads (scheduling the threads, waiting for values, waiting for cache, etc.) when the actual tasks are being executed.

Note that there are two types of different information: the creation information and the runtime information. The creation information are those who are created when the RDDs are constructed. This includes the RDD and split ID, the dependency relationship, the configuration. The runtime information are those who are created and needed when the actual computing tasks are being executed, including the cache, the thread synchronization information.

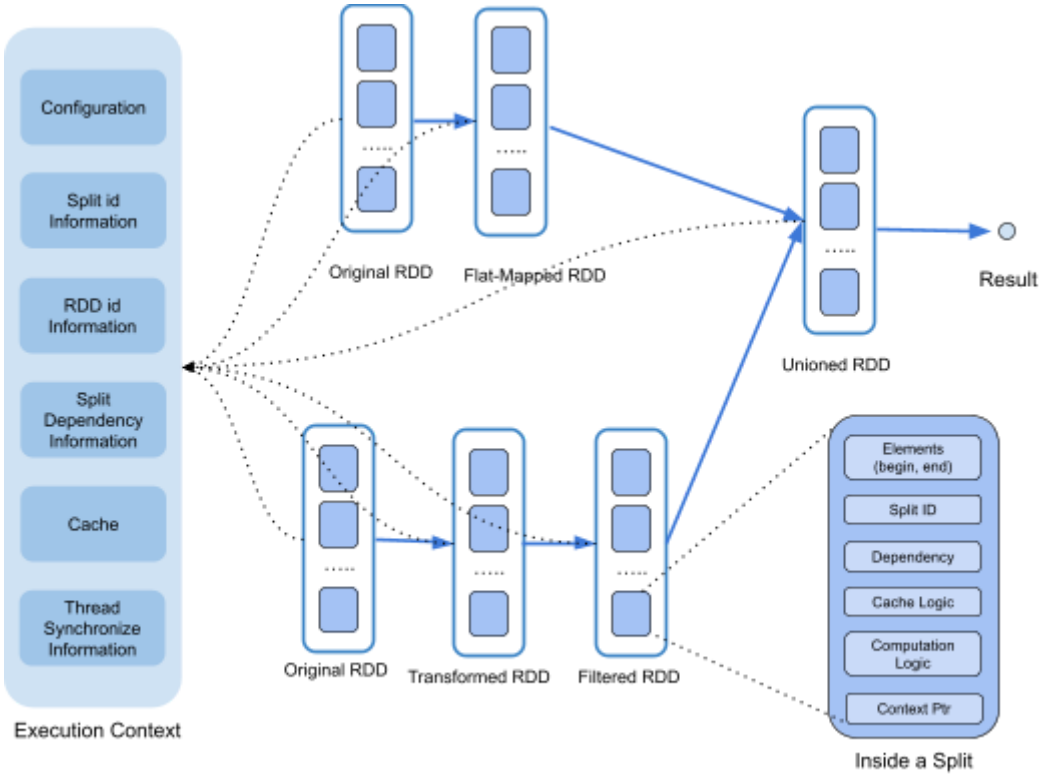


Figure: Overview of CPARK components implementation

4.2 Making Use of Iterators and Ranges

The implementation (and also the interface, of course) heavily relies on the iterators. This enables us (and the users) to better integrate it with the C++ standard library and conveniently make use of the helper functions in the standard ranges library.

The RDD's view interface is actually implemented as a pair of iterators that indicates its splits, and the splits's view interface is also implemented by a pair of iterators that indicates its elements.

The transformations are also implemented by iterators. For each transformation, we implemented our own special iterators that achieves this transformation logic.

The cache logic is also integrated in the iterators. We designed a special kind of iterator that can either compute values from the previous RDD, or read values from the cache. This makes the transformation logic very clean and reasonable.

4.3 Cache and Global Optimization

Because the CPARK library will start to execute the actual tasks after the whole computing graph is constructed, it will have a chance to analyze the global information and do some optimization based on that.

To decide how to cache the values of splits to avoid duplicate computation, we define two kinds of dependencies between the splits: the NARROW dependency and the WIDE dependency. The narrow dependent split means there is less than or equal to one of the other splits relying on this split. The wide dependent split means there are more than one splits depending on it.

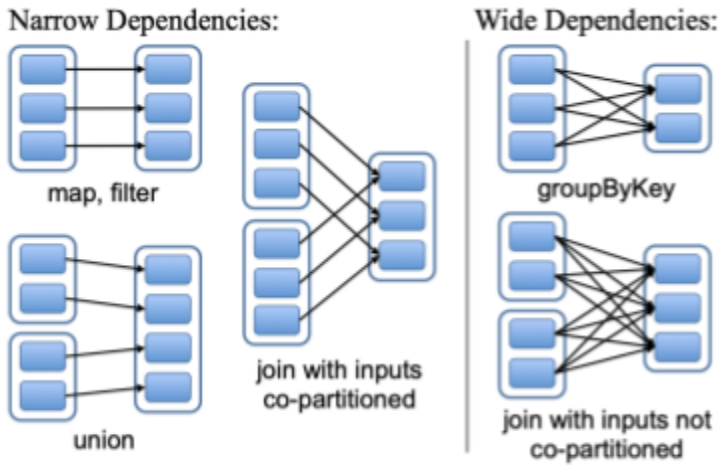


Figure: Examples of narrow and wide dependencies

Currently, if a split is a widely dependent split, we will cache its value in the execution context. The result of a chain of narrowly dependent splits will be calculated directly without storing any intermediate results. When the result of a widely dependent split is needed, the current thread will wait until the result of that split is computed and cached, and the following threads needing its value can also get it from the cache. In this case, we reduce duplicate computation and save CPU resources.

4.4 Inner Concurrency Model

Currently the CPARK library uses std::async to execute the actual tasks, and use the promise/future model

for inter-threads communication and synchronization. We would plan to add support for user customized thread pools in the future.

4.5 Achieving Zero-Overhead Abstraction

The CPARK library follows the zero-overhead abstraction principle in a best-effort manner. We have tried our best to reduce any extra runtime overhead in the implementation. The library uses complete compile-time polymorphism to avoid virtual function calls in the runtime; The argument for user customized functions (e.g. functions passed to filter or map) is generically typed (instead of using std::function) to achieve better inline-ness. The cache logic is well designed and implemented so that it will introduce no extra runtime cost for the tasks that do not use cache. Any unnecessary copy operation is carefully avoided. We believe the users will only pay what he or she used, and get almost the same performance as the same hand-written parallel program using low-level parallel computing facilities.