[Alink漫谈之三] AllReduce通信模型

目录

0x00 摘要

Alink 是阿里巴巴基于实时计算引擎 Flink 研发的新一代机器学习算法平台，是业界首个同时支持批式算法、流式算法的机器学习平台。本文将带领大家来分析Alink中通讯模型AllReduce的实现。AllReduce在Alink中应用较多，比如KMeans，LDA，Word2Vec，GD，lbfgs，Newton method，owlqn，SGD，Gbdt, random forest都用到了这个通讯模型。

因为Alink的公开资料太少，所以以下均为自行揣测，肯定会有疏漏错误，希望大家指出，我会随时更新。

0x01 MPI是什么

MPI(Message-Passing Interface)是一个跨语言的通讯协议，用于编写并行计算，支持点对点和广播。

MPI的目标是高性能、大规模性和可移植性。MPI在今天仍为高性能计算的主要模型。

其特点是

- A partitioned address space 每个线程只能通过调用api去读取非本地数据。所有的交互（Non-local Memory）都需要协同进行（握手）。
- Supports only explicit parallelization 只支持显性的并行化，用户必须明确的规定消息传递的方式。

AllReduce是MPI提供的一个基本原语，我们需要先了解reduce才能更好理解AllReduce。

- 规约函数 MPI_Reduce ： 规约是来自函数式编程的一个经典概念。其将通信子内各进程的同一个变量参与规约计算，并向指定的进程输出计算结果。比如通过一个函数将一批数据分成较小的一批数据。或者将一个数组的元素通过加法函数规约为一个数字。
- 规约并广播函数 MPI_Allreduce ： 在计算规约的基础上，将计算结果分发到每一个进程中。比如函数在得到归约结果值之后，将结果值分发给每一个进程，这样的话，并行中的所有进程值都能知道结果值了。

MPI_Allreduce和MPI_Reduce的一个区别就是，MPI_Reduce函数将最后的结果只传给了指定的dest_process号进程，而MPI_Allreduce函数可以将结果传递给所有的进程，因此所有的进程都能接收到结果。
MPI_Allreduce函数的原型也因此不需要指定目标进程号。

0x02 Alink 实现MPI的思想

AllReduce在Alink中应用较多，比如KMeans，LDA，Word2Vec，GD，lbfgs，Newton method，owlqn，SGD，Gbdt, random forest都用到了这个通讯模型。

AllReduce在算法实现中起到了承上启下的关键作用，即把原来串行跑的并行task强制打断，把计算结果进行汇总再分发，让串行继续执行。有一点类似大家熟悉的并发中的Barrier。

对比Flink原生KMeans算法，我们能看到AllReduce对应的是 `groupBy(0).reduce` 。只有所有数据都产生之后，才能做groupBy操作。

```
        DataSet<Centroid> newCentroids = points
                // compute closest centroid for each point
                .map(new SelectNearestCenter()).withBroadcastSet(loop, "centroids")
                // count and sum point coordinates for each centroid
                .map(new CountAppender())
        // 这里如果是Alink，就对应了AllReduce
                .groupBy(0).reduce(new CentroidAccumulator())
                // compute new centroids from point counts and coordinate sums
                .map(new CentroidAverager());
```

从AllReduce的注解中我们可以清晰的看出Alink实现MPI的思想。

```
 * An implement of {@link CommunicateFunction} that do the AllReduce.
 *
 * AllReduce is a communication primitive widely used in MPI. In our implementation, all worker
s do reduce on a partition of the whole data and they all get the final reduce result.
 *
 * There're mainly three stages:
 *   1. All workers send the there partial data to other workers for reduce.
 *   2. All workers do reduce on all data it received and then send partial results to others.
 *   3. All workers merge partial results into final result and put it into session context wit
h pre-defined object name.
 */
```

翻译如下：

```
所有的workers都在部分数据上做reduce操作，所有的workers都可以获取到reduce最终结果

主要有三个阶段：
1．所有workers给其他workers发送需要reduce的部分数据
2．所有workers在它收到的数据上做reduce，然后把这个部分reduce的结果发送给其他workers
3．所有workers把部分reduce的结果合并成为最终结果，然后放入预定义的session 上下文变量中
```

"纸上得来终觉浅，绝知此事要躬行。"

Alink为了实现AllReduce，在背后做了大量的工作，下面我们一一剖析。

0x03 如何实现共享

共享是实现AllReduce的第一要务，因为在归并/广播过程中需要元数据和输入输出，如果有共享变量就可以极大简化实现。我们下面就看看Alink如何通过task manager实现共享。

1. Task相关概念

- **Task**(任务)：Task 是一个阶段多个功能相同 subTask 的集合，类似于 Spark 中的 TaskSet。
- **subTask**(子任务)： subTask 是 Flink 中任务最小执行单元，是一个 Java 类的实例，这个 Java 类中有属性和方法，完成具体的计算逻辑。
- **链式优化**：按理说应该是每个算子的一个并行度实例就是一个subtask。那么，带来很多问题，由于flink的taskmanager运行task的时候是每个task采用一个单独的线程，这就会带来很多线程切换开销，进而影响吞吐量。为了减轻这种情况，flink进行了优化，也即对subtask进行链式操作，链式操作结束之后得到的task，再作为一个调度执行单元，放到一个线程里执行。
- **Operator Chains**(算子链)： Flink 将多个 subTask 合并成一个 Task(任务)，这个过程叫做 Operator Chains，每个任务由一个线程执行。使用 Operator Chains（算子链） 可以将多个分开的 subTask 拼接成一个任务。类似于 Spark 中的 Pipeline。
- **Slot**(插槽)： Flink 中计算资源进行隔离的单元，一个 Slot 中可以运行多个 subTask，但是这些 subTask 必须是来自同一个 application 的不同阶段的 subTask。结果就是，每个slot可以执行job的一整个pipeline。

Flink 中的程序本质上是并行的。在执行期间，每一个算子(Transformation)都有一个或多个算子 subTask（Operator SubTask），每个算子的 subTask 之间都是彼此独立，并在不同的线程中执行，并且可能在不同的机器或容器上执行。

同一个application，多个不同 task的 subTask，可以运行在同一个 slot 资源槽中。同一个 task 中的多个的 subTask，不能运行在一个 slot 资源槽中，他们可以分散到其他的资源槽中。对应到后面就是：AllReduceSend的多个并行度实例都不能运行在同一个slot中。

2. TaskManager

Flink 中每一个 TaskManager 都是一个JVM进程，它可能会在独立的线程上执行一个或多个 subtask。TaskManager 相当于整个集群的 Slave 节点，负责具体的任务执行和对应任务在每个节点上的资源申请和管理。

TaskManager为了对资源进行隔离和增加允许的task数，引入了slot的概念，这个slot对资源的隔离仅仅是对内存进行隔离，策略是均分。一个 TaskManager 至少有一个 slot。如果一个TM有N个Slot，则每个Slot分配到的Memory大小为整个TM Memory的1/N，同一个TM内的Slots只有Memory隔离，CPU是共享的。

客户端通过将编写好的 Flink 应用编译打包，提交到 JobManager，然后 JobManager 会根据已注册在 JobManager 中 TaskManager 的资源情况，将任务分配给有资源的 TaskManager节点，然后启动并运行任务。

TaskManager 从 JobManager 接收需要部署的任务，然后使用 Slot 资源启动 Task，建立数据接入的网络连接，接收数据并开始数据处理。同时 TaskManager 之间的数据交互都是通过数据流的方式进行的。

Flink 的任务运行其实是采用多线程的方式，一个TaskManager(TM)在多线程中并发执行多个task。这和 MapReduce 多 JVM 进行的方式有很大的区别，Flink 能够极大提高 CPU 使用效率，在多个任务和 Task 之间通过 TaskSlot 方式共享系统资源，每个 TaskManager 中通过管理多个 TaskSlot 资源池进行对资源进行有效管理。

对应到后面就是：在一个TaskManager中间运行的多个并行的AllReduceSend实例都会共享这个TaskManager中所有静态变量。

3. 状态共享

Alink就是利用task manager的静态变量实现了变量共享。其中有几个主要类和概念比较复杂。我们从上到下进行讲解，能看到随着从上到下，需要的标示和状态逐渐增加。

3.1 概念剖析

从上往下调用层次如下：

算法角度：ComContext

用户代码调用 ：context.getObj(bufferName); 这样对用户是最理想的，因为对于用户来说知道变量名字就可以经过上下文来存取。

但是ComContext则需要知道更多，比如还需要知道 自己对应的sessioin和taskID，具体下面会说明。

ComContext如此向下调用：SessionSharedObjs.put(objName, sessionId, taskId, obj);

框架角度：IterativeComQueue

IterativeComQueue 是一个框架概念。以Kmeans为例，就是Kmeans算法对应了若干IterativeComQueue。

IterativeComQueue上拥有众多compute/communicate function，每个function都应该知道自己属于哪一个IterativeComQueue，如何和本Queue上其他function进行通信，不能和其他Queue上搞混了。这样就需要有一个概念来表标示这个Queue。于是就有了下面Session概念。

Session角度：SessionSharedObjs

为了区分每个IterativeComQueue，就产生了session这个概念。这样IterativeComQueue上所有compute/communicate function都会绑定同一个session id，同一个IterativeComQueue上的所有function之间可以通信。

一个 IterativeComQueue 对应一个session，所以<"变量名" + sessionId>就对应了这个 session 能访问的某个变量。

SessionSharedObjs 包含静态成员变量：

- int sessionId = 0; 递增的标示，用来区分session。
- HashMap<Tuple2<String, Integer>, Long> key2Handle。映射，表示一个session中 某个变量名 对应某个变量handle。

正常来说 "某个名字的变量" 对应 "某个变量handle" 即可。即一个session中某个变量名 对应某个变量handle。但是Flink中，会有多个subtask并行操作的状态，这样就需要有一个新的概念来标示subtask对应的变量，这个变量应该和taskId有所关联。于是就有了下面的state概念。

SessionSharedObjs向下调用：IterTaskObjKeeper.put(handle, taskId, obj);

Subtask角度：IterTaskObjKeeper

这里就是用静态变量来实现共享。是task manager中所有的 tasks (threads)都可以访问的共享变量实例。

IterTaskObjKeeper 包含静态成员变量：

- long handle = 0L; 递增的标示，用来区分state。

- Map <Tuple2.of(handle, **taskId**), state> states; 是一个映射。即handle代表哪一种变量state，<handle, taskId>表示这种变量中 "哪个task" 对应的state实例，是针对subtask的一种细分。

在Flink中，一个算法会被多个subtask并行操作。如果只有一个handle，那么多个subtask共同访问，就会有大家都熟知的各种多线程操作问题。所以Alink这里将handle拆分为多个state。从subtask角度看，每个state用<handle, **taskId**>来唯一标示。

总结一下，就是对于同样一个变量名字，每个subtask对应的共享state其实都是独立的，大家互不干扰。共享其实就是在这个subtask上跑的各个operator之间共享。

3.2 变量实例分析

从实际执行的变量中，我们可以有一个更加清楚的认识。

```
// 能看出来 session 0 中，centroidAllReduce这个变量 对应的handle是 7
SessionSharedObjs.key2Handle = {HashMap@10480}  size = 9
 {Tuple2@10492} "(initCentroid,0)" -> {Long@10493} 1
 {Tuple2@10494} "(statistics,0)" -> {Long@10495} 2
 {Tuple2@10496} "(362158a2-588b-429f-b848-c901a1e15e17,0)" -> {Long@10497} 8
 {Tuple2@10498} "(k,0)" -> {Long@10499} 6
 {Tuple2@10500} "(centroidAllReduce,0)" -> {Long@10501} 7 // 这里就是所说的
 {Tuple2@10502} "(trainData,0)" -> {Long@10503} 0
 {Tuple2@10504} "(vectorSize,0)" -> {Long@10505} 3
 {Tuple2@10506} "(centroid2,0)" -> {Long@10507} 5
 {Tuple2@10508} "(centroid1,0)" -> {Long@10509} 4

// 下面能看出来，handle 7 这一种变量，因为有 4 个subtask，所以细分为4个state。
 com.alibaba.alink.common.comqueue.IterTaskObjKeeper.states = {HashMap@10520}  size = 36
 {Tuple2@10571} "(7,0)" -> {double[15]@10572}
 {Tuple2@10573} "(7,1)" -> {double[15]@10574}
 {Tuple2@10577} "(7,2)" -> {double[15]@10578}
 {Tuple2@10581} "(7,3)" -> {double[15]@10582}

 {Tuple2@10575} "(5,0)" -> {Tuple2@10576} "(10,com.alibaba.alink.operator.common.distance.FastD
istanceMatrixData@29a72fbb)"
 {Tuple2@10579} "(5,1)" -> {Tuple2@10580} "(10,com.alibaba.alink.operator.common.distance.FastD
istanceMatrixData@26c52354)"
 {Tuple2@10585} "(5,2)" -> {Tuple2@10586} "(10,com.alibaba.alink.operator.common.distance.FastD
istanceMatrixData@7c6ed779)"
 {Tuple2@10588} "(5,3)" -> {Tuple2@10589} "(10,com.alibaba.alink.operator.common.distance.FastD
istanceMatrixData@154b8a4d)"
```

下面让我们结合代码，一一解析涉及的类。

3.3 ComContext

ComContext 是最上层类，用来获取runtime信息和共享变量。IterativeComQueue（BaseComQueue ）上所有的compute/communicate function都通过 ComContext 来访问共享变量。比如：

```
public class BaseComQueue<Q extends BaseComQueue<Q>> implements Serializable {

    // 每一个BaseComQueue都会得到唯一一个sessionId。
    private final int sessionId = SessionSharedObjs.getNewSessionId();

    int taskId = getRuntimeContext().getIndexOfThisSubtask();

    public void mapPartition(Iterable<byte[]> values, Collector<byte[]> out) {
        // 获取到了一个ComContext
        ComContext context = new ComContext(sessionId, getIterationRuntimeContext());
        if (getIterationRuntimeContext().getSuperstepNumber() == maxIter || criterion) {
```

```
            // 利用ComContext继续访问共享变量
            List<Row> model = completeResult.calc(context);
        }
    }
}

// 用户类似这么调用

double[] sendBuf = context.getObj(bufferName);
```

可以看出来，ComContext 就是用户应该看到的最顶层上下文概念。 taskId, sessionId 是使用关键。

- sessionId 是在 SessionSharedObjs中定义的静态类成员变量，其会自动递增。每一个BaseComQueue都会得到唯一一个sessionId，即该Queue保持了唯一session。这样BaseComQueue中生成的ComContext都有相同的sessionId。
- taskId是从runtime中获得。

```
/**
 * Encapsulates task-specific information: name, index of subtask, parallelism and attempt numb
er.
 */
@Internal
public class TaskInfo {
        /**
         * Gets the number of this parallel subtask. The numbering starts from 0 and goes up to
 parallelism-1 (parallelism as returned by {@link #getNumberOfParallelSubtasks()}).
         *
         * @return The index of the parallel subtask.
         */
        public int getIndexOfThisSubtask() {
                return this.indexOfSubtask; // 这里获取taskId
        }
}
```

ComContext 具体类定义如下

```
/**
 * Context used in BaseComQueue to access basic runtime information and shared objects.
 */
public class ComContext {
        private final int taskId;
        private final int numTask;
        private final int stepNo;
        private final int sessionId;

        public ComContext(int sessionId, IterationRuntimeContext runtimeContext) {
                this.sessionId = sessionId;
                this.numTask = runtimeContext.getNumberOfParallelSubtasks();
                this.taskId = runtimeContext.getIndexOfThisSubtask();
                this.stepNo = runtimeContext.getSuperstepNumber();
        }

        /**
         * Put an object into shared objects for access of other QueueItem of the same taskId.
         *
         * @param objName object name
         * @param obj     object itself.
         */
        public void putObj(String objName, Object obj) {
```

```
                SessionSharedObjs.put(objName, sessionId, taskId, obj);
        }
}

// 比如具体举例如下
this = {ComContext@10578}
 taskId = 4
 numTask = 8
 stepNo = 1
 sessionId = 0
```

3.4 SessionSharedObjs

SessionSharedObjs是再下一层的类，维护shared session objects， 这个session 共享是通过 sessionId 做到的。

SessionSharedObjs 维护了一个静态类变量 sessionId，由此区分各个Session。

SessionSharedObjs核心是 `HashMap<Tuple2<String, Integer>, Long> key2Handle` 。即 <"变量名" + sessionId> ---> <真实变量 handle> 的一个映射。

一个 IterativeComQueue 对应一个session，所以<"变量名" + sessionId>就对应了这个 IterativeComQueue 能访问的某个变量，正常来说有一个变量handle即可。

但是因为一个 IterativeComQueue会被若干subtask并行执行，所以为了互斥和区分，所以每个handle又细分为若干state，每个state用<handle, taskId>来唯一标示。在下面会提到。

```
/**
 * An static class that manage shared objects for {@link BaseComQueue}s.
 */
class SessionSharedObjs implements Serializable {
        private static HashMap<Tuple2<String, Integer>, Long> key2Handle = new HashMap<>();
        private static int sessionId = 0;
        private static ReadWriteLock rwlock = new ReentrantReadWriteLock();

        /**
         * Get a new session id.
         * All access operation should bind with a session id. This id is usually shared among
compute/communicate function of an {@link IterativeComQueue}.
         *
         * @return new session id.
         */
        synchronized static int getNewSessionId() {
                return sessionId++;
        }

        static void put(String objName, int session, int taskId, Object obj) {
                rwlock.writeLock().lock();
                try {
                        Long handle = key2Handle.get(Tuple2.of(objName, session));
                        if (handle == null) {
                                handle = IterTaskObjKeeper.getNewHandle();
                                key2Handle.put(Tuple2.of(objName, session), handle);
                        }
      // 这里进行调用。taskId也是辨识关键。
                        IterTaskObjKeeper.put(handle, taskId, obj);
                } finally {
                        rwlock.writeLock().unlock();
                }
```

```
        }
}
```

3.5 IterTaskObjKeeper

这是最底层的共享类，是在task manager进程的堆内存上的一个静态实例。task manager的所有task (threads) 都可以分享。

看源码可知，IterTaskObjKeeper 是通过一个静态变量states实现了在整个JVM内共享。而具体内容是由 'handle' and 'taskId' 来共同决定。

IterTaskObjKeeper维持了 handle 递增来作为 “变量state” 的唯一种类标识。

用<handle, taskId>来作为“变量state”的唯一标识。这个就是在 task manager process 堆内存中被大家共享的变量。

即handle代表哪一种变量state，<handle, taskId>表示这种变量中，对应哪一个task的哪一个变量。 这是针对task的一种细分。

```
/**
 * A 'state' is an object in the heap memory of task manager process,
 * shared across all tasks (threads) in the task manager.

 * Note that the 'state' is shared by all tasks on the same task manager,
 * users should guarantee that no two tasks modify a 'state' at the same time.

 * A 'state' is identified by 'handle' and 'taskId'.
 */
public class IterTaskObjKeeper implements Serializable {
        private static Map <Tuple2 <Long, Integer>, Object> states;

        /**
         * A 'handle' is a unique identifier of a state.
         */
        private static long handle = 0L;

        private static ReadWriteLock rwlock = new ReentrantReadWriteLock();

        static {
                states = new HashMap <>();
        }

        /**
         * @note Should get a new handle on the client side and pass it to transformers.
         */
        synchronized public static long getNewHandle() {
                return handle++;
        }

        public static void put(long handle, int taskId, Object state) {
                rwlock.writeLock().lock();
                try {
                        states.put(Tuple2.of(handle, taskId), state);
                } finally {
                        rwlock.writeLock().unlock();
                }
        }
}
```

0x04. 示例代码

我们示例代码依然如下。

KMeansTrainBatchOp调用

```
        static DataSet <Row> iterateICQ(...省略...) {
                return new IterativeComQueue()
                        .initWithPartitionedData(TRAIN_DATA, data)
                        .initWithBroadcastData(INIT_CENTROID, initCentroid)
                        .initWithBroadcastData(KMEANS_STATISTICS, statistics)
                        .add(new KMeansPreallocateCentroid())
                        .add(new KMeansAssignCluster(distance))
                        .add(new AllReduce(CENTROID_ALL_REDUCE))
                        .add(new KMeansUpdateCentroids(distance))
                        .setCompareCriterionOfNode0(new KMeansIterTermination(distance, tol))
                        .closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeC
olName, longitudeColName))
                        .setMaxIter(maxIter)
                        .exec();
        }
```

AllReduce实现

Alink的AllReduce主要代码摘取如下：

```
public static <T> DataSet <T> allReduce(
    return input
                .mapPartition(new AllReduceSend <T>(bufferName, lengthName, transferBufferName,
 sessionId))
                .withBroadcastSet(input, "barrier")
                .returns(
                        new TupleTypeInfo <>(Types.INT, Types.INT, PrimitiveArrayTypeInfo.DOUBL
E_PRIMITIVE_ARRAY_TYPE_INFO))
                .name("AllReduceSend")
                .partitionCustom(new Partitioner <Integer>() {
                        @Override
                        public int partition(Integer key, int numPartitions) {
                                return key;
                        }
                }, 0)
                .name("AllReduceBroadcastRaw")
                .mapPartition(new AllReduceSum(bufferName, lengthName, sessionId, op))
                .returns(
                        new TupleTypeInfo <>(Types.INT, Types.INT, PrimitiveArrayTypeInfo.DOUBL
E_PRIMITIVE_ARRAY_TYPE_INFO))
                .name("AllReduceSum")
                .partitionCustom(new Partitioner <Integer>() {
                        @Override
                        public int partition(Integer key, int numPartitions) {
                                return key;
                        }
                }, 0)
                .name("AllReduceBroadcastSum")
                .mapPartition(new AllReduceRecv <T>(bufferName, lengthName, sessionId))
                .returns(input.getType())
                .name("AllReduceRecv");
}
```

0x05 AllReduce实现

结合上面具体代码，我们先总结AllReduce使用流程如下

- KMeansAssignCluster ： Find the closest cluster for every point and calculate the sums of the points belonging to the same cluster。然后把自己计算出来的cluster 写入到自己 task manager 的 CENTROID_ALL_REDUCE。
- 每个AllReduceSend 从自己task manager的CENTROID_ALL_REDUCE中取出之前存入的 cluster（每个AllReduceSend获取的cluster都是只有自己能看到的），然后发送给下游task。发送时根据 "下游task index 和 数据量" 来决定往哪些task发送。这里要注意的是：具体给哪一个task发送变量的哪一部分，是依据那个task 的 task index 和数据量 来计算出来的。这个计算机制（如何计算在代码中，也有部分作为元信息随着数据一起发送）被后面的AllReduceRecv复用。
- 每个 AllReduceSum 接收到 AllReduceSend 发送过来的 cluster，计算求和，然后把计算结果再发送出去。每一个AllReduceSum 都是把自己计算求和出来的数据统一发给每一个下游task。
- 每个 AllReduceRecv 都接收到 所有 AllReduceSum 发送过来的（求和之后的）cluster。存入到共享变量 CENTROID_ALL_REDUCE。具体如何存就复用AllReduceSend计算机制，这样存到共享变量的什么地方就互相不会冲突。可以理解为merge操作：比如有5个AllReduce，每个AllReduce的数据都发给了5个AllReduceRecv，每个AllReduceRecv接到这5份数据之后，会根据自己的subtask index写到自己对应的state中，但是这5份数据分别写在state什么地方都是在数据元信息中指定的，彼此不会有写的冲突，这样每个AllReduceRecv就拥有了全部5份数据。
- KMeansUpdateCentroids ： 取出CENTROID_ALL_REDUCE变量，然后Update the centroids based on the sum of points and point number belonging to the same cluster

1. KMeansAssignCluster

该类的作用是：为每个点(point)计算最近的聚类中心，为每个聚类中心的点坐标的计数和求和。

我们可以看出，KMeansAssignCluster 通过ComContext存储了CENTROID_ALL_REDUCE，为后续AllReduce使用。假如有5个KMeansAssignCluster，则他们计算出来的结果一般来说各不相同。虽然存储同一个变量名CENTROID_ALL_REDUCE，但是其state各不相同。

因为这5个KMeansAssignCluster势必对应了5个subtask，则其在共享变量中的<handle, taskId>必不相同，则对应不同的state，所以分开存储。

```
// Find the closest cluster for every point and calculate the sums of the points belonging to t
he same cluster.
public class KMeansAssignCluster extends ComputeFunction {
        // 存取共享变量
        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);
        if (sumMatrixData == null) {
            sumMatrixData = new double[k * (vectorSize + 1)];
            context.putObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE, sumMatrixData);
        }

        for (FastDistanceVectorData sample : trainData) {
            // Find the closest centroid from centroids for sample, and add the sample to sumMa
trix.
            KMeansUtil.updateSumMatrix(sample, 1, stepNumCentroids.f1, vectorSize, sumMatrixDat
a, k, fastDistance, distanceMatrix);
        }
}

// 程序中各个变量如下

sample = {FastDistanceVectorData@13274}
 vector = {DenseVector@13281} "6.3 2.5 4.9 1.5"
```

```
 label = {DenseVector@13282} "72.2"
 rows = {Row[1]@13283}

// 这个就是共享变量。4维向量 + 1 weight ---> 都是"sample和"。
sumMatrixData = {double[15]@10574}
 0 = 23.6
 1 = 14.9
 2 = 8.7
 3 = 1.7000000000000002
 4 = 5.0
 5 = 52.400000000000006
 6 = 25.1
 7 = 39.699999999999996
 8 = 13.299999999999999
 9 = 9.0
 10 = 33.0
 11 = 16.9
 12 = 28.900000000000002
 13 = 11.4
 14 = 5.0

trainData = {ArrayList@10580}  size = 19
 0 = {FastDistanceVectorData@10590}
  vector = {DenseVector@10595} "7.7 3.8 6.7 2.2"
   data = {double[4]@10601}
    0 = 7.7
    1 = 3.8
    2 = 6.7
    3 = 2.2
  label = {DenseVector@10596} "123.46000000000001"
  rows = {Row[1]@10597}
 1 = {FastDistanceVectorData@10603}
  vector = {DenseVector@10623} "5.7 2.8 4.1 1.3"
  label = {DenseVector@10624} "58.83"
  rows = {Row[1]@10625}
 2 = {FastDistanceVectorData@10604}
 3 = {FastDistanceVectorData@10605}
......
 17 = {FastDistanceVectorData@10619}
 18 = {FastDistanceVectorData@10620}
  vector = {DenseVector@10654} "6.5 3.0 5.2 2.0"
  label = {DenseVector@10655} "82.29"
  rows = {Row[1]@10656}
```

2. AllReduceSend

这里需要再把代码摘录一遍，主要是因为有withBroadcastSet。其作用是：

- 可以理解为是一个公共的共享变量，我们可以把一个dataset 数据集广播出去，然后不同的task在节点上都能够获取到，这个数据在每个节点上只会存在一份。
- 如果不使用broadcast，则在每个节点中的每个task中都需要拷贝一份dataset数据集，比较浪费内存(也就是一个节点中可能会存在多份dataset数据)。

```
                return input
                        .mapPartition(new AllReduceSend <T>(bufferName, lengthName, transferBuf
ferName, sessionId))
                        .withBroadcastSet(input, "barrier")
```

KMeansAssignCluster 会往上下文的变量centroidAllReduce中添加数据。所以 AllReduce 其实就是在等待这个变量。

AllReduce的第一步就是从上下文中取出共享变量，然后发送。这部分代码由AllReduceSend完成。

对于AllReduceSend的每个task来说，bufferName都是 centroidAllReduce。

因为每个AllReduceSend也对应不同的task，所以每个AllReduceSend读取的centroidAllReduce必然不一样，所以每个task获取的sendBuf都不一样。他们分别把自己<handle, taskId>对应的 "centroidAllReduce" state取出，发送给下游。

AllReduceSend 发给其下游时候，是以subtask的序号为基准发送给每一个task，即本task中获取的共享变量会发送给每一个task，但是具体给哪一个task发送变量的那一部分，是依据那个task 的 task index 和数据量 来计算出来的。如果数据量少，可能只给某一个或者几个task发送。

后续中的 taskId ，都是subtask id。

其中，如何计算给哪个task发送多少，是在DefaultDistributedInfo完成的。这里需要结合 pieces 函数进行分析。需要注意的是：AllReduceSend这么发送，AllReduceRecv后面也按照这个套路接受。这样AllReduceRecv就可以merge了。

```
AllReduceSend这么发送，AllReduceRecv后面也按照这个套路接受

int pieces = pieces(sendLen);//表示本人这次send的数据分成几片，比如分成50片。每片大小是TRANSFER_BUFFER_
SIZE

// 将要发给 8 个 subtask
for (int i = 0; i < numOfSubTasks; ++i) {
      // 假如第5个subtask，那么它发送的起始位置就是50/8 * 4
      int startPos = (int) distributedInfo.startPos(i, numOfSubTasks, pieces);
      // 给第5个subtask发送多少片
      int cnt = (int) distributedInfo.localRowCnt(i, numOfSubTasks, pieces);
```

具体代码如下：

```
        private static int pieces(int len) {
                int div = len / TRANSFER_BUFFER_SIZE; //本人这次send的数据分成几片，每片大小是TRANSFE
R_BUFFER_SIZE
                int mod = len % TRANSFER_BUFFER_SIZE;

                return mod == 0 ? div : div + 1;
        }

public class DefaultDistributedInfo implements DistributedInfo {

        public long startPos(long taskId, long parallelism, long globalRowCnt) {
                long div = globalRowCnt / parallelism;
                long mod = globalRowCnt % parallelism;

                if (mod == 0) {
                        return div * taskId;
                } else if (taskId >= mod) {
                        return div * taskId + mod;
                } else {
                        return div * taskId + taskId;
                }
        }

        public long localRowCnt(long taskId, long parallelism, long globalRowCnt) {
```

```
                long div = globalRowCnt / parallelism;
                long mod = globalRowCnt % parallelism;

                if (mod == 0) {
                        return div;
                } else if (taskId >= mod) {
                        return div;
                } else {
                        return div + 1;
                }
        }
}
```

具体AllReduceSend代码如下，注解中有详细说明。

```
// 这里是变量名字定义。
public static final String CENTROID_ALL_REDUCE = "centroidAllReduce";

private static class AllReduceSend<T> extends RichMapPartitionFunction <T, Tuple3 <Integer, Int
eger, double[]>> {

        int numOfSubTasks = getRuntimeContext().getNumberOfParallelSubtasks();
                // 与并行度相关，每个task都会执行相同操作
                // bufferName都是 centroidAllReduce，每个task获取的sendBuf都不一样

        // 计算怎么发送所需要的数据结构
        int pieces = pieces(sendLen);
        DistributedInfo distributedInfo = new DefaultDistributedInfo();

        // 从上下文中获取需要传送的数据
                double[] sendBuf = context.getObj(bufferName);

                        int agg = 0;
                // 可以看出来，是把需要传送的数据给每个task都发送。当然这个发送是根据发送数据的大小来确定的，
如果数据量小，可能就只给一个或者几个task发送。
                        for (int i = 0; i < numOfSubTasks; ++i) {
                // startPos : 具体发送变量的那一部分，是依据task index来决定的。
                // cnt : 具体哪一个下游 task i 发送多少数据由此决定，如果是0，就不给task i发送数据。
                                int startPos = (int) distributedInfo.startPos(i, numOfSubTasks,
 pieces);
                                int cnt = (int) distributedInfo.localRowCnt(i, numOfSubTasks, p
ieces);

                                for (int j = 0; j < cnt; ++j) {
                    // 发送哪一个部分
                                        int bufStart = (startPos + j) * TRANSFER_BUFFER_SIZE;
                                        // the last
                                        if (startPos + j == pieces - 1) {
                                                System.arraycopy(sendBuf, bufStart, transBuf, 0
, lastLen(sendLen));
                                        } else {
                                                System.arraycopy(sendBuf, bufStart, transBuf, 0
, TRANSFER_BUFFER_SIZE);
                                        }
                                        agg++;

          // i 是subTasks的index，startPos + j是buffer内的位置，后续分区实际就是按照这个 i 来分区的。本A
llReduceSend就是发送到numOfSubTasks这些task中。
                                        out.collect(Tuple3.of(i, startPos + j, transBuf));
                                }
```

```
                        }
}

        private static int pieces(int len) {
                int div = len / TRANSFER_BUFFER_SIZE; // 4096
                int mod = len % TRANSFER_BUFFER_SIZE;
                return mod == 0 ? div : div + 1;
        }

sendBuf = {double[15]@10602}
 0 = 40.3
 1 = 18.200000000000003
 2 = 33.6
 3 = 12.5
 4 = 6.0
 5 = 45.3
 6 = 30.599999999999998
 7 = 12.4
 8 = 2.0
 9 = 9.0
 10 = 24.0
 11 = 10.4
 12 = 17.1
 13 = 5.199999999999999
 14 = 4.0

this = {AllReduce$AllReduceSend@10598}
 bufferName = "centroidAllReduce"
 lengthName = null
 transferBufferName = "3dfb2aae-683d-4497-91fc-30b8d6853bce"
 sessionId = 0
 runtimeContext = {AbstractIterativeTask$IterativeRuntimeUdfContext@10606}
```

3. AllReduceBroadcastRaw

AllReduceSend发送变量给下游时候，使用了自定义的partition（partitionCustom ）。其是用 index of subtask 来作为key分区。这样就和AllReduceSend那个out.collect对应了。

```
                        .partitionCustom(new Partitioner <Integer>() {
                                @Override
                                public int partition(Integer key, int numPartitions) {
                                        return key;
                                }
                        }, 0)
                        .name("AllReduceBroadcastRaw")

// 调用到这个partition函数的调用栈

partition:102, AllReduce$2 (com.alibaba.alink.common.comqueue.communication)
partition:99, AllReduce$2 (com.alibaba.alink.common.comqueue.communication)
customPartition:235, OutputEmitter (org.apache.flink.runtime.operators.shipping)
selectChannel:149, OutputEmitter (org.apache.flink.runtime.operators.shipping)
selectChannel:36, OutputEmitter (org.apache.flink.runtime.operators.shipping)
emit:120, RecordWriter (org.apache.flink.runtime.io.network.api.writer)
collect:65, OutputCollector (org.apache.flink.runtime.operators.shipping)
collect:35, CountingCollector (org.apache.flink.runtime.operators.util.metrics)
mapPartition:257, AllReduce$AllReduceSend (com.alibaba.alink.common.comqueue.communication)
run:103, MapPartitionDriver (org.apache.flink.runtime.operators)
run:504, BatchTask (org.apache.flink.runtime.operators)
```

```
run:157, AbstractIterativeTask (org.apache.flink.runtime.iterative.task)
run:107, IterationIntermediateTask (org.apache.flink.runtime.iterative.task)
invoke:369, BatchTask (org.apache.flink.runtime.operators)
doRun:705, Task (org.apache.flink.runtime.taskmanager)
run:530, Task (org.apache.flink.runtime.taskmanager)
run:745, Thread (java.lang)


 // @AllReduceSend.mapPartition 这里开始调用
 for (int i = 0; i < numOfSubTasks; ++i) {
     // i 是subTasks的index，后续分区实际就是按照这个 i 来分区的。本AllReduceSend就是发送到numOfSubTasks这些task中。
         out.collect(Tuple3.of(i, startPos + j, transBuf));
 }

 // 从后续调用序列可以看出来，最终是用 index of subtask 来作为key分区。

// 这里发送record

 public class CountingCollector<OUT> implements Collector<OUT> {
        public void collect(OUT record) {
                this.numRecordsOut.inc();
                this.collector.collect(record);
        }
 }

 record = {Tuple3@10586} "(0,0,[40.50000000000001, 18.7, 33.300000000000004, 12.8, 6.0, 29.7, 21.0, 8.4, 1.7, 6.0, 48.1, 22.199999999999996, 36.0, 12.200000000000001, 8.0, 0.0,"
 f0 = {Integer@10583} 0
 f1 = {Integer@10583} 0
 f2 = {double[4096]@10598}

// 这里开始分区

public class OutputEmitter<T> implements ChannelSelector<SerializationDelegate<T>> {
        private int customPartition(T record, int numberOfChannels) {
                if (extractedKeys == null) {
                        extractedKeys = new Object[1];
                }

                if (comparator.extractKeys(record, extractedKeys, 0) == 1) {
            // 所以 key 是 0
                        final Object key = extractedKeys[0];
                        return partitioner.partition(key, numberOfChannels);
                }
        }
}

public final class TupleComparator<T extends Tuple> extends TupleComparatorBase<T> {
        public int extractKeys(Object record, Object[] target, int index) {
                int localIndex = index;
                for(int i = 0; i < comparators.length; i++) {
                        localIndex += comparators[i].extractKeys(((Tuple) record).getField(keyPositions[i]), target, localIndex);
                }
                return localIndex - index;
        }
}

// 就是取出第一个field的数值
```

```
key = {Integer@10583} 0
 value = 0

extractedKeys = {Object[1]@10587}
 0 = {Integer@10583} 0
  value = 0
```

4. AllReduceSum

所有workers在它收到的数据上做reduce，然后把这个部分reduce的结果（partial results）发送给其他workers。

partial results是因为每个task接受的数据不同，是上游根据task index计算位置并且发送过来的。

但是AllReduceSum的计算结果会给每一个下游 task index 发送。

```
private static class AllReduceSum extends RichMapPartitionFunction <Tuple3 <Integer, Integer, d
ouble[]>, Tuple3 <Integer, Integer, double[]>> {

        public void mapPartition(Iterable <Tuple3 <Integer, Integer, double[]>> values,Collecto
r <Tuple3 <Integer, Integer, double[]>> out) {

            // 这时候虽然也用到了context取出了sendBuf，但是只是用来获取其长度而已。
                int taskId = getRuntimeContext().getIndexOfThisSubtask();
                        int numOfSubTasks = getRuntimeContext().getNumberOfParallelSubtasks();

                        double[] sendBuf = context.getObj(bufferName);
                        int sendLen = lengthName != null ? context.getObj(lengthName) : sendBuf
.length;
                        int pieces = pieces(sendLen);
                        DistributedInfo distributedInfo = new DefaultDistributedInfo();

            // startPos : 本task接受的数据，startPos 是应该从原始数据的哪个位置开始。是依据task index来
决定的。
            // cnt : 具体哪一个下游 task i 发送多少数据由此决定。
                        int startPos = (int) distributedInfo.startPos(taskId, numOfSubTasks, pi
eces);
                        int cnt = (int) distributedInfo.localRowCnt(taskId, numOfSubTasks, piec
es);

                // 这里进行了reduce SUM工作
                        double[][] sum = new double[cnt][];
                        double[] agg = new double[cnt];
                        do {
                                Tuple3 <Integer, Integer, double[]> val = it.next();
                                int localPos = val.f1 - startPos;
                                if (sum[localPos] == null) {
                                        sum[localPos] = val.f2;
                                        agg[localPos]++;
                                } else {
                                        op.accept(sum[localPos], val.f2);
                                }
                        } while (it.hasNext());

                // 依然发送给下游，依然是用subtask index来作为partition key。
            // 注意，这里是把结果发送给所有的下游task。
                        for (int i = 0; i < numOfSubTasks; ++i) {
                                for (int j = 0; j < cnt; ++j) {
          // startPos是本task发送的数据应该从原始数据的哪个位置开始。
```

```
            // 但是给每一个 task i 发的都是同样的数据。但是 startPos + j 很重要，下游task i 会根据这个知道它应该把接收到的数据存储在预定义变量的什么地方。
                                        out.collect(Tuple3.of(i, startPos + j, sum[j]));
                                    }
                                }
        }
}

sum = {double[1][]@10605}
 0 = {double[4096]@10613}
  0 = 118.50000000000001
  1 = 77.7
  2 = 37.2
  3 = 5.9
  4 = 25.0
  5 = 621.1000000000001
  6 = 284.7
  7 = 487.59999999999997
  8 = 166.5
  9 = 99.0
  10 = 136.9
  11 = 95.7
  12 = 39.0
  13 = 7.4
  14 = 26.0
```

5. AllReduceBroadcastSum

AllReduceSum 发送变量给下游时候，使用了自定义的partition（partitionCustom ）。其是用 index of subtask 来作为key分区。

其意义和之前的 partitionCustom 相同。

6. AllReduceRecv

All workers merge partial results into final result and put it into session context with pre-defined object name.

每一个下游 AllReduceRecv 都接收到 **每一个**上游 AllReduceSum 发送过来的 cluster（求和之后的），然后把每份数据存入到自己task manager对应的预定义变量state的不同部分（这个不同部分是根据接受到的数据val.f1计算出来的）。

结合前面可知，AllReduceSend发送和AllReduceRecv接受，都是按照同样的套路计算在共享变量中的数据位置。这样AllReduceRecv就可以merge了。

这样就完成了所有workers把部分reduce sum的结果合并成为最终结果，然后放入预定义的上下文变量中。

```
        private static class AllReduceRecv<T> extends RichMapPartitionFunction <Tuple3 <Integer, Integer, double[]>, T> {
                private final String bufferName;
                private final String lengthName;
                private final int sessionId;

                @Override
                public void mapPartition(Iterable <Tuple3 <Integer, Integer, double[]>> values, Collector <T> out) throws Exception {
                        ComContext context = new ComContext(sessionId, getIterationRuntimeContext());

                        Iterator <Tuple3 <Integer, Integer, double[]>> it = values.iterator();
                        if (!it.hasNext()) {
```

```
                                return;
                        }
                        double[] recvBuf = context.getObj(bufferName);
                        int recvLen = lengthName != null ? context.getObj(lengthName) : recvBuf
.length;
                        int pieces = pieces(recvLen); // 和之前AllReduceSend一样的套路计算应该存储在
共享变量什么位置。
                        do {
                                Tuple3 <Integer, Integer, double[]> val = it.next();
                                if (val.f1 == pieces - 1) {
                                        System.arraycopy(val.f2, 0, recvBuf, val.f1 * TRANSFER_
BUFFER_SIZE, lastLen(recvLen));
                                } else {
            // 拷贝到共享变量的相应部位。val.f1 是上游发送过来的。作为merge功能的起始位置。
                                        System.arraycopy(val.f2, 0, recvBuf, val.f1 * TRANSFER_
BUFFER_SIZE, TRANSFER_BUFFER_SIZE);
                                }
                        } while (it.hasNext());
                }
        }

val = {Tuple3@10672} "(3,0,[335.3, 150.89999999999998, 277.5, 99.79999999999998, 50.0, 290.9, 1
36.3, 213.1, 67.8, 50.0, 250.3, 170.89999999999998, 73.2, 12.2, 50.0, 0.0....."
 f0 = {Integer@10682} 3
  value = 3
 f1 = {Integer@10638} 0
  value = 0
 f2 = {double[4096]@10674}
  0 = 335.3
  1 = 150.89999999999998
  2 = 277.5
  3 = 99.79999999999998
  4 = 50.0
  5 = 290.9
  6 = 136.3
  7 = 213.1
  8 = 67.8
  9 = 50.0
  10 = 250.3
  11 = 170.89999999999998
  12 = 73.2
  13 = 12.2
  14 = 50.0
  15 = 0.0
  ......

// 每个task都收到了reduce sum结果。
recvBuf = {double[15]@10666}
 0 = 404.3
 1 = 183.1
 2 = 329.3
 3 = 117.2
 4 = 61.0
 5 = 250.3
 6 = 170.89999999999998
 7 = 73.20000000000002
 8 = 12.2
 9 = 50.0
 10 = 221.89999999999998
 11 = 104.1
```

```
 12 = 161.29999999999998
 13 = 50.4
 14 = 39.0
```

7. KMeansUpdateCentroids

基于点计数和坐标，计算新的聚类中心。这里就是从task manager中取出了AllReduce存储的共享变量CENTROID_ALL_REDUCE。

```
/**
 * Update the centroids based on the sum of points and point number belonging to the same clust
er.
 */
public class KMeansUpdateCentroids extends ComputeFunction {
    public void calc(ComContext context) {

        Integer vectorSize = context.getObj(KMeansTrainBatchOp.VECTOR_SIZE);
        Integer k = context.getObj(KMeansTrainBatchOp.K);

        // 这里取出AllReduce存储的共享变量
        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);

        Tuple2<Integer, FastDistanceMatrixData> stepNumCentroids;
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        }

        stepNumCentroids.f0 = context.getStepNo();

        context.putObj(KMeansTrainBatchOp.K,
            updateCentroids(stepNumCentroids.f1, k, vectorSize, sumMatrixData, distance));
    }
}
```

0x06 参考

[我的并行计算之路（四）MPI集合通信之Reduce和Allreduce]

[Message Passing Interface(MPI)]

[Flink 之 Dataflow、Task、subTask、Operator Chains、Slot 介绍]

[Flink运行时之TaskManager执行Task]