Design doc: BarrierTaskContext.barrier()

Author: Xingbo Jiang

History:
2018-07-23: First draft
2018-08-29: Updated

Overview

Barrier execution mode SPIP:
https://docs.google.com/document/d/1JR6lWcgAI53lCUxy4qvQSv8w1jXZrbS7DC9AjYlwqJE/edit

We need to provide a communication barrier function to users to help coordinate tasks within a barrier stage(Barrier Execution Mode). Similar to MPI_Barrier function in MPI, the barrier() function call blocks until all the tasks in the same stage have reached this routine.

Requirements

Goals

- Low-latency. The tasks should be unblocked immediately after all tasks have reached this barrier. The wall-clock latency is more important than CPU cycles here.
- Support unlimited timeout with proper logging. For deep learning tasks, it might take very long to converge. We should support unlimited timeout with proper logging, so users know why a task is waiting.

Non-Goals

Interfaces

Proposed API

```
/** A [[TaskContext]] with extra info and tooling for a barrier stage. */
class BarrierTaskContext extends TaskContext {

  /**
   * Sets a global barrier and waits until all tasks in this stage hit this barrier.
   *
   * This function is expected to be called by EVERY tasks in the same barrier stage in the
   * SAME pattern, otherwise you may get a SparkException. Some examples of misuses listed
   * below:
   * 1. Only call barrier() function on a subset of all the tasks in the same barrier stage,
   * it shall lead to time out of the function call.
   * rdd.barrier().mapPartitions { iter =>
   *     val context = BarrierTaskContext.get()
   *     if (context.partitionId() == 0) {
   *         // Do nothing.
   *     } else {
   *         context.barrier()
   *     }
   *     iter
   * }
   *
   * 2. Include barrier() function in a try-catch code block, this may lead to time out of the
   * function call.
   * rdd.barrier().mapPartitions { iter =>
   *     val context = BarrierTaskContext.get()
   *     try {
   *         // Do something that might throw an Exception.
   *         doSomething()
   *         context.barrier()
```

```
   *     } catch {
   *         case e: Exception => logWarning("...", e)
   *     }
   *     context.barrier()
   *     iter
   * }
   */
  def barrier(): Unit = ...
}
```

Example Code

This is a code snippet to embed an MPI program in a barrier stage.

```
 rdd.barrier().mapPartitions { iter =>
    // Write iter to disk.
    ???

    // Fetch TaskContext
    val context = BarrierTaskContext.get()

    // Wait until all tasks finished writing.
    context.barrier()

    // The 0-th task launches a MPI job.
    if (context.partitionId() == 0) {
      val hosts = context.getTaskInfos().map(_.host)
      // Set up MPI machine file using host infos.
      ???

      // Launch the MPI job by calling mpirun.
      ???
    }

    // Wait until the MPI job finished.
    context.barrier()

    // Collect outputs and return.
    ???
  }
```

Architecture

Design Proposal

We propose to implement BarrierTaskContext.barrier() based on netty-based RPC client, introduce new BarrierCoordinator and new BarrierCoordinatorMessage, and new config to handle timeout issue.

BarrierCoordinator

BarrierCoordinator is a driver side RPCEndpoint that handles all global sync(barrier()) requests. It receives **RequestToSync** message and blocks until all the tasks have reached this routine.

A barrier() call happens on a barrier task, using `BarrierTaskContext.barrier()`. Since all the tasks in the same stage attempt (a barrier stage can attempt to retry for multiple times) share a common stageId + stageAttemptId, we can use stageId + stageAttemptId to identity a unique barrier() call.

There may be multiple active barrier() calls at a time (from different jobs containing barrier stage), possible ways to differentiate these calls at driver side include:

- Initiate separated BarrierCoordinator for each TaskSetManager, at executor side it's possible to lookup the corresponding BarrierCoordinator by stageId + stageAttemptId.
- Initiate only one BarrierCoordinator at driver side, and carry the stageId + stageAttemptId infos in each RequestToSync message.

The first solution introduces less code complexity, while the second solution requires less memory from the driver (and less RPCEndpoint resource). In practice we will implement BarrierCoordinator follow the second solution.

In case there may be multiple `barrier()` calls in a single stage, we use **barrierEpoch** to identify each call of `barrier()`. At driver side, barrierEpoch is increased when a global sync finishes (either all tasks have reached the routine or timeout), or set to -1 if the corresponding stage attempt has finished. At executor side, barrierEpoch is increased every time BarrierTaskContext.barrier() is called. ~~If the barrierEpoch from the executor side is not the same as that in the driver side, the barrier() call fails with a SparkException indicating a mismatched call of barrier() function is detected (Please refer to the example 2 in the API doc of barrier() function).~~ If the barrierEpoch from the executor side is not the same as that in the driver side, the barrier() call fails with a SparkException indicating the stage attempt has already finished.

BarrierCoordinatorMessage

RequestToSync is a BarrierCoordinatorMessage that requests global sync between all the barrier tasks in a stage. It carries the following variables:

- **numTasks** to indicate the number of global sync requests the BarrierCoordinator shall receive.
- **stageId** ID of current stage.
- **stageAttemptId** ID of current stage attempt.
- **taskAttemptId** unique ID of current task.
- **barrierEpoch** to identify which call of `barrier()` is generated from.

Timeout for BarrierTaskContext.barrier()

Add a new config **spark.barrier.sync.timeout** to specify the timeout of any call of `barrier()`. By default the value is 365 days so the barrier() call keep waiting for a year.

Security

The BarrierTaskInfos returned by BarrierTaskContext.getTaskInfos() are backed by BarrierTaskContextImpl.localproperties, which is transported from the driver to executors via RPCEndpoint, so the data security follows the security model of Spark PRC. Spark support AES-based encryption for RPC connections, it also support SASL-based encryption, which is required when talking to shuffle services from Spark versions older than 2.2.0.

Python support for BarrierTaskContext.barrier() open a socket inside executor JVM for each barrier task, to receive barrier() function calls from python side. The security is guarded by doing authentication in the beginning of socket connection, but it still worth to mention here in case there can be potential security issue we haven't realized.

Logging

Log an Info message in case a task enters the global sync (BarrierTaskContext.barrier()).
Log an Info message periodically in case a task is waiting under the global sync (with the time it has spent on waiting).
Log an Info message in case a task finishes the global sync (also with the time it has spent on waiting).
Log the process of the barrier on driver side, eg. "Barrier Update received from Task XYZ(15/500)".

Testing Plan

Unit Tests

Add a new test suite **BarrierCoordinatorSuite** to make sure the coordinator works as expected.
Add test cases for new config **spark.barrier.sync.timeout** to ensure it takes an effect and handles edge values correctly.

Benchmark

Benchmark on different size of clusters the implementation of BarrierTaskContext.barrier() to see how long it takes between the driver receives the last **RequestToSync** message and the tasks on executors end waiting for global sync.

Development Plan

Target version: Spark 2.4 (code freeze est. Aug 1st)

- Implement BarrierTaskContext.barrier() (~ July 27th)
- Add a benchmark (July 28th ~ July 29th)