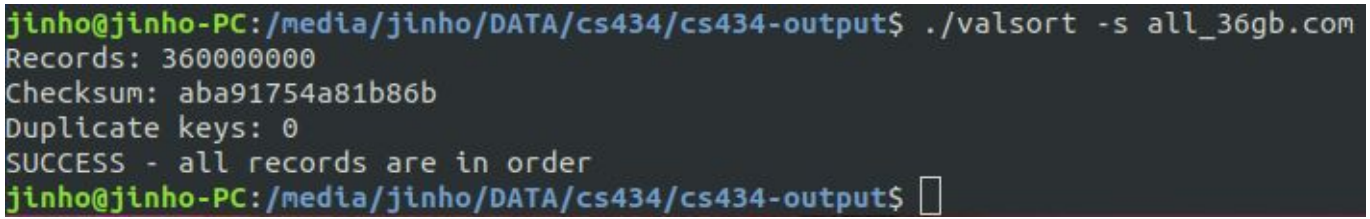
Project Final Report

(DPsort : Distributed, Parallel External MergeSort)

Jinho Ko
jinho.ko@postech.ac.kr

Result Summary

- Overall pipeline works very well with enough robustness
  - Implemented ~4,000 lines across ~50 code files
- Succeeded to sort :
  - **36GB** of unordered files *(couldn't expand more due to environment issue, but it will work for bigger sizes for sure)*
  - 3 workers, 4 threads/worker (tot. 12 threads)



- Demo video : https://www.youtube.com/watch?v=Kq2B0tbgGiE

Profiling Analysis

- (+) Good exploitation of multi-threads (4 threads / worker)
- (+) Limited use of memory (total block size / worker = 400MB)
  - Likely to scale enough for larger inputs
- (-) Highly dependent to disk throughput

Core Design Concept

- Master is in charge of everything!
  - Even for the partition list of all workers!

- Worker is just a machine that receives a tasks and reports the result
  - Goal : to make worker **stateless** as much as possible
    - Input : Task message (task type, input partition)
    - Output : Task result message (Success/Failure, sampled output)
  - Should be robust to task failure

Terminology

Stage := An execution unit in the program workflow

Task := An execution unit in each worker thread to be executed. *(A minimal unit of input!)* Typically, a task is assigned a single partition as its input.

Overall Program Pipeline

Stage	Task Used	What task does
GenBlockStage	GenBlockTask	- given a data file, split into a set of smaller partitions
LocalSortStage	LocalSortTask	- given a partition, perform sorting
SampleKeysStage	SampleKeysTask	- given a partition and sample ration, scan and sample - return the sampled data*
PartitionAndShuffleStage	PartitionAndShuffleTask	- given a partition and partitioning function, split partition into blocks - write 1 partition locally and shuffle out the rest
MergeStage**	MergeTask	- given 2 partitions, merge them linearly and save
TerminateStage	TerminateTask	- if whole pipeline succeeds, move output partitions to output directory. otherwise, delete work directory

* After the sampled data is aggregated by master, it will generate partition function and pass it to the next task
** Note that MergeStage will repeatedly executed until single partition remains to each worker

Considerations for Parallelism / Limited Resource

- Controllable performance parameters
- Resource-aware implementations / algorithms
  - A Must-follow rule => **Each worker thread should not occupy more than the size of one block!**
- Concurrency control
  - Thread-safe singleton objects

Performance Parameters

Controllable performance parameters :

- `dpsort.master.blockSizeInLines`
- `dpsort.master.maxSampleSize`
- `dpsort.master.maxSampleRatio`

- `dpsort.worker.threads`
- `dpsort.worker.maxShuffleConnections`
- `dpsort.worker.maxShuffleMsgBytes`

**These values increases as data / #workers grows*
=> Must be controlled to some level

Memory Constraint

Each worker thread should not occupy more than the size of single block!

```
def sortLines( lines: Array[Array[Byte]] ): Array[Array[Byte]] = {
  /* The algorithm is a in-place sorting algorithm
   * Thus no additional memory that exceeds
   * the partition size will be required
   * */
  logger.debug(s"in-place quicksort ${lines.size} lines")
  Sorting.quickSort(lines)( KeyOrdering )
```

```
def mergePartitions(inp1path: String, inp2path: String, outPath: String, lineSizeInBytes: Int): Unit = {

  val inp1Stream = new BufferedInputStream( new FileInputStream(inp1path) )
  val inp2Stream = new BufferedInputStream( new FileInputStream(inp2path) )
  val outStream = new BufferedOutputStream( new FileOutputStream(outPath) )
  try {
    val line1 = Array.fill[Byte]( lineSizeInBytes )( elem = 0 )
    val line2 = Array.fill[Byte]( lineSizeInBytes )( elem = 0 )
    var inp1Eof = false
    var inp2Eof = false
    inp1Stream.read( line1, off = 0, lineSizeInBytes )
    inp2Stream.read( line2, off = 0, lineSizeInBytes )
```

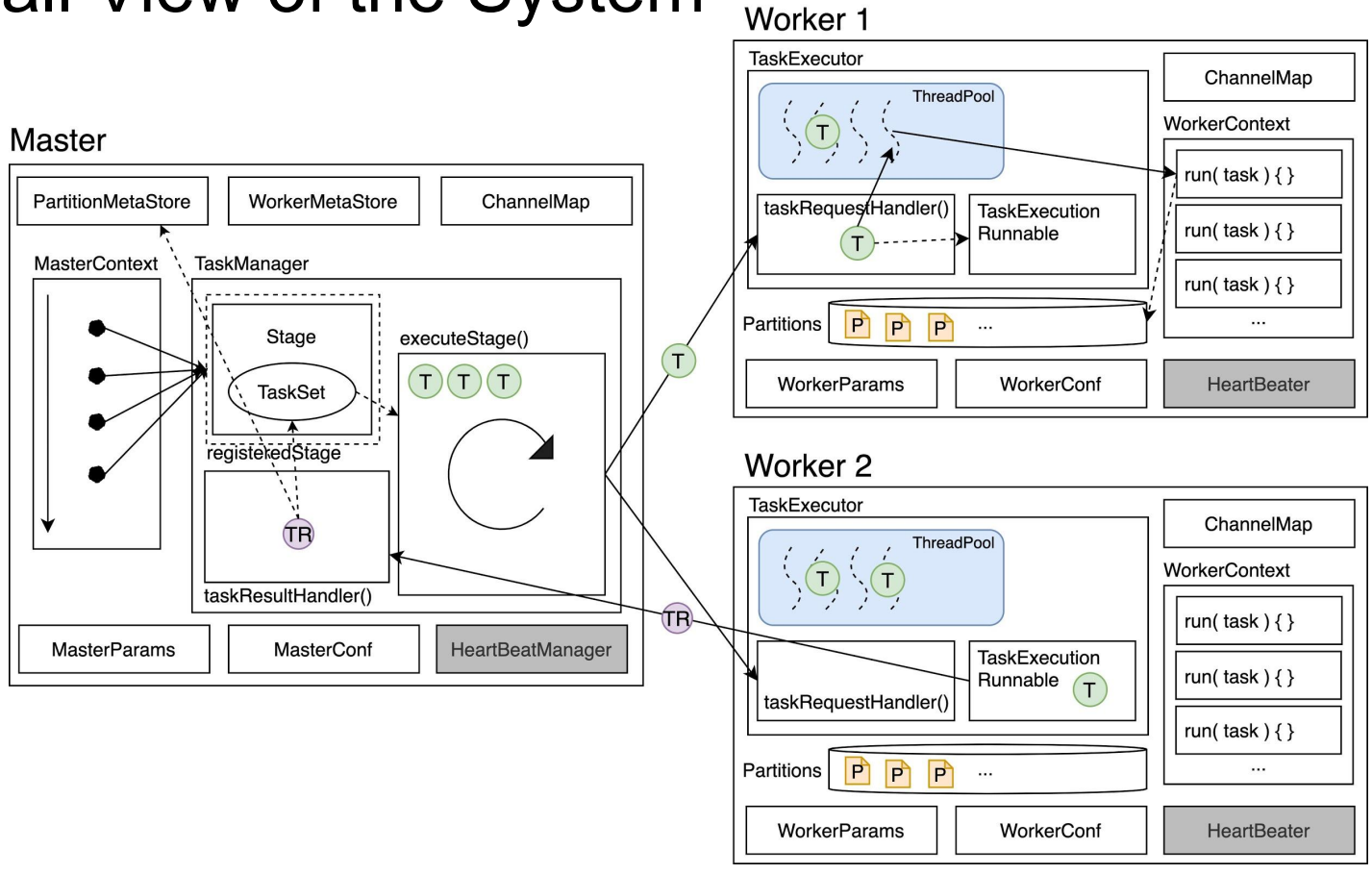
Concurrency Control

Most singleton objects (usually holding globally-accessible data) should be **thread safe**. We apply locks to resolve synchronization problem.

```
object PartitionMetaStore {

  /*
   * PartitionMetaStore requires strict concurrency control
   */
  private val partitionMetaStore
    = mutable.SortedMap[Int, ListBuffer[PartitionMeta]] ()

  val pmsLock: ReentrantReadWriteLock = new ReentrantReadWriteLock()
  def readlock: ReentrantReadWriteLock.ReadLock = pmsLock.readLock()
  def writeLock: ReentrantReadWriteLock.WriteLock = pmsLock.writeLock()

  def addPartitionMeta( workerID:Int, pmeta: PartitionMeta ): Unit = {
    writeLock.lock()
```

```
object ShuffleManager extends Logging {
...

  var numOngoingSendShuffles = 0
  private def numShuffleSendConnectionsLeft: Int = maxShuffle
  val shuffleSendLock: ReentrantLock = new ReentrantLock()  /
  private def isShuffleSendAvailable: Boolean = numShuffleSen

  var numOngoingReceiveShuffles = 0
  private def numShuffleReceiveConnectionsLeft: Int = maxShuf
  val shuffleReceiveLock: ReentrantLock = new ReentrantLock()
  private def isShuffleReceiveAvailable: Boolean = numShuffle
```

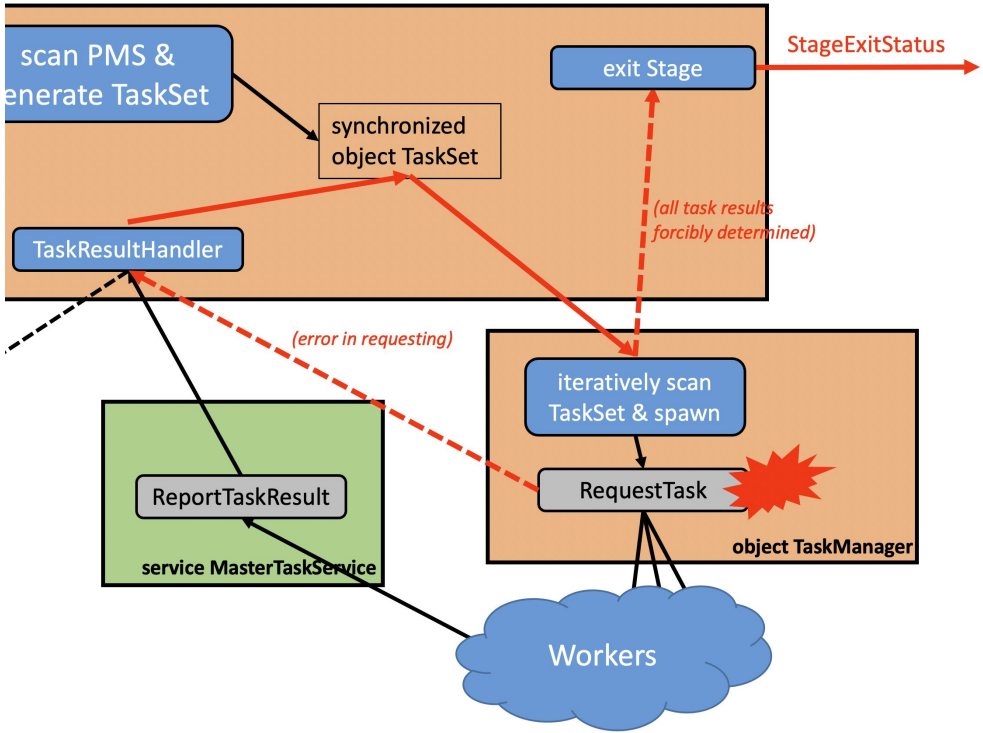
Overall View of the System



Robustness to Failure

For some reason TaskRunner cannot request tasks, it can directly call TaskResultHandler to mark failure

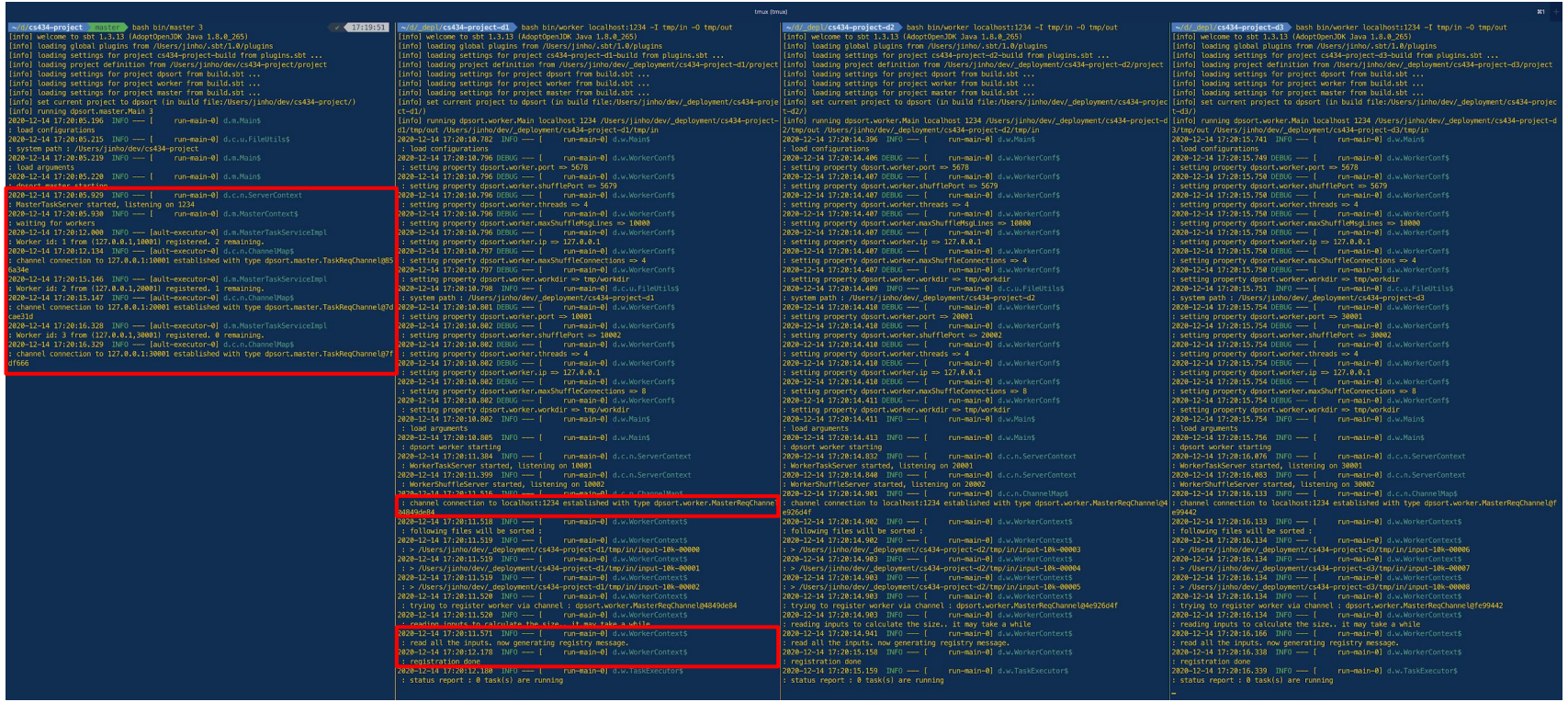
This is a very natural approach to cope with worker's error, without creating additional invariants.



Demonstration

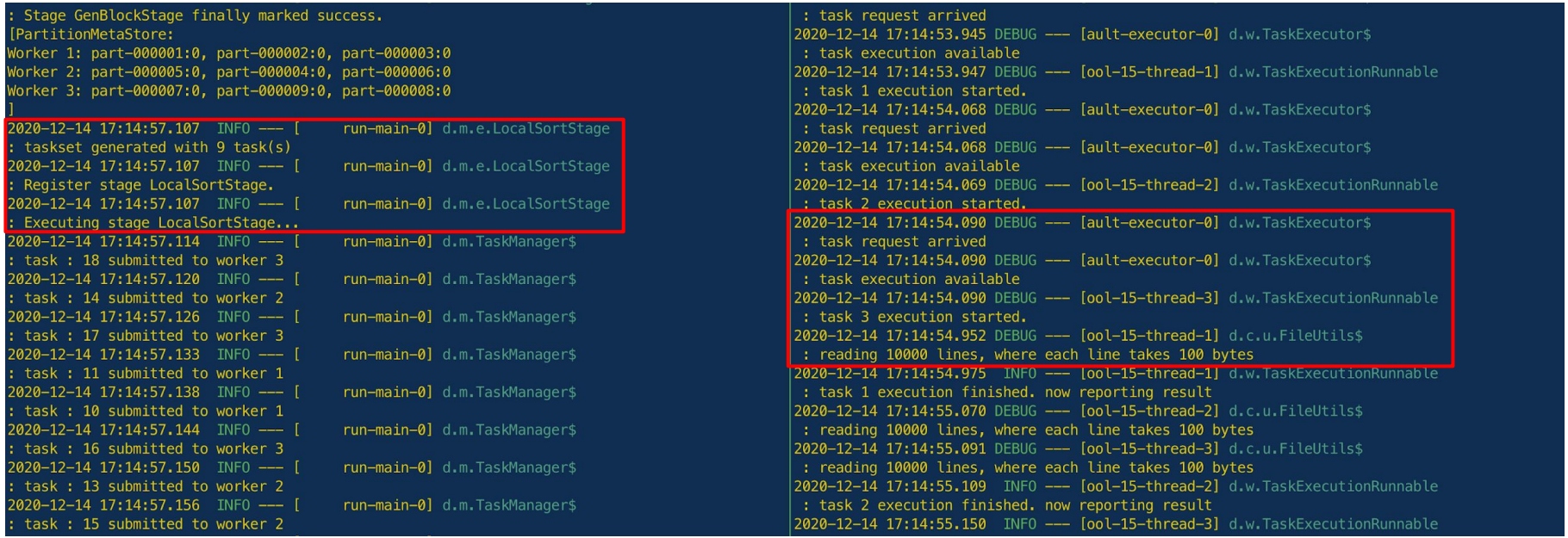
- Will mainly be done via Youtube video https://www.youtube.com/watch?v=Kq2B0tbgGiE

Running Example - Worker Registration



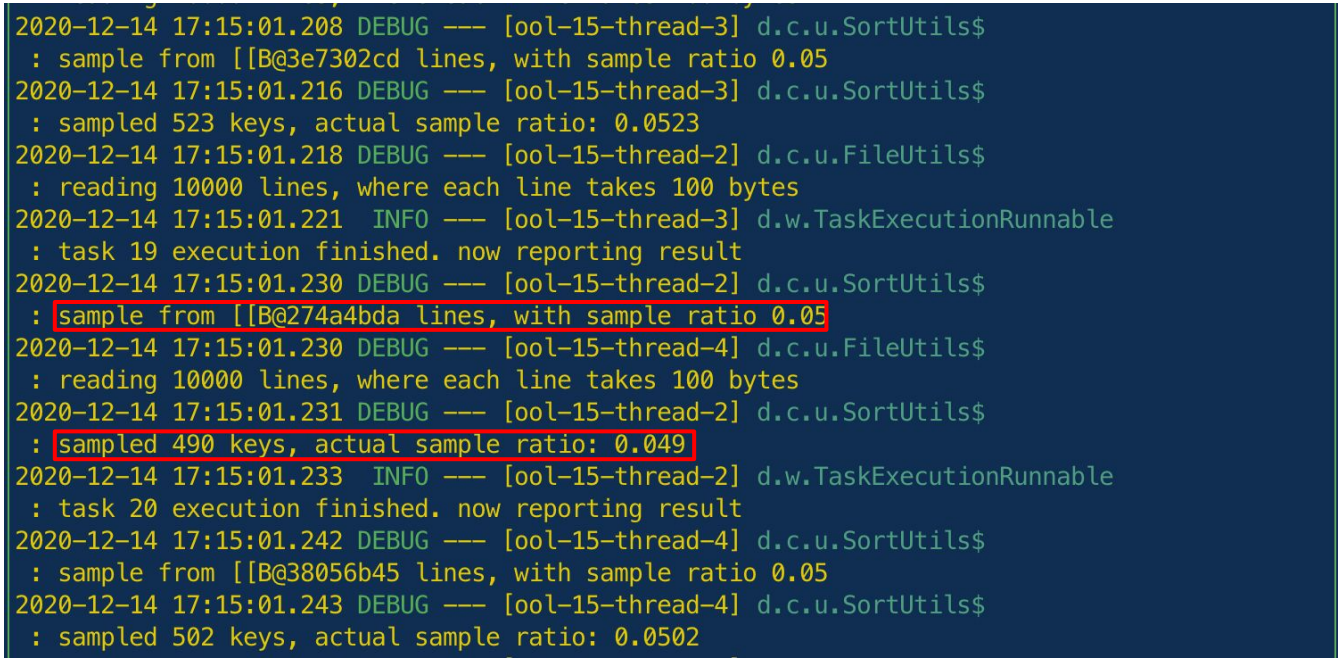
Each worker registers itself with its IP, port, and input partitions.

Running Example - Stage Generation



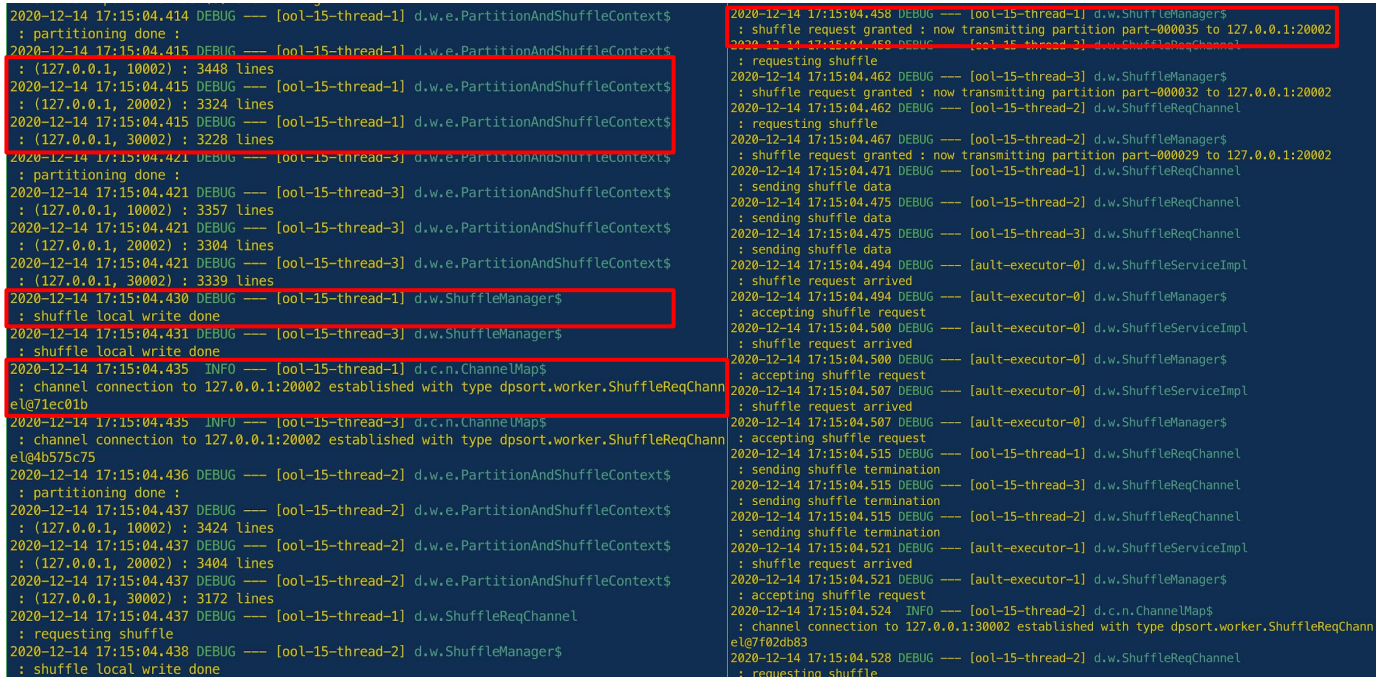
After one stage has finished, master then generates next stage with its taskset in it. It then registers the stage to TaskManager and triggers the execution. The worker will soon get task request and accept/decline upon thread pool availability.

Running Example - Sampling



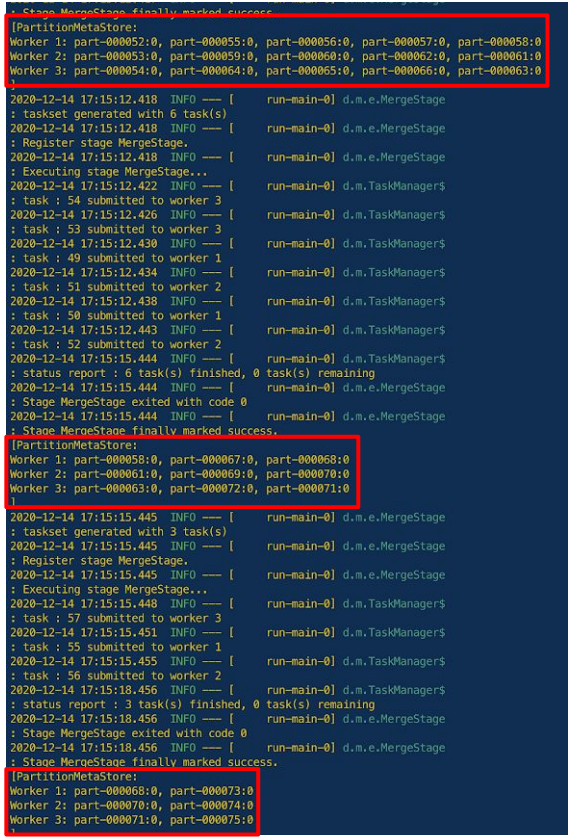
The worker is given a target sample ratio, and returns the sampled keys.

Running Example - PartitionAndShuffle



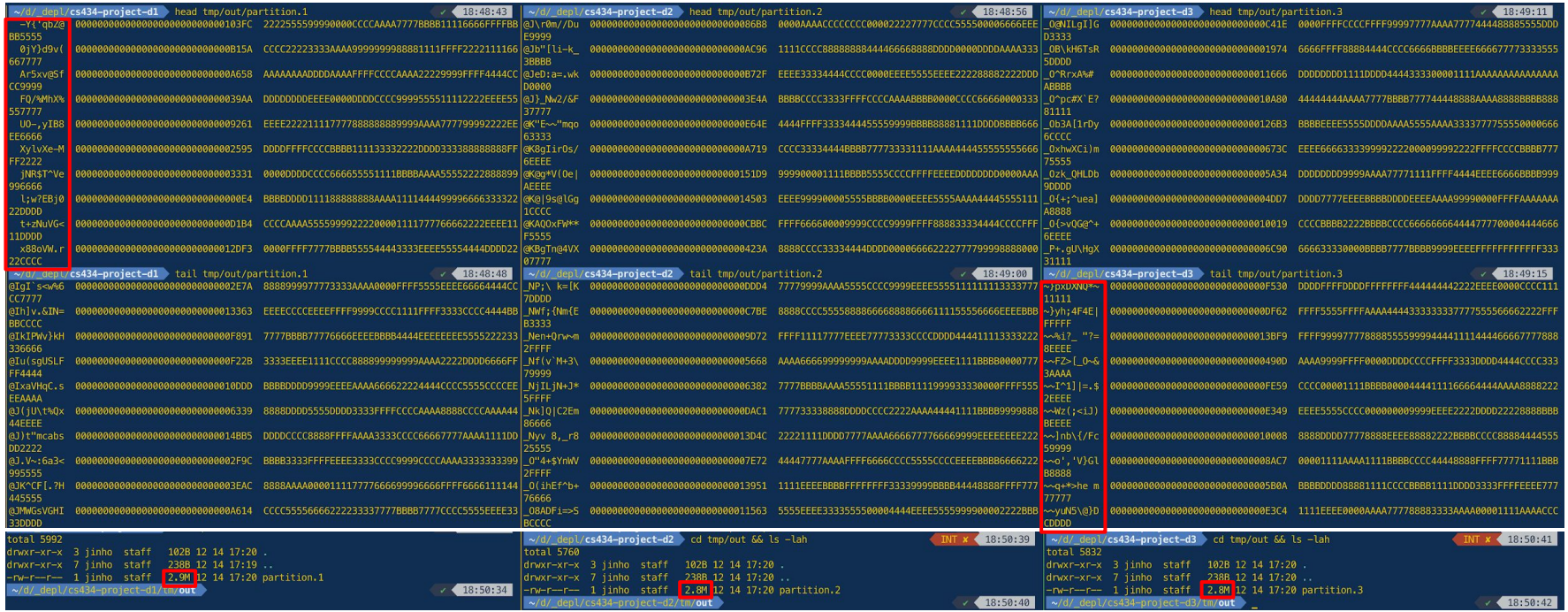
After partitioning, the worker will first write local partition to the work directory and shuffle out the rest. The ShuffleManager uses ChannelMap to retrieve the matching channel and request shuffle. The shuffle request will be granted upon availability. After the request has been granted, the worker will send chunks of partitions and finally the shuffle termination message.

Running Example - Merge



At each merge iteration, each task will merge a pair of input and output a merged partition. The stage repeats until each worker has a single partition.

Running Example - Output Validation



Sort success with equi-sized partitions

Applied Design Patterns

- Dependency Injection
  - when registering each stage to TaskManager
- Command Pattern
  - Receiver : TaskContext implementation
  - Commander : Task Object
  - Invoker : Each worker’s task executor

Dependency Injection

```
/* Execute Stages */

val stage0 = new GenBlockStage
lastStageExitStatus = stage0.executeAndWaitForTermination()
println(s"${PartitionMetaStore.toString}")

if( lastStageExitStatus == StageExitStatus.SUCCESS ) {
  val stage1 = new LocalSortStage
  lastStageExitStatus = stage1.executeAndWaitForTermination()
  println(s"${PartitionMetaStore.toString}")
}

if( lastStageExitStatus == StageExitStatus.SUCCESS ) {
  val stage2 = new SampleKeyStage
  lastStageExitStatus = stage2.executeAndWaitForTermination()
  genPartitionFunction
}

if( lastStageExitStatus == StageExitStatus.SUCCESS ) {
  val stage3 = new PartitionAndShuffleStage
  lastStageExitStatus = stage3.executeAndWaitForTermination()
  println(s"${PartitionMetaStore.toString}")
```

```
trait Stage extends Logging {

...

  def executeAndWaitForTermination(): StageExitStatus.Value = {
    logger.info(s"Register stage ${this.toString}.")
    val registerResult = TaskManager.registerStage( stage = this )
    if( !registerResult ) {
      logger.error(s"Cannot register stage to TaskRunner")
      return StageExitStatus.FAILURE
    }
    logger.info(s"Executing stage ${this.toString}...")

    val stageExitCode: Int = Await.result( executeStage, Duration.Inf )
```

```
object TaskManager extends Logging{

  private var registeredStage: Stage = null

  def taskResultHandler: TaskReportMsg => Unit = registeredStage.taskResultHandler

  def registerStage( stage: Stage ) : Boolean = {
    if ( registeredStage == null ) {
      registeredStage = stage
      true
    }
    else { false }
  }

  def executeStage: Future[Int] = {
    // submit until terminate condition is satisfied
```

In each stage execution, the function `executeAndWaitForTermination` will register(inject) itself to TaskManager and trigger its execution. *(the function waits for the future!)*

Command Pattern - Commander

```
@SerialVersionUID(1001L)
final class GenBlockTask(  i: Int,
                           wi: Int,
                           st: TaskStatus.Value,
                           inputPart: String,
                           outputPart: Array[String],
                           offsets: Array[(Int, Int)]
                    )
  extends BaseTask(i, wi, TaskType.GENBLOCKTASK, st, Array
    with Serializable {

}

@SerialVersionUID(1002L)
final class TerminateTask( i: Int,
                           wi: Int,
                           st: TaskStatus.Value,
                           inputPart: String,
                           outputPart: String,
                    )
  extends BaseTask(i, wi, TaskType.TERMINATETASK, st, Arra
    with Serializable {
}
```

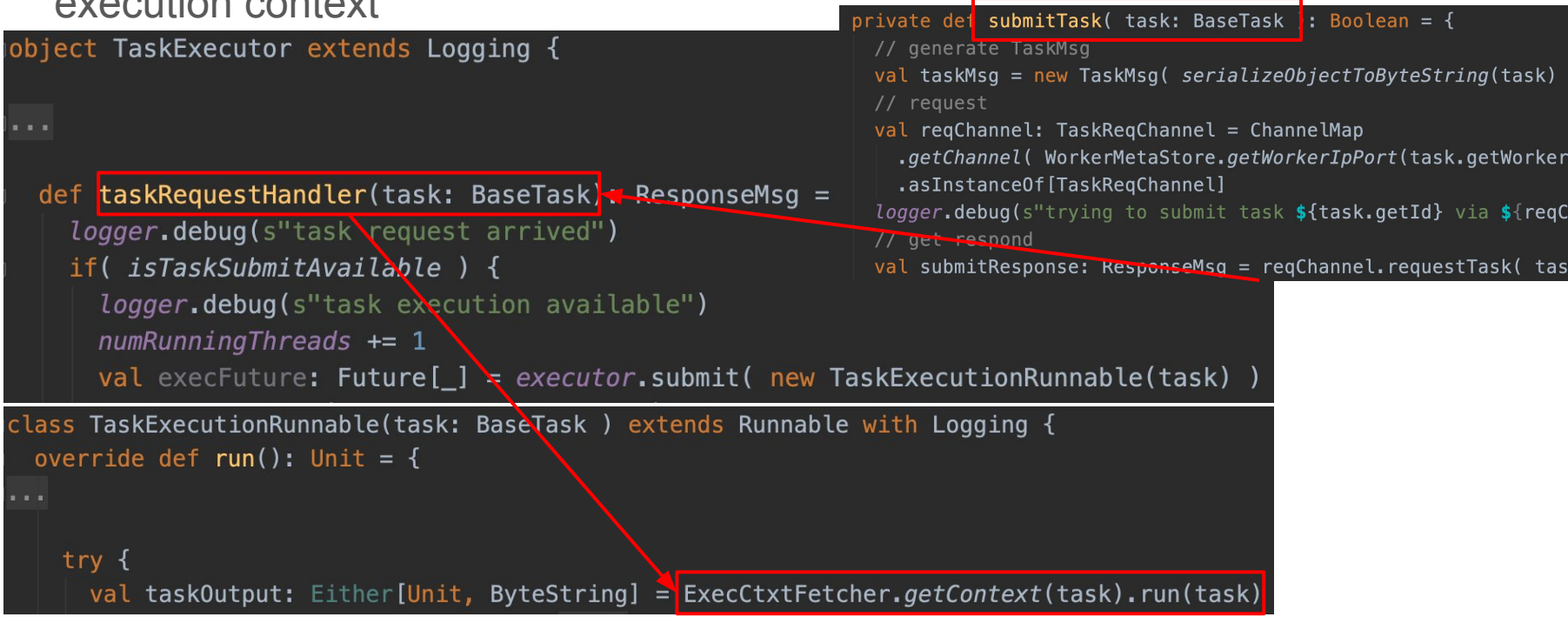
Each task object contains following parameters

- Task ID
- Worker ID which the task is assigned to
- Status
- Input partition(s)
- Output partition(s)
- etc.

Note that each task object should be serializable in order to be distributed across the network

Command Pattern - Invoker

- When task object is handled in, it invokes the `run()` method of that task's execution context



Command Pattern - Receiver

```
trait TaskExecutionContext {
  def run( _task: BaseTask ): Either[Unit, ByteString]
}

object EmptyContext extends TaskExecutionContext {

  def run( _task: BaseTask ): Left[Unit, Nothing] = {
    val task = _task.asInstanceOf[EmptyTask]
    val rndTime = new scala.util.Random(task.getId).nextInt(10)
    println(s"this is emptytask : wait for ${rndTime}(s) and finish");
    Thread.sleep( rndTime * 1000 )

    Left( Unit )
  }

}

object GenBlockContext extends TaskExecutionContext with Logging {

  def run( _task: BaseTask ): Left[Unit, Nothing] = {
    val task = _task.asInstanceOf[GenBlockTask]
    try {
      val filepath = task.inputPartition.head
      for( (outPartName,pIdx) <- task.outputPartition.zipWithIndex ){
```

Each execution context receives a task object.

Each context typecasts the input task object before accessing the parameters.

Further Improvements / Extensions

- Add robustness to worker failure *(they often can fail! => Network issue, GC overhead)*
  - Task retry policy
  - Heartbeat implementation

- Introduce buffers *(advanced partition management)*
  - Current approach is the Hadoop MapReduce approach : to read/write partitions on disk every operation
  - Spark approach can speed up significantly : to maintain in-memory as much as possible, and spill the rest to disk

What I’ve Learned

- Precautious design / Documentation matters!
  - Always write notes on *1. Design 2. Implementation scheduling & procedure*
  - Nevertheless, it is always never sufficient.
- Assertion(controlling invariants) indeed helps
  - Experienced from other course projects (OS, DB) as well
- Importance of testing
  - As code and data gets bigger, I started to regret not writing unit tests
  - The debugging time grows exponentially as the program gets closer to the production
- Automated environment configuration can reduce tons of unnecessary works
- Although the system is complicated, the code should remain simple
  - Complicacy comes from well-formed design, not from a complicated code

What I’ve Learned *(con’t)*

- Resource matters
  - Memory-aware programming
- Concurrency matters
  - Multiple threads incur deadlock / data consistency failure
  - Deadlock problem in shuffle request / resource management
- Knowledge is the power
  - An unclear notion on the external library (protobuf, fileIO, ..) will lead to misuse of time

References

- Github repo : https://github.com/dgggit/cs434-project
- Design documents : https://github.com/dgggit/cs434-doc/tree/master/project
- Scala Official Docs : https://docs.scala-lang.org/
- ScalaPB : https://scalapb.github.io/
- Stackoverflow, and other websites
- Slides from POSTECH CS434 2020F lecture videos