Assignment 3
Parallel Algorithms

Group Members:

Harshi Priya Yarragonda

Bhulakshmi Makkena

Mounika Dandamudi

1. Design a parallel sorting technique and implement it in spark.

Submit the following

a) Code through Github

b) Report with Algorithm, Sample Input and Output, Complexity and Time Performance.

c) Bonus for comparative evaluation with other techniques

GitHub Link:

https://github.com/BhulakshmiMakkena/PA

We select the Sorting technique "**Merge Sort**".

The sequential Merge Sort will be

```scala
/**
  * Created by hyarragonda on 10/23/16.
  */
object MergeSeq {
  def mergeSort(xs: List[Int]): List[Int] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      def merge(xs: List[Int], ys: List[Int]): List[Int] =
        (xs, ys) match {
          case(Nil, ys) => ys
          case(xs, Nil) => xs
          case(x :: xs1, y :: ys1) =>
            if (x < y) x::merge(xs1, ys)
            else y :: merge(xs, ys1)
        }
      val (left, right) = xs splitAt(n)
      merge(mergeSort(left), mergeSort(right))
    }
  }
}

```

Merge sort Algorithm sequential:

Algorithm:

1. If numbers >1; divide them into two halves.
2. Repeat step1 until it is divided completely
3. Sort the divided list and merge them
4. Repeat until all the lists are merged and sorted

Example:

Sample Input [3,6,5,1]

Sample Output [1,3,5,6]

1.If there is a list with numbers [3,6,5,1] then it basically partition the numbers like [3,6] and [5,1]

2. It is again partitioned into [3],[6] and [5],[1]

3. sort and merge them i.e., [3,6] and [1,5]

4.Merge again like [1,3,5,6]

Complexity: O(nlogn)

Merge Sort Parallel:

Algorithm:

1. Merge sort on individual Data list
2. After the individual list is sorted merge the sorted sequences of length n and each are distributed for 'p' processors.
3. Parallel Merge these sets

Examples:

If we are using two processors, then divide the input into half and the first half is given to 1st Processor and the other to 2nd Processor

Sample Input:[3,6,5,1]

P1-[3,6] P2-[5,1]

Sample Output:[1,3,5,6]

Complexity: If the data is divided into 2^d with N number of processors, then O(N/P*(log P/N)

For the Parallel Algorithm, the dependencies we used in SBT is

```
name := "PA"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.scala-lang" % "scala-actors" % "2.11.8"

libraryDependencies += "com.typesafe.akka" % "akka-actor_2.11" % "2.4.11"

libraryDependencies += "com.google.guava" % "guava" % "r05"

libraryDependencies += "commons-io" % "commons-io" % "2.4"
libraryDependencies ++= Seq(
  "org.scalaz" %% "scalaz-core" % "7.1.0",
  "org.scalaz" %% "scalaz-effect" % "7.1.0",
  "org.scalaz" %% "scalaz-typelevel" % "7.1.0",
  "org.scalaz" %% "scalaz-scalacheck-binding" % "7.1.0" % "test"
)

scalacOptions += "-feature"

initialCommands in console := "import scalaz._, Scalaz._"
```

The Input stream is divided into different chunks and each chunk is given to different processor to sort them individually and store them in a list

```
def sort(inputStream: InputStream, chunkSize: Int): Future[File] = {

  // open source stream
  val source: Stream[Int] = Source.fromInputStream(inputStream)
    .getLines()
    .map(_.toInt)
    .toStream

  val linesStream: EphemeralStream[EphemeralStream[Int]] = lift(source, chunkSize)
  val chunkCounter = new AtomicInteger(0)

  val sortedFileDir = Files.createTempDir()
  sortedFileDir.deleteOnExit()

  // read source stream, read n entries into memory and save it to file in parallel.
  val saveTmpFiles: Future[List[File]] = Future.sequence(
    linesStream.map(s => {
      val chunk = chunkCounter.getAndIncrement
      Future {
        println("sorting chunk: " + chunk)


        val ret = new File(sortedFileDir, "%d".format(chunk * chunkSize))
        val out = new PrintWriter(new BufferedOutputStream(new FileOutputStream(ret)))


        println("finished sorting chunk: " + chunk)
        ret
      }
    }).toList
```

These individual list are the merged together in the later step

```
// perform merge sort.
saveTmpFiles.map {
  files => {
    var merged = files
    while (merged.length > 1) {
      val splited = merged.splitAt(merged.length / 2)
      val tuple = splited._1.zip(splited._2)

      val m2 = tuple.map {
        case (f1, f2) => {
          println(s"merging ${f1.getPath} ${f2.getPath}")
          val ret = new File(sortedFileDir, UUID.randomUUID().toString)

          val source1 = Source.fromFile(f1)
          val source2 = Source.fromFile(f2)
          val out = new PrintWriter(ret)

          try {
            val stream1 = source1.getLines().map(_.toInt).toIterable
            val stream2 = source2.getLines().map(_.toInt).toIterable
            merge(stream1, stream2).foreach(out.println(_))
            ret
          } finally {
            out.close()
            source1.close()
            source2.close()

            FileUtils.deleteQuietly(f1)
            FileUtils.deleteQuietly(f2)
          }

        }
      }
      merged = if (merged.length % 2 > 0) {
        m2 :+ merged.last
      } else {
        m2
      }
    }
    merged.head
  }
}
}
```

Based on the chunk size divide the originalStream

```
private def lift[A](stream: Stream[A], chunkSize: Int): EphemeralStream[EphemeralStream[A]] = {

  def tailFn(remaining: Iterable[A]): EphemeralStream[EphemeralStream[A]] = {
    if (remaining.isEmpty) {
      EphemeralStream.emptyEphemeralStream
    } else {
      val (head, tail) = remaining.splitAt(chunkSize)
      EphemeralStream.cons(EphemeralStream.fromStream(head.toStream), tailFn(tail))
    }
  }
  val (head, tail) = stream.splitAt(chunkSize)
  return EphemeralStream.cons(EphemeralStream.fromStream(head), tailFn(tail))
}
```

Merge the separated stream into single stream as shown

```
private def merge[A](iteratorA: Iterable[A], iteratorB: Iterable[A])(implicit ord: Ordering[A]): Stream[A] = {

  (iteratorA.isEmpty, iteratorB.isEmpty) match {
    case (true, true) => Stream.Empty
    case (false, true) => iteratorA.toStream
    case (true, false) => iteratorB.toStream
    case _ => {
      def a = iteratorA.head
      def b = iteratorB.head

      if (ord.compare(a, b) > 0) {
        Stream.cons(a, merge(iteratorA.tail, iteratorB))
      } else {
        Stream.cons(b, merge(iteratorA, iteratorB.tail))
      }
```

c. Now if we consider Quick sort and compare it with the Merge sort we obtain

QuickSort:

Example:
Input: {59,45,34,76,57 }

Choose 59 as pivot.

Lower Sublist will be {45,34,57}→ choose 45 as pivot

then low list will be {34} and High list will be {57}

High List will be {76}

Final Output:{34,45,57,59,76}

Both Merge sort and Quick sort does not Parllelize good enough since the efficiency of the processor is very low.

Complexity: o(N/p * log(N/p))

Parallel mergesort - O(n)
Parallel quicksort - O(n)
But when it is unbalanced then it will be O(n^2)