Class Number : 6240
HW # 2
Name : Sneha Saran

Changes are highlighted below:

Source Code :

1. **NoCombiner**

```
public class NoCombiner {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
           String tmp = word.toString();
           if((tmp.startsWith("m")) || (tmp.startsWith("n")) ||
                              (tmp.startsWith("o")) || (tmp.startsWith("p")) ||
                              (tmp.startsWith("q")) ||(tmp.startsWith("M")) ||
                              (tmp.startsWith("N")) || (tmp.startsWith("O")) ||
                              (tmp.startsWith("P")) || (tmp.startsWith("Q"))) {
                     context.write(word, one);
           }
       }
   }
 }

 // This is the customer partitioner
// The partitioning phase takes place after the map phase and before
// the reduce phase. The number of partitions is equal to the number
// of reducers. The data gets partitioned across the reducers according
// to the partitioning function . The difference between a partitioner
// and a combiner is that the partitioner divides the data according to
// the number of reducers so that all the data in a single partition gets
// executed by a single reducer.
// However, the combiner functions similar to the reducer and processes
// the data in each partition. The combiner is an optimization to the reducer.

// Partition 0 : words starting with 'm' or 'M'
// Partition 1 : words starting with 'n' or 'N'
// Partition 2 : words starting with 'o' or 'O'
// Partition 3 : words starting with 'p' or 'P'
// Partition 4 : words starting with 'q' or 'Q'

// Input to the getPartition function is:
// The key and value are the intermediate
// key and value produced by the map function.
// The numReduceTasks is the number of reducers used in the MapReduce
// program and it is specified in the driver program.

// To prevent divide by zero exception, I am returning the
// (partitioner number) mod numReduceTasks.


public static class Partition extends Partitioner<Text, IntWritable> {

            @Override
            // arg0 -- key
            // arg1 -- value
            // arg2 -- no of reducers
            // In this example there are 5 partitioners.

            public int getPartition(Text arg0, IntWritable arg1, int numReduceTasks) {

                  String word = arg0.toString(); // key is the word starting with M,n ....

                  if( (word.startsWith("m")) || (word.startsWith("M"))) {
                        return 0;
                  }

                  if ((word.startsWith("n")) || (word.startsWith("N"))) {
                        return 1 % numReduceTasks;
                  }

                  if ((word.startsWith("o")) || (word.startsWith("O"))) {
                        return 2 % numReduceTasks;
                  }

                  if ((word.startsWith("p")) || (word.startsWith("P"))) {
                        return 3 % numReduceTasks;
                        }

                        if ((word.startsWith("q")) || (word.startsWith("Q"))) {
                                return 4 % numReduceTasks;
                        }

                        return 0;
                }

        }


public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
      int sum = 0;
      for(IntWritable val : values){
          sum += val.get();
      }
      context.write(key, new IntWritable(sum));
  }
}

public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();

  Job job = Job.getInstance(conf,"NoCombiner");
  job.setJarByClass(NoCombiner.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setNumReduceTasks(5);

  job.setMapperClass(Map.class);


  //job.setCombinerClass(Reduce.class);    //Combiner disabled
  job.setPartitionerClass(Partition.class);
  job.setReducerClass(Reduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);
```

```
   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
 }

}
```

2. SiCombiner

```
public class SiCombiner {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
           String tmp = word.toString();
           if((tmp.startsWith("m")) || (tmp.startsWith("n")) ||
                               (tmp.startsWith("o")) || (tmp.startsWith("p")) ||
                               (tmp.startsWith("q")) ||(tmp.startsWith("M")) ||
                               (tmp.startsWith("N")) || (tmp.startsWith("O")) ||
                               (tmp.startsWith("P")) || (tmp.startsWith("Q"))) {
                     context.write(word, one);
           }
       }
   }
 }

//This is the customer partitioner
// The partitioning phase takes place after the map phase and before
// the reduce phase. The number of partitions is equal to the number
// of reducers. The data gets partitioned across the reducers according
// to the partitioning function . The difference between a partitioner
// and a combiner is that the partitioner divides the data according to
// the number of reducers so that all the data in a single partition gets
// executed by a single reducer.
// However, the combiner functions similar to the reducer and processes
// the data in each partition. The combiner is an optimization to the reducer.

// Partition 0 : words starting with 'm' or 'M'
// Partition 1 : words starting with 'n' or 'N'
// Partition 2 : words starting with 'o' or 'O'
// Partition 3 : words starting with 'p' or 'P'
// Partition 4 : words starting with 'q' or 'Q'

// Input to the getPartition function is:
// The key and value are the intermediate
// key and value produced by the map function.
// The numReduceTasks is the number of reducers used in the MapReduce
// program and it is specified in the driver program.

// To prevent divide by zero exception, I am returning the
// (partitioner number) mod numReduceTasks.


 public static class Partition extends Partitioner<Text, IntWritable> {

                @Override
                // arg0 -- key
                // arg1 -- value
                // arg2 -- no of reducers
                // In this example there are 5 partitioners.

                public int getPartition(Text arg0, IntWritable arg1, int numReduceTasks) {

                        String word = arg0.toString(); // key is the word starting with M,n ....

                        if( (word.startsWith("m")) || (word.startsWith("M"))) {
                                return 0;
                        }

                        if ((word.startsWith("n")) || (word.startsWith("N"))) {
                                return 1 % numReduceTasks;
                        }

                        if ((word.startsWith("o")) || (word.startsWith("O"))) {
                                return 2 % numReduceTasks;
                        }

                        if ((word.startsWith("p")) || (word.startsWith("P"))) {
                                return 3 % numReduceTasks;
                        }

                        if ((word.startsWith("q")) || (word.startsWith("Q"))) {
                                return 4 % numReduceTasks;
                        }

                        return 0;
                }

            }


public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {
      int sum = 0;
      for(IntWritable val : values){
          sum += val.get();
      }
      context.write(key, new IntWritable(sum));
  }
}

public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();

  Job job = Job.getInstance(conf,"SiCombiner");
  job.setJarByClass(SiCombiner.class);

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setNumReduceTasks(5);

  job.setMapperClass(Map.class);


  job.setCombinerClass(Reduce.class);    //Combiner enabled
  job.setPartitionerClass(Partition.class);
  job.setReducerClass(Reduce.class);

  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);
```

```
   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
 }

}
```

3. PerMapTally

```
public class PerMapTally {

            // Input to Mapper: Offset to a line in the file, LongWritable
            // Input to mapper 2 : line itself, Text
            // Output of mapper: a single word from the line, Text
            // Output of mapper 2: count of that word, initialzed to one.
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       //Create a hashmap in Map function, this will be returned for every line in that chunk.
       HashMap<String, Integer> hm = new HashMap<String, Integer>();

       while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
           String tmp = word.toString();
           if((tmp.startsWith("m")) || (tmp.startsWith("n")) ||
                         (tmp.startsWith("o")) || (tmp.startsWith("p")) ||
                         (tmp.startsWith("q")) ||(tmp.startsWith("M")) ||
                         (tmp.startsWith("N")) || (tmp.startsWith("O")) ||
                         (tmp.startsWith("P")) || (tmp.startsWith("Q"))) {
                     if(hm.containsKey(tmp)) {
                                 hm.put(tmp, hm.get(tmp) +1);
                     }
                     else{
                                 hm.put(tmp, 1);
                     }
                     // Remove context.write, because we don't want to emit for each word.
                     // we want to emit for each line.
```

```
                    // Therefore move it outside while loop.
          }
      }
      // Iterate on hashmap and emit all keys and values -- this is for each line.

      for (String oneKey : hm.keySet()) {
          context.write (new Text (oneKey), new IntWritable(hm.get(oneKey)));
      }
      //context.write(word, one);
   }
 }

//This is the customer partitioner
// The partitioning phase takes place after the map phase and before
// the reduce phase. The number of partitions is equal to the number
// of reducers. The data gets partitioned across the reducers according
// to the partitioning function . The difference between a partitioner
// and a combiner is that the partitioner divides the data according to
// the number of reducers so that all the data in a single partition gets
// executed by a single reducer.
// However, the combiner functions similar to the reducer and processes
// the data in each partition. The combiner is an optimization to the reducer.

// Partition 0 : words starting with 'm' or 'M'
// Partition 1 : words starting with 'n' or 'N'
// Partition 2 : words starting with 'o' or 'O'
// Partition 3 : words starting with 'p' or 'P'
// Partition 4 : words starting with 'q' or 'Q'

// Input to the getPartition function is:
// The key and value are the intermediate
// key and value produced by the map function.
// The numReduceTasks is the number of reducers used in the MapReduce
// program and it is specified in the driver program.

// To prevent divide by zero exception, I am returning the
// (partitioner number) mod numReduceTasks.


 public static class Partition extends Partitioner<Text, IntWritable> {

                    @Override
                    // arg0 -- key
                    // arg1 -- value
                    // arg2 -- no of reducers
                    // In this example there are 5 partitioners.

                    public int getPartition(Text arg0, IntWritable arg1, int numReduceTasks) {

                            String word = arg0.toString(); // key is the word starting with M,n ....

                            if( (word.startsWith("m")) || (word.startsWith("M"))) {
                                    return 0;
                            }

                            if ((word.startsWith("n")) || (word.startsWith("N"))) {
                                    return 1 % numReduceTasks;
                            }

                            if ((word.startsWith("o")) || (word.startsWith("O"))) {
                                    return 2 % numReduceTasks;
                            }

                            if ((word.startsWith("p")) || (word.startsWith("P"))) {
                                    return 3 % numReduceTasks;
                            }

                            if ((word.startsWith("q")) || (word.startsWith("Q"))) {
                                    return 4 % numReduceTasks;
                            }

                            return 0;
                    }

            }


public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {

      int sum = 0;
      for(IntWritable val : values){
          sum += val.get();
      }
      context.write(key, new IntWritable(sum));
  }
}
```

```
 public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();

   Job job = Job.getInstance(conf,"PerMapTally");
   job.setJarByClass(PerMapTally.class);

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setNumReduceTasks(5);
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(IntWritable.class);
   job.setMapperClass(Map.class);


   //job.setCombinerClass(Reduce.class);    //Combiner disabled
   job.setPartitionerClass(Partition.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
 }

}
```

4. **PerTaskTally**

```
public class PerTaskTally {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   HashMap<String, Integer> hm ;

   public void setup(Context context) throws IOException, InterruptedException{
         // Need to return a HM at the chunk level
         // Changes are updated in the global HM, thread, synchronization and other things
```

```
            // are taken care by Hadoop.
             hm = new HashMap<String, Integer>();
    }

    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
             String tmp = word.toString();
             if((tmp.startsWith("m")) || (tmp.startsWith("n")) ||
                                   (tmp.startsWith("o")) || (tmp.startsWith("p")) ||
                                   (tmp.startsWith("q")) ||(tmp.startsWith("M")) ||
                                   (tmp.startsWith("N")) || (tmp.startsWith("O")) ||
                                   (tmp.startsWith("P")) || (tmp.startsWith("Q"))) {
                         if(hm.containsKey(tmp)) {
                                   hm.put(tmp, hm.get(tmp) +1);
                         }
                         else{
                                   hm.put(tmp, 1);
                         }
             }
        }
    }

    public void cleanup(Context context) throws IOException, InterruptedException {
             //write context.write part here
             for (String oneKey : hm.keySet()) {
             context.write (new Text (oneKey), new IntWritable(hm.get(oneKey)));
       }

    }
 }

//This is the customer partitioner
// The partitioning phase takes place after the map phase and before
// the reduce phase. The number of partitions is equal to the number
// of reducers. The data gets partitioned across the reducers according
// to the partitioning function . The difference between a partitioner
// and a combiner is that the partitioner divides the data according to
// the number of reducers so that all the data in a single partition gets
// executed by a single reducer.
// However, the combiner functions similar to the reducer and processes
// the data in each partition. The combiner is an optimization to the reducer.

// Partition 0 : words starting with 'm' or 'M'
// Partition 1 : words starting with 'n' or 'N'
// Partition 2 : words starting with 'o' or 'O'
// Partition 3 : words starting with 'p' or 'P'
// Partition 4 : words starting with 'q' or 'Q'

// Input to the getPartition function is:
// The key and value are the intermediate
// key and value produced by the map function.
// The numReduceTasks is the number of reducers used in the MapReduce
// program and it is specified in the driver program.

// To prevent divide by zero exception, I am returning the
// (partitioner number) mod numReduceTasks.

 public static class Partition extends Partitioner<Text, IntWritable> {

                @Override
                // arg0 -- key
                // arg1 -- value
                // arg2 -- no of reducers
                // In this example there are 5 partitioners.

                public int getPartition(Text arg0, IntWritable arg1, int numReduceTasks) {

                        String word = arg0.toString(); // key is the word starting with M,n ....

                        if( (word.startsWith("m")) || (word.startsWith("M"))) {
                                return 0;
                        }

                        if ((word.startsWith("n")) || (word.startsWith("N"))) {
                                return 1 % numReduceTasks;
                        }

                        if ((word.startsWith("o")) || (word.startsWith("O"))) {
                                return 2 % numReduceTasks;
                        }

                        if ((word.startsWith("p")) || (word.startsWith("P"))) {
                                return 3 % numReduceTasks;
                        }

                        if ((word.startsWith("q")) || (word.startsWith("Q"))) {
                                return 4 % numReduceTasks;
                        }

                        return 0;
                }

        }


public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
       int sum = 0;
       for(IntWritable val : values){
           sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}

public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();

   Job job = Job.getInstance(conf,"PerTaskTally");
   job.setJarByClass(PerTaskTally.class);

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setNumReduceTasks(5);

   job.setMapperClass(Map.class);


   //job.setCombinerClass(Reduce.class);   //Combiner disabled
   job.setPartitionerClass(Partition.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
```

```
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
  }
```

Explanation:
The input key to the map function is the offset of each and every line in the input file, starting at 0. The input value to the map function is line in the input file. The file is split into lines and each line is the value passed to the map function.
Space acts as the delimiter.
This was found using display or System.out.println() function. Using this function, values were passed to key and then were checked to see what values were assigned to the variables.

Performance Comparison:

Program Name	**Running time, Run1**	**Running time, Run2**
No Combiner Config1	**3.47 seconds**	**3.50 seconds**
No Combiner Config2	**3.15 seconds**	**2.35 seconds**
Si Combiner Config1	**3.23**	**3.26**
Si Combiner Config2	**2.13**	**2.13**
PerMapTally Config1	**3.53**	**3.48**
PerMap Tally Config2	**2.43**	**2.38**
PerTask Tally Config1	**3.05**	**3.05**
PerTask Tally Config2	**2.01**	**2.01**

Refer to the terminology below:

- **Config 1**: 6 small machines (1 master and 5 workers)
- **Config 2**: 11 small machines (1 master, 10 workers)

<u>Questions:</u>

1. **Do you believe the combiner was called at all in program SiCombiner?**

From the log files generated, I can see that the combiner was called. The line which shows the combiner was called:

```
Combine input records=42842400
Combine output records=18678
```

2. **What difference did the use of a combiner make in SiCombiner compared to NoCombiner?**

Difference between sicombiner and nocombiner:
Because combiner was not called in noCombiner the number of inputs to reducer phase in no combiner was increased and hence increasing the load on network. It can be seen from :
SiCombiner:
Reduce input records=18678
NoCombiner:
Reduce input records=42842400

3. **Was the local aggregation effective in PerMapTally compared to NoCombiner?**

Considering the time there was no much of difference in noCombiner and perMapTally, No Combiner took almost the same time. Considering the ouput records from map phase:

NoCombiner: Map output records=42842400
PerMapTally: Map output records=40866300

The number of output records from Map to next phase got recuded by 2000000.

4. **What differences do you see between PerMapTally and PerTaskTally? Try to explain the reasons.**
   There is a drastic difference between perMapTally and perTask tally, this can be seen from log files:

   PerMapTally: Map output records=40866300
   PertaskTally: Map output records=18678

   As we can see the number of records got reduced to 18678 from 40866300, This is because instead of emitting the output record for each line we are sending output word count for particular chunk, this reduces the load on network.

5. **Which one is better: SiCombiner or PerTaskTally? Briefly justify your answer.**

Considering this situation PerTaskTally is much better than SiCombiner considering time required for the program to run since there is one more step involved i.e. the combiner step. But considering the number of records to be processed by reducer task it is same, this is same because the records emitted from the map task goes to combiner Which serves the same purpose as siCombiner since it combines the records together. So considering just the time perTaskTally is better considering the number of records processed by combiner they are same.

6. **NEW: Comparing the results for Configurations 1 and 2, do you believe this MapReduce program scales well to larger clusters? Briefly justify your answer.**

   Considering the time required by the program to run we can see that there is not much of an improvement as compared to number of machines used. This happens because we have a practitioner which sends the data to only 5 machines, Now since we are sending the data to only 5 machines adding more machines doesn't help much.