3D Transpose of Seismic Volume Data on Apache Spark

2015 Fall Parallel Computing Final Project
Advisor: Dr. Lei Huang
Student: Chao Chen

1. Introduction

1.1 Seismic 3D Volume Data

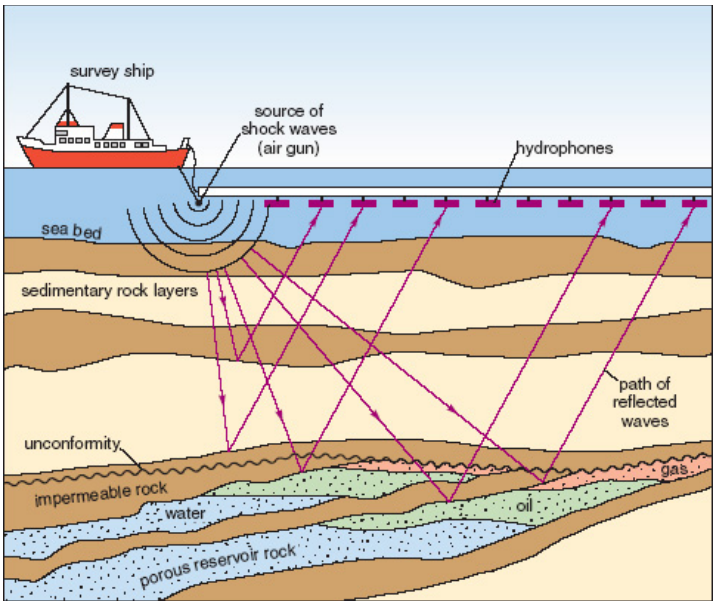


Figure 1.1 Reflection seismology Survey

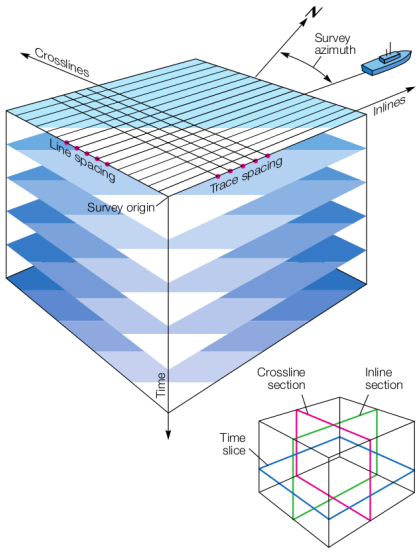


Figure 1.2 Seismic Volume Data

Seismic 3D volume is a collection of estimated property values of the Earth's subsurface, obtained through seismic reflection survey and organized in 3D spacing form. It's widely used in energy companies for geophysics analysis, which could conduct more accurate subsurface exploration and exploit.

As shown in Figure 1.2, the seismic volume data is defined through 3 different directions in 3D spacing: inline, cross-line and timeline. The standard industry data is usually stored in one specific direction, named inline-data, which stored in file slice by slice in cross-line direction and each slice is a inline section.

1.2 Transpose Problem and Sequential Code

Although it's convenient to do inline slices data fetching, this file sequence conducts a problem: If user needs data in other directions, organized as cross-line slice or time-line slice, the data fetching program must do seek between different file offsets to collect data of a single cross-line or time-line slice. This tedious procedure will slow down the whole software performance.

To accelerate the data analytic tasks, it's necessary to provide interface to freely transpose 3D volume data format between any direction pairs. The sequential version transposing procedure is straight forward, as shown in Figure 1.3 and 1.4:

```
val bis = new BufferedInputStream(new FileInputStream(data))
val bytes = new Array[Byte](I * J * K * 4)
val blocksize = 8192
var rb = blocksize
var in = 0

// Read volume data to inlines' array
while (bis.available > 0) {
    if (bis.available < blocksize) rb = bis.available
    bis.read(bytes, in, rb)
    in = in + rb
}
bis.close
val bb = ByteBuffer.wrap(bytes)
val inlineArray = new Array[Float](I * J * K)
for (i <- 0 until I) {
    for (j <- 0 until J) {
        for (k <- 0 until K) {
            inlineArray(i * K * J + j * K + k) = bb.getFloat()
        }
    }
}
```

Figure 1.3 Stencil code of inline 3D volume import

```
// Transpose
if (dstType == "i2x") {    //Inlines to Crosslines
    val xlineArray = new Array[Float](I * J * K)
    for (j <- 0 until J) {
        for (i <- 0 until I) {
            for (k <- 0 until K) {
                xlineArray(j * K * I + i * K  + k) = inlineArray(i * K * J + j * K + k)
            }
        }
    }
}
```

Figure 1.4 Stencil code of in-lines to cross-lines transpose

Obviously, it is an O(n3) time complexity procedure for the transposing part. However, when it comes to the actual industry or scientific datasets, usually beyond tera-bytes size, the performance bottleneck is not only the tedious transposing but also the sequential IO reading of the traditional file system(the while loop in Figure 1.3).

In this use case, I execute this sequential program to transpose an 1.7GB inlines volume file to cross-lines array on a single node of our cluster. The timeline statistics are shown as Figure 1.5. It takes 13 seconds to read data from storage into memory, and 1 second to transpose a 600 x 481 x 1501 3D volume.

```
Data: /home/cchen/data/Penobscot.a3d
Size I: 600
Size J: 481
Size K: 1501
Trans: i2x
start read inline volume: Fri Dec 04 13:04:36 CST 2015
bytes count: 1732754400
bytes array ready: Fri Dec 04 13:04:49 CST 2015
bytes buffer ready: Fri Dec 04 13:04:49 CST 2015
inline volume ready: Fri Dec 04 13:04:55 CST 2015
start transposing: Fri Dec 04 13:04:55 CST 2015
transposing done: Fri Dec 04 13:04:56 CST 2015
```

Figure 1.5 Performance statistics

1.3 Spark RDD Programming Model

Apache Spark is an open source cluster computing framework originally developed in the AMPLab at University of California, Berkeley but was later donated to the Apache Software Foundation where it remains today. In contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's multi-stage in-memory primitives provides performance up to 100 times faster for certain applications. By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well-suited to memory access intensive applications.

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Figure 1.6 SPARK RDD Programing Example: Words count

As shown in Figure 1.6, the main feature of Spark programming is utilizing RDD(Resilient Distributed Datasets) to distribute data, and RDD provides series of interfaces, such as map() function, for developer to perform algorithm on each distribution.

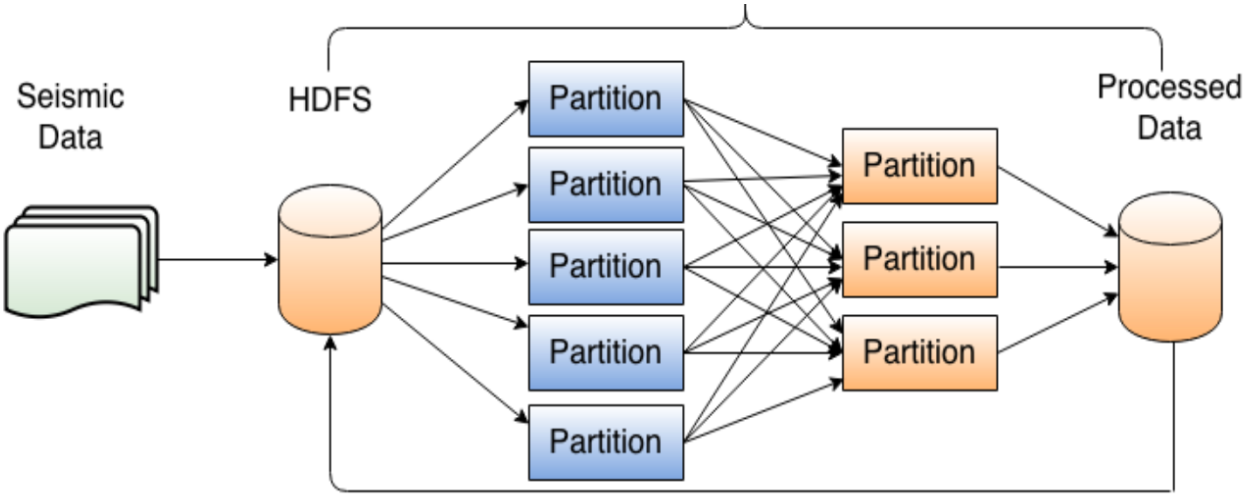


Figure 1.7 Seismic data distribution via Spark-HadoopFile RDD

As shown in Figure 1.7, our Spark Cluster is built on top of Hadoop distribution filesystem, which provides a full set of interface to distribute our data to the storage of each nodes. Thus, we could create a RDD of our distributed seismic data through Spark HadoopFile interface. This parallel procedure will be the biggest beneficial for the performance contrast to the sequential program, in which the data importing consuming the most time. At last, the remaining work, transposing, will be implemented in parallel mode by utilizing some Spark RDD operations.

2 Implementation

2.1 Traces Indexing

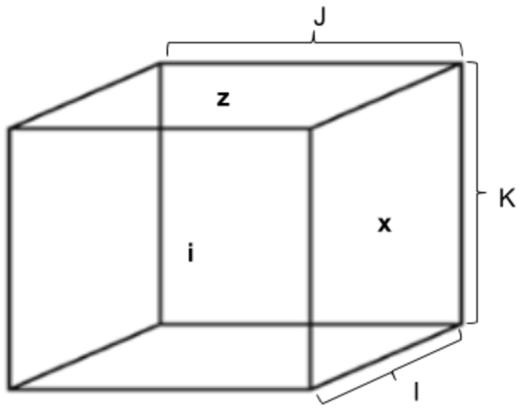


Figure 2.1 Seismic volume data dimensions: I x J x K

First, let's denote seismic volume data as shown in Figure 2.1, in which 'i' means inline slice, 'x' means cross-line slice and 'z' stands for time-line slice. The data is stored in i-slices format. To resolve transpose problem in each distribution evenly, we split the volume to I of i-slices in the seismic volume RDD and each i-slice is 2D matrix.

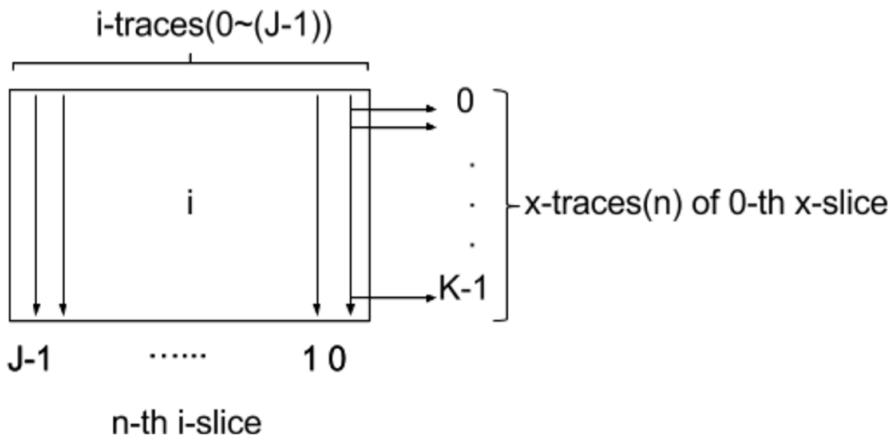


Figure 2.2 Inline slice and traces

As shown in Figure 2.2, each i-slice matrix consists of J of i-traces which have the length of K. An i-slice matrix could be iterated i-trace by i-trace. Since in 3D spacing, each i-trace is also the trace of x-slice, for example, the i-traces(0) is the x-trace of the 0-th x-slice, the i-traces(1) is the x-trace of 1st x-slice, etc.

Thus, we implement a map function to index all i-traces of the volume. The index is combined by index of i-trace and index of i-slice, as the transposeI2X() function shown in Figure 2.3.

```
def transposeI2X() = {

    def i2x(rdd:(BytesWritable,BytesWritable), dimK:Int, dimJ:Int, dimI:Int) = {
        val idxI = SeismicData.getIndex(rdd._1);
        val cnt = SeismicData.getNum(rdd._1);
        val lines = SeismicData.getFloatsPair(rdd._2, dimJ, dimK, cnt, idxI);
        var traces = new Array[(Int, Array[Float])](dimJ * cnt);
        for (i <- 0 until cnt) {
            val iline = lines(i)._2;
            val ikey = lines(i)._1;
            for (j <- 0 until dimJ) {
                val trace = iline.slice(j * dimK, (j + 1) * dimK);
                traces(i * dimJ + j) = ((j << 16) + ikey, trace);
            }
        }
        traces
    }

    println("transposeI2X");
    val ktraces = inlineBytesRDD.flatMap( {case (k,v) => i2x((k,v),sizeK,sizeJ,sizeI)} )
    xlineArrayRDD = ktraces.groupBy(grpFunc).map(sortFunc).sortByKey(true, sizeJ + 1);
    xlineArrayRDD
}
```

Figure 2.3 Inline slice and traces

2.2 Grouping

After indexing the map, we got a volume RDD with (i-trace-index)-(i-slice-index) index as the key, trace data as the value. As we mentioned in Figure 2.2, to get a x-slice, we need group all the traces with the same i-trace-index. Spark provides a RDD function groupBy() for this sort of operations, as shown in Figure 2.4. The code

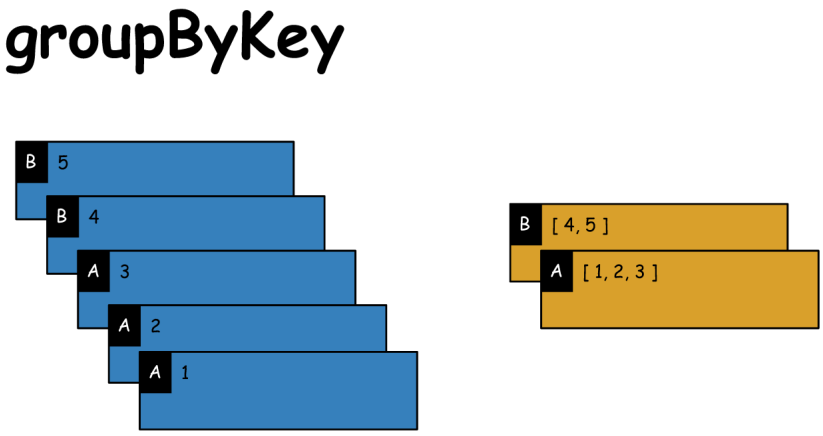


Figure 2.4 RDD groupBy operation.

2.3 Sorting

After grouping, we have already got the x-slices (cross-lines) data in our RDD distribution map. To organize them as a cross-line volume, all we need to do is sorting them by i-trace-index. Also, Spark provides a RDD function sortByKey() for this purpose.

After all previous steps, we get a cross-line RDD now.

3. Performance Analysis

3.1 Tools

I use Spark Web Interface and NMONVisualizer to do the performance and time statistics. Spark Web Interface is the official monitoring and instrumentation tool, which provides a web visualization interface to manage and monitoring the spark applications, especially helpful in task timeline analysis.
We installed NMON (general purpose performance monitor tool) in all nodes of cluster, it will collect all performance statistics and save as log files. To visualize these performance data, we use NMONVisualizer, a Java GUI tool for analyzing nmon system files from both AIX and Linux, to analysis logs and generate visual result.

3.2 Timeline Statistics

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20151202150525-0028	org.pvamu.sac.sdk.TransTest	224	15.6 GB	2015/12/02 15:05:25	root	FINISHED	17 s

Figure 3.1 Spark Web UI: Application execution time

Spark Jobs (?)

Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	saveAsHadoopFile at SeismicData.scala:776	2015/12/02 15:05:34	5 s	2/2 (1 skipped)	1082/1082 (600 skipped)
0	sortByKey at SeismicVolume.scala:264	2015/12/02 15:05:26	8 s	2/2	1200/1200

Figure 3.2 Spark Web UI: Application jobs timeline

Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total
1	sortByKey at SeismicVolume.scala:264	+details	2015/12/02 15:05:32	2 s	600/600
0	groupBy at SeismicVolume.scala:264	+details	2015/12/02 15:05:26	6 s	600/600

Figure 3.3 Spark Web UI: Job stages timeline

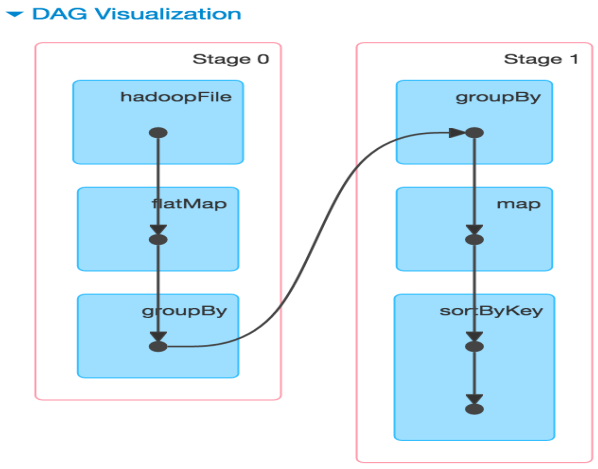


Figure 3.4 Spark Web UI: Operations timeline

As show in Figure 3.1, the total running time of spark transpose program is 17 seconds, however, we need to remove the time I spent on writing data in to disk for debug purpose, and also remove the job submitting and some server overhead time, which could be ignored in actual Tera-bytes scale application scenario. After focus on distributed data importing and transposing part, we can see the actual execution time is 8 seconds of job0, as shown in Figure 3.2.

From Figure 3.3 and 3.4 we can infer that, for the same 1.7GB seismic data, importing from

Figure 3.6 Nmon for all workers: Memory Usage

Hadoop filesystem and indexing all i-traces together costs 6 seconds (job0-stage0), while the sequential code takes 13 seconds only for read data into memory. The groupBy and sortByKey in job0-stage1 which equivalent to transposing part of sequential code cost 2 seconds, while the stencil code cost 1 second.

We can see Spark takes a huge advantage of sequential code in the aspect of file IO performance. Although in transposing part they are almost tied with each other in this case, it's only because of the relatively small 1.7GB data size, in which the performance benefits are not big enough to overcome the spark server and communication overheads. In actual industry application scenarios the data is usually tera-bytes level, in which a single machine could barely afford enough memory for this program, no mention to the parallel system will take lots of advantages in that case.

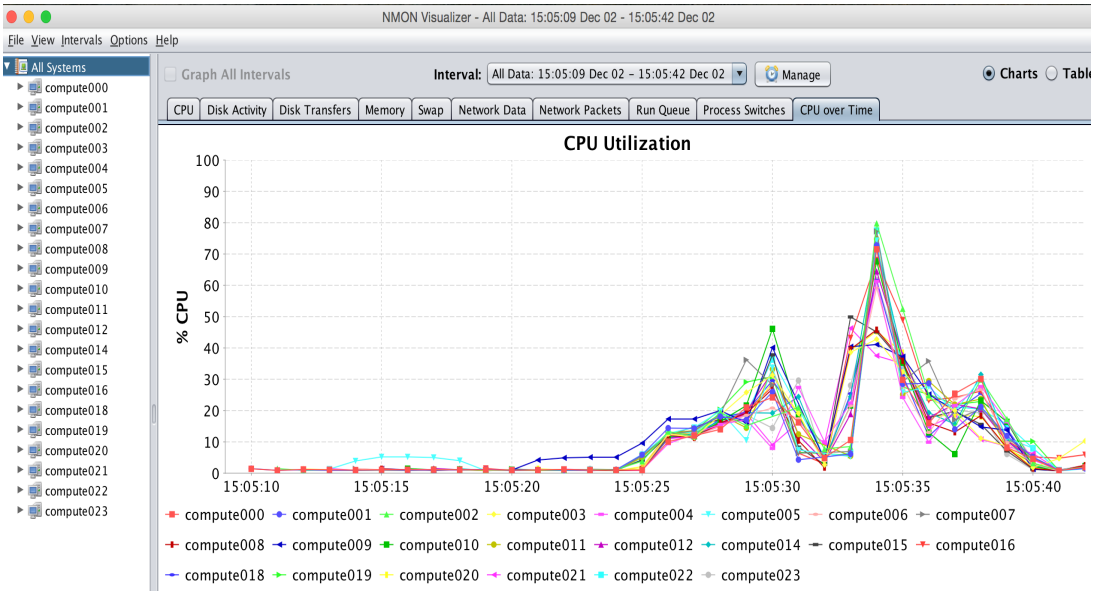
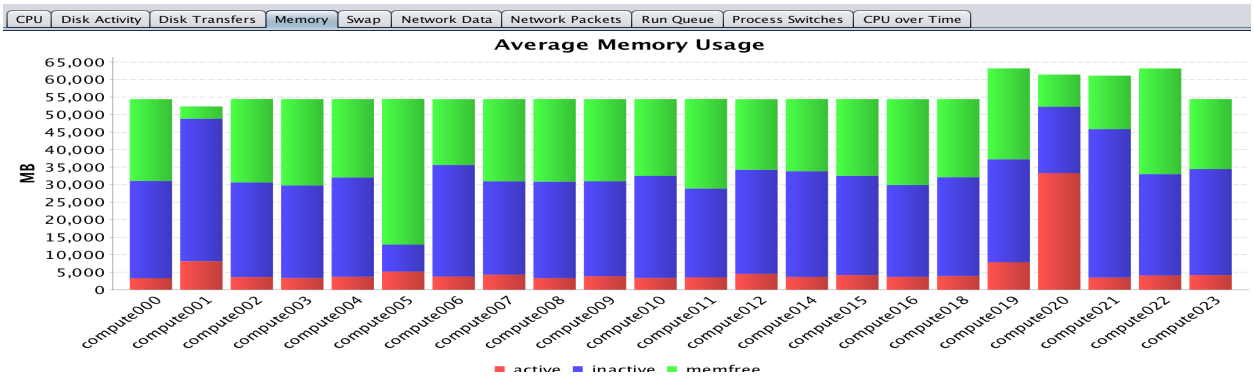


Figure 3.5 Nmon for all workers: CPU Utilization



3.3 Workers(Nodes) Activity Statistics

NMONVisualizer provides very detail working nodes statistics. As shown in below figures, the tasks and resources consuming are both evenly distributed to all nodes.

4. Conclusion

The Spark-Hadoop distribution system conducts a better performance in this 3D volume transpose case, especially in file IO performance aspect. From the performance statistics we could infer that when the data dependency exists among different distributions, the shuffle operation of Spark RDD needed for re-partition, such as groupBy(), will be the main bottleneck in performance. However, Spark is still a good choice for large scale budget applications, especially it provides a fault tolerance framework and an user-friendly management interface.