Big Data Approach to fMRI Data Analysis with Intel DAAL and Full Correlation Matrix Analysis

Yin Lok Wong

Abstract—This report describes the combination of the use and testing of the open source Spark NIfTI reader library biananes, dedicated for scalable fMRI data analysis, together with Intel Data Analytics Acceleration Library (DAAL), to realize analytic operations of fMRI data on the Big Data framework Apache Spark and replacing the use of Apache Spark MLlib with DAAL under the Computational Science for Undergraduate Research Experience programme (CSURE 2016).

I. INTRODUCTION

fMRI data analysis deals with data represented in large scale matrices. Operations on those matrices generate intermediate and resulting matrices and data that are often even larger, for example, the use of data in gigabytes might generate results in terabytes, which breeds an ideal scenario for the use of Big Data framework such as Apache Spark. However, the use of Big Data framework in fMRI data analysis is limited due to the difficulty of representing fMRI data in distributed data sets that can then be utilized by such frameworks.

The Resilient Distributed Dataset (RDD) is the primary data abstraction in Apache Spark, which is essentially the fault-tolerance collection of records. In memory transformations and actions on RDD support the core of operations on Spark. The use of open source Spark NIfTI Reader library -biananes enables NIfTI image files of fMRI data to be read into RowMatrix RDD for Apache Spark, and thus available to MLlib function calls.

However, some MLlib functions are limited in performance. According to the benchmark results from Intel upon the release of Intel Data Analytics Acceleration Library, on computation of principal component analysis under Spark, DAAL shows a most significant 7x performance boost compared with MLlib. Therefore, in the hope to enhance the analysis process, the use of MLlib is replaced by DAAL with tests to verify performance difference.

To replace MLlib with DAAL, adaptations are needed to make biananes return the desired data structure needed by DAAL. Conversion from RowMatrix to Numeric Table, or more specifically, Homogeneous Numeric Table, is needed. The adaptation essentially, and theoretically, is to read NIfTI data content into rows in Numeric Table, thus a different approach was used. Figure 1 shows the basic implementation logic of the updated biananes.

II. BIANANES\textsuperscript{4} UPDATE FOR DAAL

biananes is a library for fMRI data analysis on large datasets running on the Apache Spark framework, by reading fMRI NIfTI files into RowMatrix RDD in Spark MLlib. It is basically written in scala, and compiled with Java compiler. NIfTI is the most common file type in representing fMRI data, which is basically a collection of arrays representing different dimension of the fMRI image, plus some header information. biananes extracts the data stored in the multi-dimensional array in NIfTI files, and concatenates the data in a continuous row ordered by time volume. The resulting row is in RowMatrix RDD that can then be divided into partitions and distributed for computation on memory of respective worker nodes.

RowMatrix, however, is the data structure in Spark MLlib. The end goal of this project is to test and replace the use of MLlib with Intel DAAL. Therefore, adaptations on biananes is needed. The data format utilized by Intel DAAL in its sample codes for Spark is JavaPairRDD\langle Integer, HomogenNumericTable\rangle, where Integer takes the index number of the partition, or volume, HomogenNumericTable takes the Homogeneous Numeric Table storing the data content.

Homogeneous Numeric Table in DAAL represents features in columns and feature vectors in rows. The adaptation essentially, and theoretically, is to read NIfTI data content into rows in HomogenNumericTable. However, difficulties were encountered when attempting to translate data content in NIfTI to rows in Numeric Table, thus a different approach was used. Figure 1 shows the basic implementation logic of the updated biananes.

Development in later stage of the project required correlation computation by matrix multiplication on Numeric Table input. The construction of Numeric Table in the updated biananes as specified above was then problematic as the multiplication of a single column with its transpose results in the dot product of a single value, which deviates from the expectation of getting a correlation matrix.


3. Yida Wang et al., Full correlation matrix analysis of fMRI data, technical report ().
A new approach was then developed to generate Numeric Table with multiple rows representing vectors and columns representing features, the implementation logic is illustrated in figure 2.

In this approach, the maximum number of rows in a HomogenNumericTable is limited to the number of partitions, i.e., working nodes. Instead of instantiating a new table from every volume of nifti content, a new volume of nifti content will be appended to an existing table where possible, and a table will only be instantiated when the number of rows has reached the maximum number of rows or when there is no extra partition left.

A. Code Snippet

As mentioned above, the theoretical amendment made to biananes is to translate the construction of rows in RowMatrix to the construction of rows in HomogenNumericTable. However, the test image file contains a total of nearly 500,000 data points per volume, meaning nearly 500,000 columns in 1 single row if the structure of RowMatrix is retained.

This approach was built successfully. But when tested with DAAL sample code on Spark, segmentation fault occurred. An experimental attempt to avoid the problem, corresponding to the flowchart in figure 1, was to construct the table in its transpose manner, i.e. taking rows as a representation of features and columns as a representation of vectors. Basic implementation as in figure 3.

Fig. 3. Core implementation of single-column construction of JavaPairRDD(Integer, HomogenNumericTable)

Figure 3 shows the basic implementation of the first approach which mimics the construction of RowMatrix in the original biananes code. With this implementation, each volume of data is concatenated into a single column of a HomogenNumericTable.

Sample codes of DAAL, running SVD and QR computation returned results with this updated biananes. This was taken as the implementation for the benchmark tests. However, in later stage of development where matrix-matrix multiplication was needed, new changes were made to biananes.

Figure 4 shows the implementation of the second major update to biananes where multi-row Numeric Table are constructed according to the logic in figure 2.

Fig. 4. Core implementation of second update of biananes to give multi-row Numeric Table

This approach was built successfully, but when tested with data containing "fat" matrices - matrices with over 10,000 rows, segmentation fault occurred at runtime as with the row-orientated construction of Numeric Table.

Sample codes of DAAL, including QR decomposition and Covariance computation, returned result with the same implementation of biananes on input data with small scale matrices - around 500 rows and columns. The contrary in results with the same update of biananes has speculated doubts on whether DAAL supports the analysis of "fat" matrices.

B. Setup and Run Walkthrough

biananes is an open source library written in scala with dependencies on a number of other packages and libraries written in C/C++ or Java, including the Spark packages and the NIFTI i/o libraries named niftilib. The dependencies are managed by Apache Maven and thus the compilation and package of the library is done by updating the dependencies in the pom.xml correspondingly and then build with Maven.

The following is the note on how the library was built and used during this project. The platform used throughout the project is the cluster Beacon in NICS.

biananes was built in this project in the following steps,
1) install niftilib i/o library
2) install and load Apache Maven
3) install Intel DAAL
4) install daal.jar locally with maven-install
5) update dependencies in pom.xml
6) update NiftiTools.scala, which contains the implementation of functions in the library
7) use (maven clean & maven compile & maven package to build the library

Points to note when building the library,
1) daal.jar was installed and linked locally in the maven project, installation and dependencies update are needed when building the library on another platform or user login
2) sparkniftireader.c in bianaes_src/main/C/sparkniftireader/src/ contains functions to be called in NiftiTools.scala, compilation of this code into sparkniftireader.so is needed in advance for the build of biananes.jar
3) lib_so_path in NiftiTools.scala needs to be updated to refer to the correct location sparkniftireader.so
4) Spark version 1.5.2 was used, make sure dependencies involving Spark core in pom.xml are referring to spark 1.5.2 or above
5) inappropriate locale setting might cause error on build, export LC_CTYPE=C if needed

On successful build of biananes, a biananes.jar will result. Import and link the jar file when compile and run the source code containing biananes function calls.

When using the interactive spark-shell or submitting a standalone script with spark-submit, add the following flag:

    --jar path_to_biananes.jar

Notes on build and usage are documented in the README.md in the biananes folder uploaded to a svn repository in NICS for the project.

C. Remarks and Evaluation

Discrepancies in results using the same biananes update, i.e. segmentation error on large scale matrix input vs successful return on small scale matrix or tall skinny matrix input might worth investigation on whether it is problem with code implementation or it is that Intel DAAL does not support operations on matrices with huge amount of features.

It is worth noting that the default number of partitions used in biananes is 32, which essentially does not affect the implementation as data with less volumes will just idle the partition without force attempt to have calculation done on those extra partitions.

However, the dimension of Numeric Table instantiated does affect the implementation. Except the case where large scale matrices are concatenated into row-oriented Numeric Table which gives segmentation fault, segmentation error also results when the dimension of Numeric Table exceeds the dimension of data points in the Numeric Table, while a smaller Numeric Table will be constructed and computed without problem, e.g. for a test case with a data containing array size of 1*577, a HomogenNumericTable in the size of 1*578 would return SIGSEGV while a 1*576 table would return results without error.

III. Benchmark Tests

According to benchmark results released by Intel,\(^5\) on computation of Principal Component Analysis on Spark, DAAL shows a max of 7x performance boost compared with MLlib.

In the hope of verifying the performance difference and compare such with fMRI data analysis, tests were carried out using DAAL and MLlib on Spark respectively, with the use of the updated biananes as implemented in figure 3.

Figure 5 shows the computation time of DAAL and MLlib on Singular Value Decomposition. Note a volume of data contains a table of 480115 rows and 1 column. To achieve proper scaling effect, for computation with 1 node and 2 nodes, data with 2 volumes are used, and when more nodes are used, data with corresponding volumes are used, e.g. 8-volume data is used for 8-node computation. Also note that for single node computation, it was carried out in local mode while others were carried out in cluster mode.

\(^5\) Intel,
Fig. 7. update to DAAL sample code to read from NIfTI file instead of reading CSV files into HDFS into 3 steps during distributed computation. The first step is local computation of partial results from the distributed input Numeric Table in each local node, the partial will be kept in the local node for step 3 as well as sent to the master node for step 2; step 2 requires the master node to collect the partial results and split the tables across local nodes to finish up the computation and step 3 is to finish the remaining computation with partial results from step 1 and those acquired from step 2.

QR decomposition shares the same flow of implementation as that of SVD. The implementation scripts of both computations were not touched but remained in their default implementation during the test. While computations in DAAL are from sample codes which call steps to perform the computations, implementations in MLlib are simply the replicate of Scala commands in Spark-shell in Java codes.

Figure 8 shows the SVD code and figure 9 the QR for MLlib in Java.

```java
// Create JavaSparkContext that loads defaults from the system properties and the classpath and sets the name
SparkContext sc = new SparkContext(new SparkConf().setAppName("Spark OM"));
SparkContext jsc = JavaSparkContext.fromSparkContext(sc);
// Read from the distributed HDFS data set at a specified path
StringDataSource templateDataSource = new StringDataSource(context, "" );
DistributedMetadataFile dd = new DistributedMetadataFile("/Spark/OM/data", templateDataSource);
JavaPairRDD<BigInteger, HomogenNumericTable> dataHDD = dd.getPairRDD(xl); //
JavaPairRDD<BigInteger, HomogenNumericTable> dataHDD = JavaPairRDD.<BigInteger, HomogenNumericTable> newJavaPairRDD(xl, 
// NIfTItools.NIfTIImageReadMasked("/.../all/tct.nifti", 
// NIfTItools.NIfTIImageReadMasked("/.../all/tct.nifti", 
// NIfTItools.NIfTIImageReadMasked("/.../all/automask.nifti", sc));

// Compute the top n singular values and corresponding singular vectors.
<singularValues, <leftSingularMatrix, <rightSingularMatrix> result = mat.computeSVD(right, true, 1.0e-6);>
<singularValues, <leftSingularMatrix, <rightSingularMatrix> result = mat.computeSVD(right, true, 1.0e-6);>
Fig. 8. MLlib SVD computation in Java
```

```java
SparkConf conf = new SparkConf().setAppName("SampleSvdJava");
SparkContext sc = new SparkContext(conf);
// Create a RowMatrix from JavaRDD[column]s:
RowMatrix mat = new DenseRowMatrix(500000, 1000000, sp);
// Compute the top n singular values and corresponding singular vectors.
<singularValues, <leftSingularMatrix, <rightSingularMatrix> result = mat.computeSVD(right, true, 1.0e-6);>
<singularValues, <leftSingularMatrix, <rightSingularMatrix> result = mat.computeSVD(right, true, 1.0e-6);>
Fig. 9. MLlib QR computation in Java
```

An additional build script from DAAL has been amended to build the Java codes and run the Spark-submit command. Code snippet of the build script of SVD computation as in figure 10.

```java
# Build
javac -d ./build/ -sourcepath . ./SampleSvd.java ./SparkSvd.java
cd build/
# Creating jar
jar --create -vf sparkvd.jar NIfTI2DAAL_SampleSvd.sbt
```

1) Update BIANANES_JAR to locate the desired biaananes.jar
2) Update SampleSvd.java/SampleQR.java for DAAL or computeSVDmllib.java/tallSkinnyQRmllib.java for MLlib to locate the desired input data file
3) Update the corresponding build scripts to choose from local mode and cluster deployment
4) Load spark/1.5.2
5) Run the corresponding build scripts to build and run the codes

In the test, each and every configuration was run for 10 times. The times obtained are the averaged result from the 10 trials.

C. Remarks and Evaluation

Note that in the test, the structure of Numeric Table used with DAAL is in column orientation, while with MLlib, data are represented by the row in RowMatrix. The difference in structure resulted in difference in the results returned. Figure 11 shows results of SVD with DAAL and figure 12 shows that with MLlib. Despite the difference in U and V returned, the singular values are the same.

While the difference in results returned were neglected, the data used in the test, despite having matrices of around or over 500000 data points, are quite small in size, i.e. around 1MB for the 2-volume data. The computation time was small and trivial compared with the overhead of function setup in the Big Data environment, which could possibly explain the increase in time when the same data file was used in 1-node and 2-node test. With the above said, it is to expect that the comparison results here are too preliminary to serve as an accurate benchmark test on DAAL and MLlib on fMRI data.

Fig. 11. Returned result from SVD with DAAL

Fig. 12. Returned result from SVD with MLlib

Fig. 13. Correlation can be computed by updating return result from ResultId.covariance to ResultId.correlation

IV. FCMA

This section describes the last stage of the project, which is the attempt to migrate the Full Correlation Matrix Analysis toolbox by Princeton University\(^9\) to the Big Data framework Apache Spark with the use of Intel DAAL.

There is the following three stage pipeline in FCMA\(^10\) that this project aims to replicate on the Big Data framework.

1) Correlation computation by matrix-matrix multiplication
2) Normalization of resulted correlation matrix
3) Support Vector Machine to validate result

This FCMA pipeline realizes full correlation matrix analysis in a fairly short period of time, and accuracy can be improved as compared with single-variate analysis. It is originally a toolbox of algorithms and libraries, with its core implementation in parallel computing with MPI. As the nature of FCMA, which deals with a lot of large scale data, is ideal for computation under the Big Data framework, the goal of the project is to employ the concept of the toolbox under Spark.

So far as this project proceeds, development on stage 1 and 3 has started. The intuition was to use matrix-matrix multiplication provided in MKL to develop stage 1. However, DAAL encapsulated the use of MKL that the direct call of matrix-matrix multiplication was not possible.

A. Code Snippet

A workaround to the problem is to make use of Covariance computation. Covariance is provided in the sample codes from DAAL, and result Numeric Table supports the change from type Covariance to type Correlation. By updating the implementation code in figure 14, a multiplication of the input Numeric Table(matrix), with its own transpose is resulted.

V. CONCLUSION

This project is under the Computational Science for Undergraduate Research Experience programme(CSURE 2016), which is conducted by the Joint Institute for Computational Sciences and led by the University of Tennessee, Knoxville and the Oak Ridge National Laboratory. Due to the time constrain of the programme, this project has not finished.

This project has shown preliminary integration of Biananes and Intel DAAL, and in turn shown the start of fMRI data analysis under the Big Data framework.

Future development of the project shall reside with the revision of the biananes updates and most importantly, the review of Intel DAAL. More data are expected to come which would facilitate the testing and reviewing of codes. And the underlying implementation of algorithms and support of its
Numeric Table data structure are crucial to the incorporation of DAAL in fMRI data analysis. The remaining stage of the FCMA pipeline shall be continued to accomplish the project.

VI. Reference
2) Wang et al., Full correlation matrix analysis of fmri data