Tutorial of Frovedis Spark Interface

1. Introduction

This document is a tutorial of Frovedis Spark interface.

Frovedis is a MPI library that provides

- Matrix library using above API
- Machine learning algorithm library
- Dataframe for preprocessing

The Spark interface wraps these functionalities can make it possible to call from Spark. Since the library is optimized for SX-Aurora TSUBASA, you can utilize vector architecture without being aware of it. You can use it also on x86 servers.

It is implemented by using a server program. An MPI program with Frovedis functionalities (frovedis_server) is invoked and Spark communicates with it.

2. Environment setting

In this tutorial, we assume that Frovedis is installed from rpm. Please follow /opt/nec/nosupport/frovedis/getting_started.md. As described in the file, if you want to use frovedis_server on x86, please do:

```
$ source /opt/nec/nosupport/frovedis/x86/bin/x86env.sh
```

If you want to use vector engine (VE), please do:

```
$ source /opt/nec/nosupport/frovedis/ve/bin/veenv.sh
```

Main purpose of the script is to set environment variables like SCALA_HOME, SPARK_HOME, SPARK_SUBMIT_OPTIONS. It also switches `mpirun` to call (x86 or ve). If you did not source MPI set up script for VE, veenv.sh also source it internally.

Scala and Spark is installed together with the Frovedis rpm; above environment variables point to them. If you want to use your own Scala and/or Spark, please change these environment variables and/or x86env.sh/veenv.sh scripts. (The Spark interface can be seen as the application of Spark, so it does not depend on the Spark version ideally.) Java virtual machine (openjdk) should have been installed together with Frovedis rpm as its dependency.

In this tutorial, we assume sbt as the building tool of Spark program. If you did not install sbt, please install it by following https://www.scala-sbt.org/release/docs/Installing-sbt-on-Linux.html.

It should be like:

```
$ curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
$ sudo yum install sbt
```

If you are behind proxy, you would need to set up proxy. You can specify it to sbtopts, which is in /usr/share/sbt/conf. Add proxy information to the file like this:

```
-Dhttp.proxyHost="your.proxy.example.com"
-Dhttp.proxyPort="8080"
-Dhttps.proxyHost="your.proxy.example.com"
-Dhttps.proxyPort="8080"
```

Alternatively, you can use Zeppelin, which is a kind of notebook tool that can be used to edit Spark program from Web browser. A script that downloads and extracts Zeppelin is in /opt/nec/nosupport/frovedis/x86/opt/zeppelin/extract_zeppelin.sh

```
$ /opt/nec/nosupport/frovedis/x86/opt/zeppelin/extract_zeppelin.sh somewhere_in_your_homedir
```

This command downloads and extracts Zeppelin into the specified directory. You can start Zeppelin by calling

```
$ extracted_path/bin/zeppelin-daemon.sh start
```

By default, Zeppelin daemon is waiting connection at 8080 port; please access the 8080 port of the server with your Web browser. (Please make sure that firewall is configured to allow the access.)

To stop the daemon, you can just call

```
$ extracted_path/bin/zeppelin-daemon.sh stop
```

3. Simple example

Since you need to build the program, please copy the “src” directory to your home directory.

Please look at “src/tut3/”. It includes build.sbt file and src directory, which includes source file as src/main/scala/LR.scala.

You can run sbt by

```
$ sbt package
```

to build the package. Please note that the first call of sbt will take a long time (e.g. 10 min.) to download dependent files.

The build.sbt file is standard one; but it includes

```
unmanagedBase := file("/opt/nec/nosupport/frovedis/x86/lib/spark/")
```

to refer to the jar file that is provided by Frovedis.

It loads “breast cancer” data and run logistic regression on the data. To use Frovedis, you need to import FrovedisServer:

```
import com.nec.frovedis.Jexrpc.FrovedisServer
```

Then, import LogisticRegression in this case:

```
import com.nec.frovedis.mllib.classification.LogisticRegressionWithSGD
```

In the case of Spark, following package is imported instead (commented out):

```
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
```

Before using the logistic regression routine, you need to invoke frovedis_server:

```
FrovedisServer.initialize("mpirun -np 4 " + sys.env("FROVEDIS_SERVER"))
```

You need to specify the command to invoke the server as the argument of initialize. Since the server is an MPI program, `mpirun` is used here. The option `-np` is for specifying the number of MPI processes. Here, 4 processes will be used. You can use multiple cards (in the case of vector engine) and/or multiple servers by specifying command line option appropriately.

The last argument of `mpirun` is the binary to execute. Here, the path of the binary is obtained from the environment variable `FROVEDIS_SERVER`, which is set in x86env.sh or veenv.sh.

In the program, loaded data is parsed and given to LogisticRegressionWithSGD. The call is the same as Spark. Within the call, the data in Spark is sent to frovedis_server and the machine learning algorithm is executed there.

Created model is used to predict the label to check the accuracy.

After executing the machine learning algorithm, please shutdown the server:

```
FrovedisServer.shut_down()
```

As you can see, what you need to do is changing the importing module and add initialize / shutdown the server.

You can run the sample by

```
$ ${SPARK_HOME}/bin/spark-submit ${SPARK_SUBMIT_OPTIONS} \
 --master local[4] target/scala-2.11/tut_2.11-1.0.jar
```

Here, `${SPARK_HOME}` and `${SPARK_SUBMIT_OPTIONS}` are set by x86env.sh or veenv.sh. The contents of `${SPARK_SUBMIT_OPTIONS}` is

```
--driver-java-options \
"-Djava.library.path=/opt/nec/nosupport/frovedis/x86/lib" \
--jars /opt/nec/nosupport/frovedis/x86/lib/spark/frovedis_client.jar \
--conf spark.driver.memory=8g
```

The option –driver-java-options "-Djava.library.path…" is for loading .so file. The option –jars /opt/nec/… is for loading .jar file.

You would see a lot of log messages. To suppress them, please edit `${SPARK_HOME}/conf/log4j.properties` like

```
log4j.rootCategory=FATAL, console
```

Then, the execution will produce output like

```
Accuracy = 0.9104
```

If you modify the importing package to original Spark one (and remove Frovedis initialization and shut_down), you can run the same program using Spark. It should also produce similar result.

You can run the same program using spark-shell or Zeppelin. If you use them, please comment out `val sc = new SparkContext(conf)` line because SparkContext is already created on these platforms.

If you use spark-shell,

```
$ ${SPARK_HOME}/bin/spark-shell ${SPARK_SUBMIT_OPTIONS} --master local[4]
```

Then,

```
scala> :load src/main/scala/LR.scala
```

will load the file (or you can use :paste for pasting the contents). Then,

```
scala> LR.main(Array(""))
```

will run the program.

If you use Zeppelin, you can paste the program to the note. At the first line, you need to add "%spark" to use spark from Zeppelin. At the last line, please add "LR.main(Array(""))" like spark-shell case. Then, you can run the paragraph.

If it does not work, please make sure that environment variables are properly set by x86env.sh or veenv.sh *before* starting the zeppelin daemon.

The frovedis server will be terminated when the interpreter exits. If it is not terminated because of abnormal termination, please kill the server manually by calling command like `pkill mpi`.

4. Machine learning algorithms

At this moment, we support following algorithms:

- `mllib.classification.LogisticRegressionWithSGD`
- `mllib.classification.LogisticRegressionWithLBFGS`
- `mllib.classification.SVMWithSGD`
- `mllib.classification.SVMWithLBFGS`
- `mllib.classification.NaiveBayes`
- `mllib.regression.LinearRegressionWithSGD`
- `mllib.regression.LinearRegressionWithLBFGS`
- `mllib.regression.RidgeRegressionWithSGD`
- `mllib.regression.RidgerRegressionWithLBFGS`
- `mllib.regression.LassoWithSGD`
- `mllib.regression.LassoWithLBFGS`
- `mllib.clustering.KMeans`
- `mllib.tree.DecisionTree`
- `mllib.recommendation.ALS`

Please add `com.nec.frovedis.` to import these packages. (In the case of original Spark, `org.apache.spark.` is added to import them.) The interface is almost the same as Spark.

SVD and PCA are also supported. In this case, the interface is a bit different. You need to "import com.nec.frovedis.matrix.RowMatrixUtils._". Then,"RowMatrix" is extended to have

- `computePrincipalComponentsUsingFrovedis(k)`
- `computeSVDUsingFrovedis(k)`

methods. The result can be converted to Spark data using to_spark_result().

Other than Spark algorithms, we support following algorithms.

- `mllib.fm.FactorizationMachine`

You can use both dense and sparse matrix as the input of machine learning just like Spark. It is automatically sent to Frovedis server, and automatically distributed among MPI processes. (SX-Aurora TSUBASA shows much better performance with sparse matrix.)

For more information, please refer to the manual. You can also find other samples in /opt/nec/nosupport/frovedis/x86/foreign_if_demo/spark/.

5. Frovedis server side data and distributed matrix

As we mentioned, you can use variable of Spark side directly as the input of machine learning algorithms that works on Frovedis server. In addition, you can also use the Frovedis server side data explicitly, which can be used as input of the machine learning algorithms.

Since you can keep the data at Frovedis server side, you can reduce the communication cost of sending data from Spark to the server if you reuse the data.

Please look at "src/tut5-1/".

```
val data = MLUtils.loadLibSVMFile(...)
```

This creates `RDD[LabeledPoint]` where the matrix part of the LabeledPoint is sparse.

```
val fdata = new FrovedisLabeledPoint(data)
```

Here, `fdata` is Frovedis side LabeledPoint, which contains matrix and label vector. You can use this as the input of machine learning algorithm like LigisticRegressionWithSGD.

```
fdata.debug_print()
```

To check if it is really created, `debug_print()` is called. It should print like:

```
matrix:
num_row = 16, num_col = 7
node 0
local_num_row = 9, local_num_col = 7
val : 2 9 1 4 8 2 3 8.9 2 9 1 4 8 2 3 8.9 2 9
idx : 0 4 0 2 3 0 1 6 0 4 0 2 3 0 1 6 0 4
```

```
off : 0 2 3 5 8 10 11 13 16 18
node 1
local_num_row = 7, local_num_col = 7
val : 1 4 8 2 3 8.9 2 9 1 4 8 2 3 8.9
idx : 0 2 3 0 1 6 0 4 0 2 3 0 1 6
off : 0 1 3 6 8 9 11 14
dvector(size: 16):
 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
```

The sparse matrix part is represented as CRS format. It is printed at the server side. (Therefore, if you are using Zeppelin, it will not be shown.) It shows that first 9 rows are in the node 0 and other 7 rows are in the node 1. The label part is represented as the distributed vector dvector.

You need to explicitly release it by calling `fdata.release()` to avoid memory leakage.

```
val mat = data.map(_.features)
val fmat = new FrovedisSparseData(mat)
fmat.debug_print()
```

In this case, mat is RowMatrix where the contents is sparse matrix. The variable fmat is sparse matrix at Frovedis server. You can use it as the input of functions, like `SVD.compute(fmat, k)`.

The memory of the server side is released when the variable `fmat` is

Next “src/tut5-2/” is dense matrix version. In this case, `RDD[Vector]` is created by

```
val rdd_vec = sc.textFile("...mat_4x4"). \
                .map(x => Vectors.dense(x.split(' ').map(_.toDouble)))
```

It is used as the input of creating FrovedisRowmajorMatrix by

```
val rmat = new FrovedisRowmajorMatrix(rdd_vec)
```

To check if it is really created, `debug_print()` is called. It should print like:

```
matrix:
node = 0, local_num_row = 2, local_num_col = 4, val = 1 0 0 0 1 0 0 1
node = 1, local_num_row = 2, local_num_col = 4, val = 1 0 6 1 4 0 0 1
```

There are some operations like transpose() are defined on FrovedisRowmajorMatrix.

```
val t_rmat = rmat.transpose()
t_rmat.debug_print()
```

After calling transpose(), `debug_print()` shold print like:

```
matrix:
node = 0, local_num_row = 2, local_num_col = 4, val = 1 1 1 4 0 0 0 0
node = 1, local_num_row = 2, local_num_col = 4, val = 0 0 6 0 0 1 1 1
```

There are other format of matrix. FrovedisColmajorMatrix can be created like:

```
val cmat = new FrovedisColmajorMatrix(rdd_vec)
```

FrovedisBlockcyclicMatrix can be created like:

```
val bmat = new FrovedisBlockcyclicMatrix(rdd_vec)
```

FrovedisBlockcyclicMatrix supports distributed matrix operationos that is backed by ScaLAPACK/PBLAS. It can be utilized for large scale matrix operations. Please see "src/tut5-3/".

First, input RDD[Vector] is created by reading from a file. By using it, two FrovedisBrockcyclicMatrix is created, bmat1 and bmat2.

The contents is checked by calling `get_rowmajor_view()`, which prints the contents after converting it to row major matrix. Then,

```
val gemm_ret = PBLAS.gemm(bmat1,bmat2)
```

is called. It does matrix-matrix multiplication. The result is also newly created blockcyclic matrix. You can seve FrovedisBlockcyclicMatrix using `save`.

Please note that save is executed at the server side; the saving directory should be shared all MPI ranks of the frovedis_server (typically using NFS). If you are using Zeppelin for testing, please note that the saving directory is the current working directory of the zeppelin daemon.

`PBLAS.dot` calculates the dot product of two vectors. Here, vectors are the FrovedisBlockcyclicMatrix whose number of column is 1. Such vectors can be converted to Spark Vector by calling `to_spark_Vector()`, which is demonstrated here. In the case of Matrix, you can use `to_spark_Matrix()` or `to_spark_RowMatrix()`.

`PBLAS.scal` is multiplication by scalar. As you see, PBLAS interface overwrites the original matrix.

`PBLAS.nrm2` calculates L2 norm of a vector.

There are other PBLAS functions like swap, copy, axpy, gemv, ger, and geadd.

Next demonstration is loading FrovedisBlockcyclicMatrix from a file. This is also done at the server side, so the directory should be shared all MPI ranks of the frovedis_server.

Using the loaded matrix, `getrf` is called, which does LU factorization.

```
val rf = ScaLAPACK.getrf(bmat2)
```

The argument matrix is overwritten to factorized matrix. The return value contains pivoting information (ipiv), which is needed to use the factorized matrix later.

Next, by using the factorized matrix, inverse of the matrix is calculated using `getri`.

```
val stat = ScaLAPACK.getri(bmat2,rf.ipiv())
```

As mentioned, `rf.ipif()` is used as the input of `getri`. The result is overwritten to the argument matrix. The result is saved to a file. The result would be like:

```
0.882353 -0.117647 0.196078
0.176471 0.176471 0.0392157
0.0588235 0.0588235 -0.0980392
```

You can also use the result of LU factorization for solving the system of linear equation by using `getrs`, or directly solve it by using `gesv`. You can also calculate least squre by using `gels`.

Last example is singular value decomposition (SVD) by `gesvd`. Unlike `computeSVD` or `computeSVDUsingFrovedis`, it computes full SVD (it takes more time if you only need part of the SVD result).

```
val svd_ret = ScaLAPACK.gesvd(bmat2)
```

Calling `gesvd` creates an object that contains result. You can save it to the file, and load it from a a file. It can be converted to Spark's result by using `to_spark_result(sc)`.

6. DataFrame

In addition to machine learning algorithms, we also support DataFrame. Please see "src/tut6/".

First, Spark DataFrame `pdf1` and `pdf2` are created. Then, FrovedisDataframe is created from Spark DataFrame as `fdf1` and `fdf2`.

```
val peopleDF = sc.textFile(input + "/people.txt")
                 .map(_.split(","))
                 .map(attributes => (attributes(0).trim,
                                     attributes(1).trim.toInt,
                                     attributes(2).trim))
                 .toDF("EName","Age","Country")

val countryDF = sc.textFile(input + "/country.txt")
                 .map(_.split(","))
                 .map(attributes => (attributes(0).trim,
                                     attributes(1).trim))
                 .toDF("CCode","CName")

val df1 = new FrovedisDataFrame(peopleDF)
val df2 = new FrovedisDataFrame(countryDF)
```

To show the contents of FrovedisDataframe, you can use show():

```
fdf1.show()
fdf2.show()
```

They should produce output like:

```
EName   Age Country
Michael 29  USA
Andy    30  England
Tanaka  27  Japan
Raul    19  France
Yuta    31  Japan

CCode   CName
01  USA
02  England
03  Japan
04  France
```

To filter the rows, you can write like:

```
df1.filter($$"Age" > 25).show()
```

It should produce output like:

```
EName   Age Country
Michael 29  USA
Andy    30  England
Tanaka  27  Japan
Yuta    31  Japan
```

To sort the rows, you can write like:

```
df1.sort($$"Age").show()
```

It is sorted in descending order of Age. Output should be like:

```
EName   Age Country
Raul    19  France
Tanaka  27  Japan
Michael 29  USA
Andy    30  England
Yuta    31  Japan
```

Join can be done like this:

```
df1.join(df2, df1("Country") === df2("CName"),"outer","hash")
   .select("EName","Age","CCode","CName").show()
```

It produces output like: EName Age CCode CName Tanaka 27 03 Japan Raul 19 04 France Yuta 31 03 Japan Michael 29 01 USA Andy 30 02 England

To join tables, it is required that the column names are unique in the current implementation. If you want rename the column, you can do like this:

```
df1.withColumnRenamed("Country", "Cname")
```

You can chain operations. Here, join, select, filter, and sort are chained.

```
df1.join(df2, df1("Country") === df2("CName"),"outer","hash")
   .select("EName","Age","CCode","CName")
   .when($$"Age" > 19)
   .sort($$"CCode", $$"Age").show()
```

It produces output like:

```
EName   Age CCode   CName
Michael 29  01  USA
Andy    30  02  England
Tanaka  27  03  Japan
Yuta    31  03  Japan
```

You can get the statistics of the columns like min, max, sum, avg, std, and count by calling like `min("Age")`. Like Spark DataFrame, you can also call `describe()` to see all these information.

```
println("Total rows: " + df1.count())
df1.count("Age").foreach(println)
df1.min("Age").foreach(println)
df1.max("Age").foreach(println)
df1.sum("Age").foreach(println)
df1.avg("Age").foreach(println)
df1.std("Age").foreach(println)
df1.describe().show()
```

This prints like:

```
Total rows: 5
5
19
31
136
27.200000
4.816638
+-------+---------+
|summary|      Age|
+-------+---------+
|  count|        5|
|    sum|      136|
|   mean|27.200000|
| stddev| 4.816638|
|    min|       19|
|    max|       31|
+-------+---------+
```

Frovedis DataFrame can be converted to matrix. First, Spark DataFrame is created and converted to Frovedis DataFrame.

```
val sampleDF = sc.textFile(input + "/sample.txt")
                 .map(_.split(","))
                 .map(attributes => (attributes(0).trim.toInt,
                                     attributes(1).trim.toInt))
                 .toDF("A","B")
val df3 = new FrovedisDataFrame(sampleDF)
df3.show()
```

The DataFrame is:

```
A   B
1   35
2   40
5   60
2   30
1   100
```

You can create `FrovedisRowmajorMatrix` by specifying the columns. The columns should be integer or floating point values. In this case,

```
val rmat = df3.toFrovedisRowmajorMatrix(Array("A", "B"))
rmat.debug_print()
```

In this case, columns `A` and `B` are selected and converted to matrix. This produces

```
node = 0, local_num_row = 3, local_num_col = 2, val = 1 35 2 40 5 60
node = 1, local_num_row = 2, local_num_col = 2, val = 2 30 1 100
```

You can also create `FrovedisCorlmajormatrix` by `toFrovedisColmajorMatrix`.

Then, you can specify columns as category variable. In this case, it can be any data type; it is converted using on-hot encoding. In this case, the result becomes FrovedisCRSMatrix.

```
val (crsmat1,info) = df3.toFrovedisSparseData(Array("A", "B"),
                                              Array("A"), true)
crsmat1.debug_print()
```

Here, columns "A" and "B" is selected to create the matrix. The second argument is to specify which column is used as categorical variable. In this case column "A" is specified. If last argument is true, `info` data structure is also returned. It is used to create a matrix from FrovedisDataFrame next time (explained later).

The result of debug print is as follows:

```
matrix:
num_row = 5, num_col = 4
node 0
local_num_row = 3, local_num_col = 4
val : 1 35 1 40 1 60
idx : 0 3 1 3 2 3
off : 0 2 4 6
node 1
local_num_row = 2, local_num_col = 4
val : 1 30 1 100
idx : 1 3 0 3
off : 0 2 4
```

If it is shown as dense matrix, it should look like:

```
1 0 0 35
0 1 0 40
0 0 1 60
0 1 0 30
1 0 0 100
```

Here "1" is assigned to 0th column, "2" is assigned to 1st column, and "5" is assigned to 2nd column.

If you use this data for machine learning, you would want to convert other matrix using the same way for inference, for example. The `info` structure is used for this purpose.

```
val crsmat2 = df3.toFrovedisSparseData(info)
crsmat2.debug_print()
```

Since `info` contains the information needed for conversion, the result should be the same.

7. Manuals

Manuals are in `../manual` directory. In addition to PDF file, you can also use `man` command (MANPATH is set in x86env.sh or veenv.sh). For python interface, the section is `3s` (same name of the manual may exist in section `3` or `3p`.), so you can run like `man -s 3s logistic_regression`. Currently, there are following manual entries:

- `logistic_regression`
- `logistic_regression_model`
- `linear_regression`
- `linear_regression_model`
- `lasso_regression`
- `ridge_regression`
- `linear_svm`
- `svm_model`
- `kmeans`
- `kmeans_model`
- `als`
- `frovedis_sparse_data`
- `matrix_factorization_model`
- `blockcyclic_matrix`
- `scalapack_wrapper`
- `pblas_wrapper`
- `arpack_wrapper`
- `getrf_result`
- `gesvd_result`