MIPL: Mining Integrated Programming Language

The Tutorial

Project Manager: Younghoon Jeon < yj2231@columbia:edu >
System Architect: Young Hoon Jung < yj2244@columbia:edu >
Language Expert: Jinhyung Park < jp2105@columbia:edu >
System Integrator: Daniel Wonjoon Song < dws2127@columbia:edu >
System Validation: Akshai Sarma < as4107@columbia:edu >

Introduction

These tutorials explain the MIPL language from its basics up to the complex features, including basic concepts such as Fact, Rule, and Query (FRQ) or Job and advanced concepts such as parallel and accelerated execution of a Job. The tutorial is oriented in a practical way, with working example programs in all sections to start practicing each lesson right away.

0. Comments

MIPL uses shell script-like comments, which begin with the character *'#'* and terminate at the end of the line, separated by the newline character.
Undoubtedly, comments do not have any effect on the program's operation.

```
#     This line is a comment. The line below is an actual code.
difficult_to_read(X) <- too_many_comments(X).
###   This is also a comment. The number of '#' does not matter.
```

1. FRQ

The main purpose of MIPL is to retrieve meaningful information from large data with variable data mining algorithms or inference rules. In order to achieve this goal conveniently, the grammar of MIPL is based on the paradigm of logic programming languages, such as Prolog. Briefly speaking, the basic inference of MIPL is processed with fact-rule-query chain, as known as FRQ.

Here is an example of a simple FRQ.

```
cat(tom).   # fact
cat(john).  # fact

cat(tom)?   # query, we get true.
dog(tom)?   # query, we get false.

animal(X) <- cat(X).    # rule
```

```
animal(tom)?            # query, we get true.
animal(paul)?           # query, we get false.
```

1.1 Fact

In MIPL, a fact is knowledge that is the basis of inferences. To simplify, you can just assume that facts are values already considered to be true. Facts are defined in three ways: simple fact, complex fact, and dynamic fact.

1.1.1 Simple fact

A simple fact is defined by specifying an identifier where you consider the identifier to be true. That is, you can just write a simple fact like this.

```
raining.          # a simple fact
```

In the above example, we assigned a simple fact named *raining*. The last character '.' denotes the completion of the simple fact statement. We can simply check whether a fact is true or not with a query. Query statements end with '?'.

```
raining?          # this is true
sunny?            # this is false
```

1.1.2 Complex fact

Simple facts are not enough to represent meaningful ideas. A complex fact is used to define a relationship among one or more simple facts. The syntax for a complex fact is the following.

```
complex_fact (fact1, ...)
```

For example,

```
cat(tom).         # complex fact
cat(tom)?         # this is true
cat(X)?           # we get (X, tom)
```

One thing to note here is that MIPL distinguishes identifiers and variables. An identifier starts with a lowercase alphabet; whereas a variable starts with an uppercase alphabet. In the example above, *cat* and *tom* are identifiers, and *X* is a variable. If you use variables in your query, you get bindings of a variable and an identifier that makes the query true, or you get a false value if there are no such bindings. Here is another example.

```
weather(ny, cloudy).    # complex fact
weather(la, sunny).     # complex fact

weather(ny, cloudy)?    # true
weather(la, snowy)?     # false
weather(la, W)?         # we get (W, sunny)
```

1.1.3 Using complex facts to define a matrix

In MIPL, the most important type is a matrix. Instead of making the grammar complicated by adding a special syntax to define a matrix, MIPL uses a sequence of complex facts that have same the identifier to define a matrix. For example,

```
mat(1, 2, 3, 4).        # the first row
mat(2, 3, 4, 5).        # the second row
mat(3, 4, 5, 6).        # the third row
```

The complex facts in the example above define the following 4x3 matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix}$$

1.1.4 Dynamic fact

As mentioned in the last section, the most important type in MIPL is a matrix. However, it is difficult and ineffective to allow only the method above to define a matrix. Thus, MIPL supports a special method to generate matrices by invoking a procedure. MIPL uses the term 'dynamic fact' for this method. You have to be careful not to be confused because the grammar for the dynamic fact is quite similar with the grammar of the rule, which will be covered below. The syntax for the dynamic fact is the following.

```
[target1, …] <- job_name(arg1, …).
```

Job is a feature similar to function or procedure in other programming languages. Details on job are covered later. Job in MIPL returns one or more matrices, and the statement above assigns the matrices returned from the job to the target facts. The targets above are not variables. They are facts. This can be a little confusing, so let's see the next example. There are two predefined fact matrices, *a* and *b*, and we want to make a new matrix by adding them. We can do this like this.

```
# job definition for adding matrices
job sum (A, B) {
      @(A + B).
}

# matrix a
a(1, 1).
a(1, 1).

#matrix b
b(1, 0).
b(0, 1).

[c] <- sum(a, b). # get c from a and b
```

The matrix c generated above can be also defined with complex fact like this.

```
c(2, 1).
c(1, 2).
```

1.2 Rule

Rules are regulations that produce new facts from existing facts. We can use rules like this.

```
new_fact(X, Y) <- old_fact(X, Y).   # new_fact is true if old_fact is true.
```

When we define a rule, we can also use ':-' instead of '<-'. These two are identical. MIPL supports ':-' to make MIPL compatible with Prolog.

1.3 Query

As seen above, we use queries when we want to know which fact is true or want to retrieve values of variables that make a fact true. Queries end with '?'.

We also showed how to use simple queries when we introduced fact and rule. Let's take a look at a special type of query here.

'_' is used to find out if there is any matching identifier. For example,

```
cat(tom).
cat(_)?           # true because we have tom that makes cat(tom) true.
```

We can also use wildcards for queries. * is used to denote one or more variables. For example,

```
say(hello, world).
# we know fact say has two parameters.
say(X, Y)?              # we get (X, hello), (Y, world)
# we do not know how many parameters say has, but we can use a wildcard.
say(*)?                 # we get * = (hello, world)
```

We can use a regular expression for queries, which is an advanced syntax. Here is an example.

```
cat(tom).
cat(john).

^.a.$(X)?
# cat(X), X = tom.
# cat(X), X = john.
```

2. Job

Job is a concept which is similar to functions in C and Java.

Job definition requires ‘job’ keyword that indicates the following is a job definition, job name, parameter list, and job body. Job name should be an identifier. Parameters are a list of variables, concatenated with “,”. Job body is a sequence of statements that are surrounded with braces.

2.1 Job definition

Here is a typical definition of job.

```
job myJob1(A, B, C) {
  # statements here...
}
```

Parameters can be empty.

```
job myJob2() {
  # statements here...
}
```

As stated above, MIPL uses different type of strings for identifiers and variables. Job names should be identifiers, and parameters should be variables. That is, the job definition below will cause compile errors.

```
job MyJob(A, B, C) {
}
```

or

```
job myJob(a, b, c) {
}
```

The braces that form job body are mandatory.

```
# This is not possible.
job myJob(A, B, C)
  @(A + B + C).
```

2.2 Statements

Statements are instructions that do something useful with jobs. Each statement ends with ‘.’.

2.2.1 Statement block

Several statements can form a statement block. We can make a statement block simply by embracing statements with braces.

2.2.2 Return statement

To return a variable from the current job, you can just write this.

```
@Var.
```

‘@’ is a keyword specifying return. Using ‘@’ rather than ‘return’ makes users sure that they are currently working on generating "target" within this "job", rather than producing return "value" from this "function". Unlike return statement in general functions or procedures in other programming languages, return statement is MIPL does not terminate the current job. Jobs in MIPL do not terminate until the control reaches the end of the job body. Therefore, MIPL job can return more than one time before it ends, and this is a significant feature of MIPL, which is useful when MIPL is used on distributed computation environments such as map-reduce. Here is an example.

```
# a Job that returns two or three values
job multi_return(A, B, C) {
  if (A == 1) {
    @(B + C).
  }
  @C.
  @B.
}

x(10, 20).
y(30, 40).

[sum, c, b] <- multi_return(1, x, y).
# sum <- b+c, b <- B, c <- C

sum(*)?
# *0 = 40, *1 = 60
c(*)?
# *0 = 30, *1 = 40
b(*)?
# *0 = 10, *1 = 20

[c, b] <- multi_return(0, x, y).
# b <- B, c <- C
```

2.2.3 Conditional statement

MIPL supports if-else statements, which are similar to those in C.

```
if (condition) statements
if (condition) statements else statements
```

This is an example that returns an absolute value of a variable.

```
abs(V) {
  if (V >= 0)
    @V.
```

```
  else
    @(-V).
}
```

We can combine several if-else statements.

```
if (condition1)
      # do something…
else if (condition2)
      # do something…
else if (condition3)
      # do something…
else
      # do something…
```

For the conditions, we can only include expressions that can be evaluated as boolean value. That is, the conditions should have only logical expressions (‘&&’ and ‘||’) and comparison expression (‘<’, ‘<=’, ‘>’, ‘>=’, ‘==’, ‘!=’). The following are not allowed.

```
if ((Var = job()) …

Var = 1
if (Var) …
```

2.2.4 Loop statement

MIPL has two kinds of loop statements: while and do-while. The syntax and operation of these statements are similar to those in C or Java.

```
do
      statement or statement block
while (condition)

while (condition)
      statement or statement block
```

For example, we can use them as the following.

```
I = 1.
Sum = 0.
while (I <= 10) {
      Sum += I.
      I += 1.
}
```

Note that MIPL does not have for loop.

2.3 Expression

2.3.1 Assignment expression

Variables can be assigned values using '='. The operator '<-' also has the same meaning with '='. '+=', '-=', '*=', '-=', and '%=' are also available in MIPL.

2.3.2 Arithmetic expression

MIPL provides arithmetic operators used in general programming languages. We can use binary operators [+ - * / %], and unary operators [+ -]. MIPL do not have '++' and '--'. The evaluation of an arithmetic expression depends on the type of its operands. Also, there are cases in which operation is not allowed depending on the status of the value included in the variables. For example, the expression $X = A * B$ will cause a runtime error if A and B are matrices and the number of rows in A and the number of columns in B are not the same. Please refer to the reference manual for more detail.

2.3.3 Comparison expression

We can use '=', '!=', '<', '<=', '>', '>=' for comparison. The evaluation of the comparison expression can be only used in conditional statements, and are not allowed to be assigned to variables. For example, the following is not possible.

```
Cmp = Var1 < Var2.
if (Cmp)
  …
```

The above should be like this.

```
if (Var1 < Var2)
  …
```

2.3.4 Logical expression

MIPL supports '&&' (logical and) and '||' (logical or). As with comparison expression, they can only be use in conditional statements. There is no logical not in MIPL.

2.3.5 Matrix access expression

MIPL provides a way to access one or more elements in a matrix, with '[]' operator. In this case, a matrix is considered to be a 2x2 matrix. For example,

```
# This assigns the (1, 2) element of the matrix G to E.
E = G[1][2].
```

We may want to retrieve the subset of a matrix, rather than an element. In this case, we use '~' to specify the subset of a matrix. For example,

```
# GA is the 0th row of matrix G.
GA = G[0][~].
```

```
# GB is the rest of G, after excluding the 0th row.
GB = G[1~][~].
```

2.3.6 Job invoking expression

In a job, it is also possible to invoke another job and save the return value in a variable. It is easy to use. It is same as calling functions.

```
Var = job(arg1, ..).
```

3. I/O

3.1 Matrix load and save

MIPL provides two built-in Jobs, load() and save(), which loads a matrix from a file to a matrix and saves a matrix to a file, respectively. However, the matrix is not actually loaded into memory unless it is accessed by the local machine. For example,

```
job addJob(A, B) {
 @A + B.
}


[a] <- load("very_large1.dat").
[c] <- addJob(a, load("very_large2.dat")).

[] <- save(c, "very_large_result.dat").
```

Above example shows that there is no actual matrix loading in the local machine, provided the matrices are sufficiently larger than the threshold for parallel computation. The file is directly transferred to the Hadoop File System (HDFS) and accessed by MapReduce clusters. After the computation the result file will be transferred to the local machine without loading into the local machine's memory. However, if a line like the following is added after the line that loads the file "very_large1.dat", the matrix will be loaded into local memory to be read from it.

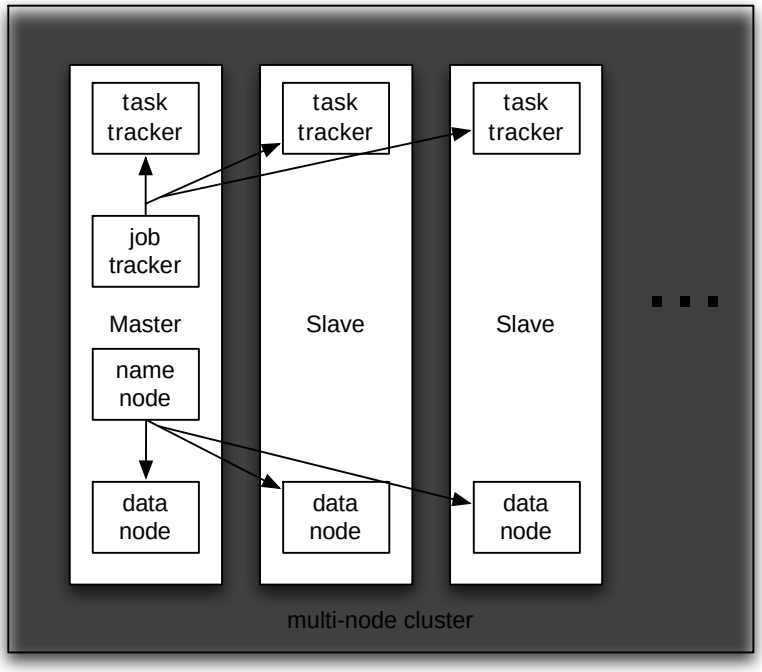
```
[a] <- load("very_large1.dat").
a(*)?
```

Appendix A. Hadoop Setting for parallel executions

Cluster Overview

We will build a multi-node cluster using several single-node clusters. We will set up Hadoop for each of the node and will merge those nodes into a multi-node cluster. Our Hadoop Architecture has one master node and several slave nodes in one multi-node cluster.

The master node will run the “master” daemons for each layer: NameNode for the HDFS storage layer, and JobTracker for the MapReduce processing layer. The master and other slave nodes will run the “slave” daemons: DataNode for the HDFS layer, and TaskTracker for MapReduce processing layer. Basically, the “master” daemons are responsible for coordination and management of the “slave” daemons while the latter will do the actual data storage and data processing work.



Hadoop Configuration Step

1. First we need to add a dedicated Hadoop system user to each node.
2. Setup SSH access so that the master can connect to slaves without user.
3. Specify master and slave node using “conf/masters” and “conf/slaves”.
4. Configure NameNode at “conf/core-site.xml”, JobTracker at “conf/mapred-site.xml”, and dfs.replication at “conf/hdfs-site.xml”.
5. Format the HDFS file system via the NameNode.

Reference

http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/
http://hadoop.apache.org/common/docs/current/cluster_setup.html