Universidade de Santiago de Compostela

Perldoop v0.6.3

User Manual

José M. Abuín Mosquera

Centro de Investigación en Tecnoloxías da Información (CiTIUS)

November 17, 2014

Contents

List of Figures

1 Introduction

Perldoop [1] is a new tool developed by researchers from University of Santiago de Compostela (Spain) as part of the project "High Performance Computing for Natural Language Processing – HPCNLP". This tool automatically translates Hadoop-ready Perl scripts into its Java counterparts, which can be directly executed on a Hadoop cluster while improving their performance significantly.

Hadoop provides an utility to execute applications written in languages different from Java, known as Hadoop Streaming. To use this tool the only requirement is that applications should read from stdin and write to stdout. Even though Hadoop Streaming is a very useful tool, important degradations in the performance were detected using Hadoop Streaming with respect to Hadoop Java codes [2]. Only for computational intensive jobs whose input/output size is small, the performance of Hadoop Streaming is sometimes better because of using a more efficient programming language

Therefore, the best choice in terms of performance is to develop Hadoop applications using Java. Nevertheless, translating Perl codes into Java is a long and tedious task, especially when the applications consist of many regular expressions. For this reason, we developed the Perldoop tool, which allows to automate the translation process increasing the performance and efficiency as well as the productivity.

The general case of automatically translating an arbitrary Perl code into its Java equivalent is a very hard problem, due to the characteristics of both languages. Note that our objective in this work was not to develop a powerful tool that allows to automatically translate any existent Perl code to Java, but a simple and easy-to-use tool that takes as input Perl codes written for Hadoop Streaming, and produces Hadoop-ready Java codes.

2 Perldoop

Perldoop requires a well defined files structure (see Figure 1). In particular, two files are needed to generate the new Java code:

1. First, as is clear, the file containing **the Perl script to translate**. The Perl code should:
   - Follow strictly several programming rules (see Section 2.2).
   - Be correctly labeled (see Section 2.3).
2. In addition, a **Java template** is required to assure the correctness of the Java codes automatically generated by Perldoop (Section 2.1). The template should include the definition and constructor of the class, as well as the imports. Optionally, the programmer could also include additional code. We must highlight that the code in the template depends on the considered Perl code to translate.

Once these two files are available, Perl source code and Java template, Perldoop is ready to start the translation process.

2.1 Java Template

The creation of the templates is a simple process. Using templates is mandatory because some Java code cannot be automatically translated from the Perl code, and this code is necessary to the correct execution of the translated Java application. Some examples of this type of code are:

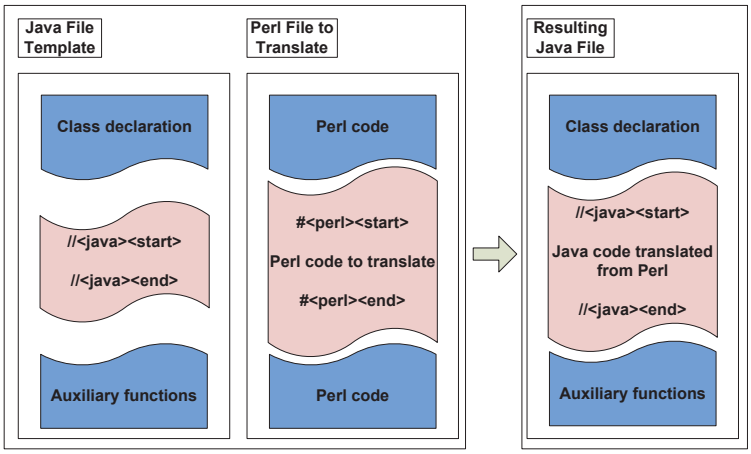

Figure 1: Structure of the files required by PERLDOOP.

- Imports, such as the Apache Hadoop libraries.
- Declaration and constructor of the class that the programmer is implementing.
- *try-catch* blocks required by Java.

Figure 2 shows an example of a Java template. In particular, the class is a *Mapper* of Hadoop. The template also includes the header of the *map* function and the *try - catch* block. The Java code generated by PERLDOOP will be inserted between the labels `<java><start>` and `<java><end>`. It can be seen that after the end label there is additional Java code.

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public static class WordCountMap extends Mapper<Object, Text, Text, Text>{
  @Override
  public void map(Object incomingKey, Text value,
  Context context) throws IOException, InterruptedException {
    try{

      //<java><start>

      //<java><end>

    }
    catch(Exception e){
      System.out.println(e.toString());
    }
  }
}
```

Figure 2: Example of a template for the Mapper of the WordCount application.

2.2 Programming rules

In order to assure the correctness of the Java codes automatically generated by PERLDOOP there are a few simple programming rules that the developers should follow when implementing Perl codes. These rules are:

1. Use always ordered `if` blocks:

```
if ($variable) {
$num = $num2 + 1;
}
```

   instead of:

```
$num = $num2 + 1 if($variable);
```

2. Boolean variables do not exist in Perl, basically they are integers. In this way, a programmer may find in the Perl code:

```
my $found = 1;
if($found) {
```

   This expression returns *true* when the variable is different than zero. However, boolean variables exist in Java. Therefore, in case that any variable in the Perl code is used as boolean, the programmer should label it (see how to label in Section 2.3):

```
my $found=1;     #<perl><var><boolean>
if($found) {
```

3. Perl allows to include variables in a text string. For example:

```
$variable = "$var1␣$var2";
```

   Instead of that, programmers should perform string concatenations with the "." operator:

```
$variable = $var1."␣";
$variable = $variable.$var2;
```

4. Restrict the access to array positions not previously allocated including restrictions in the conditional expressions:

```
elsif ($i<($#tokens-3) && ($tokens[$i+4] == 7)) {
```

5. Use different names for different variables, although Perl allows to use the same name for variables of different type. For example:

```
# Non-Permitted:
my $feat;
my @feat;

# Permitted:
my $feat1;
my @feat2;
```

6. The next condition in Perl:

```
if($adiantar) {
```

returns *true* if `adiantar` is different than zero. If the variable is an integer (not a boolean, see rule 2), the Perl code should check exactly that. That is:

```
if($adiantar!=0) {
```

7. When cheking that a string exists, the Perl code:

```
if(!$tags{tokens[$i]}) {
```

must be replaced by:

```
if(!$tags{$tokens[$i]} || $tags{$tokens[$i]} == "") {
```

2.3 Labeling the Perl code

The Perl code to translate should be labeled. This is because the type of the variables in Perl is not defined until a value is assigned to them. However, the type of a variable in Java must be specified once is declared. In addition, labels help in some tasks of the translation process as, for example, recognizing if the Perl code is a *mapper* or a *reducer*. Figure 3 displays the labeled Perl code of the *mapper* of the WordCount application, while Figure 4 shows the Java code generated by PERLDOOP from that Perl code. Note that labels in the Java template begin with `//`, while in Perl begin with `#`.

Next the existent labels are detailed:

Java Template

- `<java><start>`: The Java code generated by PERLDOOP will be inserted in the template after this label.

```
#!/usr/bin/perl -w

#<perl><start>

my $line;            #<perl><var><string>
my @words;           #<perl><array><string>
my $key;             #<perl><var><string>
my $valueNum = "1"; #<perl><var><string>
my $val;             #<perl><var><string>


while ($line = <STDIN>) { #<perl><map>
    chomp ($line);
    @words = split ("␣",$line);
    foreach my $w (@words) {    #<perl><var><string>
        $key = $w."\t";         #<perl><var><key>
        $val = $valueNum."\n";  #<perl><var><value>

        print $key.$val;
    }
} #<perl><ignoreline>

#<perl><end>
```

Figure 3: Perl code of the Mapper of the WordCount application.

- `<java><end>`: End of the automatically generated code.

Perl Code

- `<perl><start>`: From this point begins the translation.
- `<perl><end>`: End of the Perl code to be translated.
- `<perl><header><start>`: Start of the header. It will be used in future versions of Perldoop.
- `<perl><header><end>`: End of the header. It will be used in future versions of Perldoop.
- `<perl><var><boolean>`: A boolean variable is declared or accessed. Example:

```
my $found=0; #<perl><var><boolean>
$found = 1;  #<perl><var><boolean>
```

Its translation is:

```
boolean found = false;
found = true;
```

- `<perl><var><string>`: A string variable is declared.
- `<perl><var><integer>`: An integer variable is declared.

```
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public static class WordCountMap extends Mapper<Object, Text, Text, Text>{
  @Override
  public void map(Object incomingKey, Text value,
  Context context) throws IOException, InterruptedException {
    try{

      //<java><start>
      String line;
      String[] words;
      String key;
      String valueNum  = "1";
      String val;
      line  = value.toString();
      line = line.trim();
      words  = line.split("␣");
      for (String w : words) {

        key = w;
        val = valueNum;
        context.write(new Text(key),new Text(val));
      }
      //<java><end>
    }
    catch(Exception e){
      System.out.println(e.toString());
    }
  }
}
```

Figure 4: Java code of the Mapper automatically generated by PERLDOOP from the Perl code of Figure 3.

- `<perl><var><double>`: A double float variable is declared.
- `<perl><var><long>`: A long variable is declared.
- `<perl><array><string>`: The array declared contains strings.
- `<perl><array><boolean>`: The array declared contains booleans.
- `<perl><var><string><null>`: A string variable is declared, and it is initialized to *null*.
- `<perl><arraylist><string>`: ArrayList declaration of string type.
- `<perl><map>`: It is used to label a line of code containing a *while* loop reading from `STDIN` to indicate that the code is reading the input values of a *mapper*.
- `<perl><reduce>`: It is used to label a line of code containing a *while* loop reading from `STDIN` to indicate that the code is reading the input values of a *reducer*.

- `<perl><ignoreline>`: Ignore this line in the translation process.
- `<var><key>`: Hadoop type. It indicates that the variable is the *key* of a *key* - *value* pair. See example in Figure 3.
- `<var><value>`: Hadoop type. It indicates that the variable is the *value* of a *key* - *value* pair.
- `<cast><string>`: Cast to string. For example:

```
$returnValue = $count;    #<perl><cast><string>
```

  Translation:

```
returnValue  = String.valueOf(count);
```

- `<cast><int>`: Cast to integer. For example:

```
$returnValue = $count;    #<perl><cast><int>
```

  Translation:

```
returnValue  = Integer.parseInt(count);
```

3 How to use Perldoop

After downloading the Perldoop source code, the user will find three directories containing:

- `src/`
  Source code of Perldoop implemented using Python.
- `examples/`
  Simple examples to illustrate the use of Perldoop. In the current version it includes **HelloWorld** and **WordCount** applications written in Perl.
- `applications/`
  More complex Perl applications. The current version includes three natural language processing modules written in Perl. In particular, the modules process plain text to perform the following tasks: **Named Entity Recognition** (NER), **Part-of-Speech Tagging** and **Named Entity Classification** (NEC). All the modules process text in Spanish language.

The main file of the tool is `Perldoop.py`, located in `src/`, which has two input parameters:

1. The route to the Perl script to translate. The Perl code should be labeled and programmed following the rules detailed in Sections 2.2 and 2.3 respectively.
2. The route to the Java template where the translated code will be inserted.

Therefore, the correct syntax to execute PERLDOOP is:

```
python Perldoop.py [perl-file] [java-template-file] > [output-java-file]
```

Note that PERLDOOP prints to the standard output the translated Java file.

For example, if the user decides to translate the *mapper* of the Perl WordCount application, the command would be:

```
python Perldoop.py ../examples/WordCount/Perl/WordCountMap.pl
../examples/WordCount/JavaTemplates/WordCountMap.java >
../examples/WordCount/JavaTranslatedCode/WordCountMap.java
```

Inside the WordCount directory (`/examples/WordCount`), `WordcountCompile.sh` performs the automatic translation of the whole application using PERLDOOP, an also compiles the generated Java codes. The resulting `jar` file will be written in
`examples/WordCount/JavaProgram/`, which is ready to be executed on a Hadoop cluster.

3.1 Natural Language Processing (NLP) Modules

Perl is well-known for its unrivaled ability to process text using very powerful features such as regular expressions. The native Java support for regular expressions is not as good as the provided by Perl. For this reason, in order to improve the handle of regular expressions in Java, PERLDOOP takes advantage of the *jregex* library [3].

We have included three more complex Perl applications to explain the benefits of using Perldoop. NLP applications are located in `applications/NLP/`, and their codes contain hundreds of regular expressions in order to process plain text. The modules must be executed in a particular order because the output of one is the input of the next one. As we have commente before, there are three modules that perform the next tasks:

1. Named Entity Recognition (NER)
2. PoS Tagger
3. Named Entity Classification (NEC)

A brief explanation of the different modules can be found in [1]. A performance comparison between the original Perl codes using Hadoop Streaming and the Java ones with Hadoop is also detailed in the paper. Performance results show that Java codes generated using Perldoop execute up to 12× faster than the original Perl modules.

To translate the Tagger module, for example, the command would be the following:

```
python Perldoop.py ../applications/NLP/Perl/Modules/tagger-es.perl
../applications/NLP/JavaTemplates/Tagger.java >
../applications/NLP/JavaTranslatedCode/WordCountMap.java
```

Inside `applications/NLP/`, `NLP-Compile.sh` translates the Perl modules using PERLDOOP and compiles the Java applications to execute them on a Hadoop cluster. The script contains the commands to use the *jregex* library in the compilation of the Java codes. The resulting *jar* file executes all the modules in order, that is, NER, Tagger and NEC. In case that the user wants to select some of the modules, the Java template (`applications/NLP/JavaTemplates/Prolnat.java`) should be modified. For example, to deselect the NEC module the next lines must be commented out:

```
    ner = modNer.runNer(splits);

    splits.clear();

    tagger = modTagger.runTagger(ner);

    ner.clear();

    for(i = 0; i<nec.size();i++){
          context.write(NullWritable.get(), new Text(tagger.get(i)));
        }

    tagger.clear();

    /*
    nec = modNec.nec_es(tagger);

    tagger.clear();

    for(i = 0; i<nec.size();i++){
      context.write(NullWritable.get(), new Text(nec.get(i)));
    }

    nec.clear();
    */
```

In the same folder where is the *jar* file, the user can find the `Execute.sh` script. It contains an example of how to execute the *jar* file on a Hadoop cluster. Note that the NLP modules require as argument the `Diccionarios.zip` file, located in the same directory. This is a simple example of how to execute the NLP modules using Hadoop:

```
cd applications/NLP/JavaProgram/

hdfs dfs -copyFromLocal ../Text/InputText.txt InputText.txt

hadoop jar Prolnat.jar -archives ./Diccionarios.zip
InputText.txt outputInputText
```

3.2 Debug Mode

PERLDOOP also includes a Debug mode to perform tests. To enter this mode it is only necessary execute the next command inside `src/`:

```
python Perldoop.py
```

Next the user can introduce a line of Perl code. After pressing *Intro* the result will be the Perldoop translation of that line. We must highlight that information about the function calls is also shown.

```
python Perldoop.py

$$$$ Perldoop> my $variable = "String␣content"; #<perl><var><string>

Function procesaLinha :: my $variable = "String␣content";
#<perl><var><string>

Processing expression with comment :: <var><string>

Function procesaLinha :: my $variable = "String␣content";

Function procesaLinha :: Procesando asignacion a variable :: variable

Function procesaVariable. Valor a procesar :: variable

Function procesaVariable:: expresionVariable :: variable

Function procesaOperacion :: expresionString "String␣content"

String variable  = "String␣content";

$$$$ Perldoop>
```

3.3 Config file `config.txt`

In `src/` there is a file named `config.txt`, which allows the user to specify or modify the behavior of Perldoop. In the current version only two options are available:

- `hadoop=[true|false]`
- `sequential=[true|false]`

Note that if `hadoop` is *true*, `sequential` must be *false*, and vice versa. These options define how to translate the Perl function `print`. The translation is different if we are translating from Perl to sequential Java, or if the translation is for Hadoop. For instance, the next line of Perl code:

```
print $key.$val;
```

- if `Sequential = true`, the output is:

```
System.out.print(key+val);
```

- if Hadoop = true:

```
context.write(new Text(key),new Text(val));
```

References

[1] José M. Abuín, Juan C. Pichel, Tomás F. Pena, Pablo Gamallo, and Marcos García. Perldoop: Efficient execution of perl scripts on hadoop clusters. In *IEEE International Conference on Big Data*, pages 766–771, 2014.

[2] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: the performance evaluation of Hadoop streaming. In *ACM Symp. on Research in Applied Computation*, pages 307–313, 2011.

[3] JRegex, Regular Expressions for Java. `http://jregex.sourceforge.net/`. [Online; accessed July, 2014].