SeqPig Manual

November 8, 2012

Contents

1 Introduction

SeqPig is a bioinformatics library for the Apache Pig language for distributed analysis of large datasets. It provides import and export functions for file formats commonly used in bioinformatics, as well as a collection of Pig user-defined-functions (UDF's) that allow for processing of aligned and unaligned sequence data. Currently SeqPig supports BAM/SAM, FastQ and Qseq input and output. It is built on top of the Hadoop-BAM library. Note that this document refers to the latest version inside the MASTER git branch. This document can also be found under `http://seqpig.sourceforge.net/`.

Contact information: `mailto:seqpig-users@lists.sourceforge.net`

Releases of SeqPig come bundled with Picard/Samtools, which were developed at the Wellcome Trust Sanger Institute, and Biodoop/Seal, which were developed at the Center for Advanced Studies, Research and Development in Sardinia. See

`http://samtools.sourceforge.net/` and `http://biodoop-seal.sourceforge.net/`

For more examples of SepPig scripts see also the wiki of two past COST hackathons:

`http://seqahead.cs.tu-dortmund.de/meetings:fastqpigscripting`

`http://seqahead.cs.tu-dortmund.de/meetings:2012-05-hackathon:pileuptask`

`http://seqahead.cs.tu-dortmund.de/meetings:2012-05-hackathon:seqpig_life_savers_page`

2 Installation

2.1 Dependencies

Install Hadoop (tested with Hadoop 0.20.2) and Pig. Note that some of the example scripts require the latest release of Pig (currently 0.10.0) to be installed.

2.2 Environment variables

1. Set `HADOOP_HOME` and `PIG_HOME` to the installation directories of Hadoop and Pig, respectively, and `SEQPIG_HOME` to the installation directory of SeqPig. On a Cloudera Hadoop installation with a local installation of the most recent Pig release, this would be done for example by

```
export HADOOP_HOME=/usr/lib/hadoop
export PIG_HOME=/root/pig-0.10.0
export SEQPIG_HOME=/root/seqpig
```

2. To make life simpler also add the shell script directory to your `PATH`:

```
$ export PATH=${PATH}:${SEQPIG_HOME}/bin
```

3. Finally, for convenience add the following line to your `.bashrc`:

```
$ alias pig='${PIG_HOME}/bin/pig␣-Dpig.additional.jars=${SEQPIG_HOME}/lib/hadoop-bam
    -5.0.jar:${SEQPIG_HOME}/build/jar/SeqPig.jar:${SEQPIG_HOME}/lib/seal.jar:${
    SEQPIG_HOME}/lib/picard-1.76.jar:${SEQPIG_HOME}/lib/sam-1.76.jar␣-Dudf.import.list=
    fi.aalto.seqpig'
```

Note that some of the example scripts below (e.g., Section 3.2.2) require functions from *PiggyBank*, which is a collection of publicly available User-Defined Functions (UDF's) that are distributed with Pig but need to be built separately. For more details see `https://cwiki.apache.org/confluence/display/PIG/PiggyBank`. If you would like to make use of these you should also add `piggybank.jar` to the library path, which means the statement above becomes

```
$ alias pig='${PIG_HOME}/bin/pig␣-Dpig.additional.jars=${SEQPIG_HOME}/lib/hadoop-bam
    -5.0.jar:${SEQPIG_HOME}/build/jar/SeqPig.jar:${SEQPIG_HOME}/lib/seal.jar:${
    SEQPIG_HOME}/lib/picard-1.76.jar:${SEQPIG_HOME}/lib/sam-1.76.jar:${PIG_HOME}/contrib
    /piggybank/java/piggybank.jar␣-Dudf.import.list=fi.aalto.seqpig'
```

2.3 Instructions for building SeqPig.jar

1. Download hadoop-bam from `https://sourceforge.net/projects/hadoop-bam/`.
2. Download and compile the latest biodoop/seal git master version from `http://biodoop-seal.sourceforge.net/`. Note that this requires setting `HADOOP_BAM` to the installation directory of hadoop-bam.
3. Inside the cloned git repository (`$SEQPIG_HOME`), create a `lib/` subdirectory and copy the following jar files contained in the hadoop-bam release to this location:

   ```
   seal.jar   hadoop-bam-${HADOOP_BAM_VERSION}.jar sam-${SAM_VERSION}.jar picard-${
       SAM_VERSION}.jar
   ```

   Note: the Picard and Sam jar files are contained in the hadoop-bam release for convenience.
4. Run `ant` to build `SeqPig.jar`.

2.4 Usage

2.4.1 Pig grunt shell for interactive operations

Assuming that all the environment variables have been set correctly, it suffices to start the grunt shell via

```
$ pig
```

2.4.2 Starting scripts from the command line for non-interactive use

Alternatively to using the Pig grunt shell (which can lead to delays due to Hadoop queuing and execution delays), users can write scripts that are then submitted to Pig/Hadoop for execution. This type of execution has the advantage of being able to handle parameters, for example for input and output files. See `/scripts` inside the seqpig directory and the examples below.

3 Examples

SeqPig is based on Hadoop-BAM, which among other things provides IO access to BAM files from within Hadoop. Hadoop-BAM provides access to the data on the bases of SAMRecords (see `http://picard.sourceforge.net/javadoc/net/sf/samtools/SAMRecord.html`), which is the Picard type for a (typically aligned) read. As such, all fields and optional attributes become accessible from within Pig.

3.1 Operations on BAM files:

All examples assume that an input BAM file is initially imported to HDFS via

```
prepareBamInput.sh input.bam
```

and then loaded in the grunt shell via

```
grunt> A = load 'input.bam' using BamUDFLoader('yes');
```

(the 'yes' chooses read attributes to be loaded; choose 'no' whenever these are not required).
Once some operations have been performed, the resulting (modified) read data can then be stored into a new BAM file via

```
grunt> store A into 'output.bam' using BamUDFStorer('input.bam.asciiheader');
```

and can also be exported from HDFS to the local filesystem via

```
prepareBamOutput.sh output.bam
```

(note: the Pig store operation requires a valid header for the BAM output file, for example the header of the source file used to generate it, which is generated automatically by the `prepareBamInput.sh` script used to import it)
Note that dumping the BAM data to the screen (similarly to samtools view) can be done simply by

```
grunt> dump A;
```

Another very useful Pig command is describe, which returns the schema that Pig uses for a given data bag. Example:

```
grunt> describe A;
```

which returns for BAM data

```
A: {name: chararray,start: int,end: int,read: chararray,cigar: chararray,
 basequal: chararray,flags: int,insertsize: int,mapqual:int,matestart: int,
 materefindex: int,refindex: int,refname: chararray,attributes: map[]}
```

Note that all fields except the attributes are standard data types (strings or integers). Specific attributes can be accessed via attributes#'name', for example

```
grunt> B = FOREACH A GENERATE name, attributes#'MD';
grunt> dump B;
```

will output all read names and their corresponding MD tag. Other useful commands are LIMIT and SAMPLE, which can be used for example for obtaining a subset of reads from a BAM/SAM file which can be useful for debugging.

```
grunt> B = LIMIT A 20;
```

will assign the first 20 records of A to B, while

```
grunt> B = SAMPLE A 0.01;
```

will sample from A with sampling probability 0.01.

3.1.1 Filtering out unmapped reads and PCR or optical duplicates

Since the `flags` field of a SAM record is exposed to Pig, one can simply use it to filter out all tuples (i.e., SAM records) that do not have the corresponding bit set.

```
grunt> A = FILTER A BY (flags/4)%2==0 and (flags/1024)%2==0;
```

For convenience SeqPig provides a set of filters that allow a direct access to the relevant fields. The previous example is equivalent to

```
grunt> run scripts/filter_defs.pig
grunt> A = FILTER A BY not ReadUnmapped(flags) and not IsDuplicate(flags);
```

For more details on available filters consider looking at file `scripts/filter_defs.pig`.

3.1.2 Filtering out reads with low mapping quality

Other fields can also be used for filtering, for example the read mapping quality value as shown below.

```
grunt> A = FILTER A BY mapqual > 19;
```

3.1.3 Filtering by regions (samtools syntax)

SeqPig also supports filtering by samtools *region* syntax. The following examples selects base positions 1 to 44350673 of chromosome 20.

```
grunt> DEFINE myFilter CoordinateFilter('input.bam.asciiheader','20:1-44350673');
grunt> A = FILTER A BY myFilter(refindex,start,end);
```

Note that filtering by regions requires a valid SAM header for mapping sequence names to sequence indices. This file is generated automatically when BAM files are imported via the `prepareBamInput.sh` script.

3.1.4 Sorting BAM files

Sorting an input BAM file by chromosome, reference start coordinate, strand and readname (in this hierarchical order):

```
grunt> A = FOREACH A GENERATE name, start, end, read, cigar, basequal, flags, insertsize,
mapqual, matestart, materefindex, refindex, refname, attributes, (flags/16)%2;
grunt> A = ORDER A BY refname, start, $14, name;
```

NOTE: this is roughly equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=input_sorted.bam ${SEQPIG_HOME}/scripts/
    sort_bam.pig
```

3.1.5 Computing read coverage

Computing read coverage over reference-coordinate bins of a fixed size, for example:

```
grunt> B = GROUP A BY start/200;
grunt> C = FOREACH B GENERATE group, COUNT(A);
grunt> dump C;
```

will output the number of reads that lie in any non-overlapping bin of size 200 base pairs.

3.1.6 Computing base frequencies (counts) for each reference coordinate

```
grunt> A = FOREACH A GENERATE read, flags, refname, start, cigar, basequal, mapqual;
grunt> A = FILTER A BY (flags/4)%2==0;
grunt> RefPos = FOREACH A GENERATE ReadRefPositions(read, flags, refname, start, cigar,
    basequal), mapqual;
grunt> flatset = FOREACH RefPos GENERATE flatten($0), mapqual;
grunt> grouped = GROUP flatset BY ($0, $1, $2);
grunt> base_counts = FOREACH grouped GENERATE group.chr, group.pos, group.base, COUNT(flatset
    );
grunt> base_counts = ORDER base_counts BY chr,pos;
grunt> store base_counts into 'input.basecounts';
```

NOTE: this is roughly equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=input.basecounts -param pparallel=1 ${
    SEQPIG_HOME}/scripts/basefreq.pig
```

3.1.7 Pileup

Generating samtools compatible pileup (for a correctly sorted BAM file with MD tags aligned to the same reference, should produce the same output as `samtools mpileup -A -f ref.fasta -B input.bam`):

```
grunt> A = load 'input.bam' using BamUDFLoader('yes');
grunt> B = FILTER A BY (flags/4)\%2==0 and (flags/1024)\%2==0;
grunt> C = FOREACH B GENERATE ReadPileup(read, flags, refname, start, cigar,
     basequal, attributes#'MD', mapqual), start, flags, name;
grunt> C = FILTER C BY $0 is not null;
grunt> D = FOREACH C GENERATE flatten($0), start, flags, name;
grunt> E = GROUP D BY (chr, pos);
grunt> F = FOREACH E { G = FOREACH D GENERATE refbase, pileup, qual, start,
     (flags/16)\%2, name; G = ORDER G BY start, $4, name; GENERATE group.chr,
     group.pos, PileupOutputFormatting(G, group.pos); }
grunt> G = ORDER F BY chr, pos;
grunt> H = FOREACH G GENERATE chr, pos, flatten($2);
grunt> store H into 'input.pileup' using PigStorage('\t');
```

NOTE: this is equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=input.pileup -param pparallel=1
   ${SEQPIG_HOME}/scripts/pileup.pig
```

The script essentially does the following:

1. Import BAM file and filter out unmapped or duplicate reads (`A, B`)
2. Break up each read and produce per-base pileup output (`C, D`)

3. Group all thus generated pileup output based on a (chromosome, position) coordinate system (`E`)
4. For each of the groups, sort its elements by their position, strand and name; then format the output according to samtools (`F`)
5. Sort the final output again by (chromosome, position) and perform some Pig operation by unnesting tuples (`G, H`)
6. Store the output to a directory inside HDFS (last line)

There are two optional parameters for pileup.pig: `min_map_qual` and `min_base_qual` (both with default value 0) that filter out reads with either insufficient map quality or base qualities. Their values can be set the same way as the other parameters above.
There is an alternative pileup script which typically performs better but is more sensitive to additional parameters. This second script, `pileup2.pig`, is based on a *binning* of the reads according to intervals on the reference sequence. The pileup output is then generated on a by-bin level and not on a by-position level. This script can be invoked with the same paramters as `pileup2.pig`. However, it has tunable parameters that determien the size of the bins (`binsize`) and the maximum number of reads considered per bin (`reads_cutoff`), wich is similar to the maximum depth parameter that samtools accepts. Note, however, since it is set on a per-bin level you may chose it dependent on the read length and bin size, as well as the amount of memory available to the nodes.

3.1.8 Collecting read-mapping-quality statistics

In order to evaluate the output of an aligner, it may be useful to consider the distribution of the mapping quality over the collection of reads. Due to Pig's GROUP operator this is fairly easy.

```
grunt> A = load 'input.bam' using BamUDFLoader('yes');
grunt> B = FILTER A BY (flags/4)\%2==0 and (flags/1024)\%2==0;
grunt> read_stats_data = FOREACH B GENERATE mapqual;
grunt> read_stats_grouped = GROUP read_stats_data BY mapqual;
grunt> read_stats = FOREACH read_stats_grouped GENERATE group, COUNT($1);
grunt> read_stats = ORDER read_stats BY group;
grunt> STORE read_stats into 'mapqual_dist.txt';
```

NOTE: this is equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=mapqual_dist.txt ${SEQPIG_HOME}/scripts/
    read_stats.pig
```

3.1.9 Collecting per-base statistics of reads

Sometimes it may be useful to analyze a given set of reads for a bias towards certain bases being called at certain positions inside the read. The following simple script generates for each reference base and each position inside a read the distribution of the number of read bases that were called.

```
grunt> A = load 'input.bam' using BamUDFLoader('yes');
grunt> B = FILTER A BY (flags/4)\%2==0 and (flags/1024)\%2==0;
grunt> C = FOREACH B GENERATE ReadSplit(name,start,read,cigar,basequal,flags,mapqual,refindex
    ,refname,attributes#'MD');
grunt> D = FOREACH C GENERATE FLATTEN($0);
grunt> base_stats_data = FOREACH D GENERATE refbase, basepos, UPPER(readbase) AS readbase;
grunt> base_stats_grouped = GROUP base_stats_data BY (refbase, basepos, readbase);
grunt> base_stats_grouped_count = FOREACH base_stats_grouped GENERATE group.$0 AS refbase,
    group.$1 AS basepos, group.$2 as readbase, COUNT($1) AS bcount;
grunt> base_stats_grouped = GROUP base_stats_grouped_count by (refbase, basepos);
grunt> base_stats = FOREACH base_stats_grouped {
       TMP1 = FOREACH base_stats_grouped_count GENERATE readbase, bcount;
       TMP2 = ORDER TMP1 BY bcount desc;
       GENERATE group.$0, group.$1, TMP2;
   }
grunt> STORE base_stats into 'outputfile_readstats.txt';
```

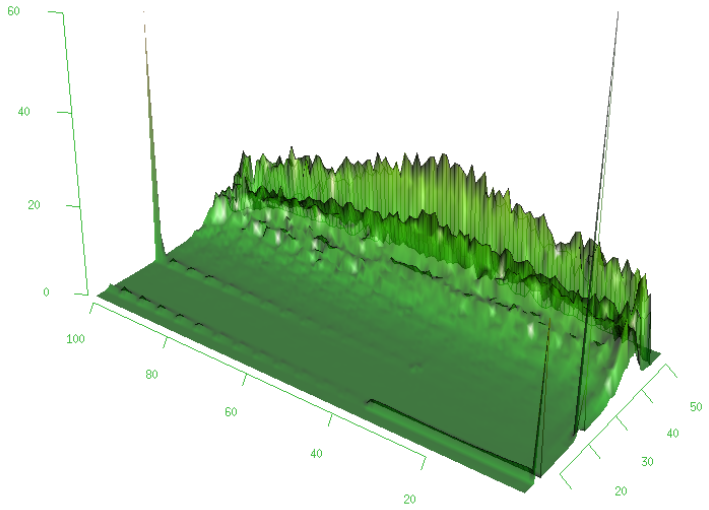

Figure 1: 3-D Histogram of base qualities over read length for a sample BAM file. The x-axis (values range 1 to 100) shows the index of bases in the read, while the y-axis shows base quality. The z-axis is the scaled frequency. The plot was generated by converting the output of `basequal_stats.pig` using `tools/basequal_stats2matrix.pl` and then plotted using `tools/plot_basequal_stats.R`.

A possible output has the form (for a BAM file with 50 reads):

```
A       0       {(A,19),(G,2)}
A       1       {(A,10)}
A       2       {(A,18)}
A       3       {(A,16)}
A       4       {(A,14)}
A       5       {(A,15)}
A       6       {(A,16),(G,2)}
...
A       98      {(A,7)}
A       99      {(A,14)}
C       0       {(C,6)}
C       1       {(C,11)}
C       2       {(C,9)}
...
```

NOTE: this example script is equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=outputfile_readstats.txt $SEQPIG_HOME/
    scripts/basequal_stats.pig
```

Figure 1 shows the distribution obtained from a sample BAM file.

3.1.10 Collecting per-base statistics of basequalities for reads

Similarly as previously for the read bases themselves, we can also collect frequencies for base-qualities depending on the position of the base inside the reads. If these fall off too quickly for later positions, it may indicate some quality issues with the run. The resulting script is actually fairly similar to the previous one with the difference of not grouping over the reference bases.

```
grunt> A = load 'input.bam' using BamUDFLoader('yes');
grunt> B = FILTER A BY (flags/4)\%2==0 and (flags/1024)\%2==0;
grunt> C = FOREACH B GENERATE ReadSplit(name,start,read,cigar,basequal,flags,mapqual,refindex
    ,refname,attributes#'MD');
```

```
grunt> D = FOREACH C GENERATE FLATTEN($0);
grunt> base_stats_data = FOREACH D GENERATE basepos, basequal;
grunt> base_stats_grouped = GROUP base_stats_data BY (basepos, basequal);
grunt> base_stats_grouped_count = FOREACH base_stats_grouped GENERATE group.$0 as basepos,
    group.$1 AS basequal, COUNT($1) AS qcount;
grunt> base_stats_grouped = GROUP base_stats_grouped_count BY basepos;
grunt> base_stats = FOREACH base_stats_grouped {
       TMP1 = FOREACH base_stats_grouped_count GENERATE basequal, qcount;
       TMP2 = ORDER TMP1 BY basequal;
       GENERATE group, TMP2;
}
grunt> STORE base_stats into 'outputfile_basequalstats.txt';
```

A possible output has the form (for a BAM file with 50 reads):

```
0      {(37,10),(42,1),(51,20),(52,1),(59,1),(61,1),(62,1),(67,2),(68,2),(70,2),(71,4),(72,3)
    ,(73,1),(75,2)}
1      {(53,1),(56,1),(61,1),(63,1),(64,1),(65,2),(67,4),(68,3),(69,2),(70,7),(71,3),(72,3)
    ,(73,1),(74,4),(75,2),(76,5),(77,6),(78,2),(80,1)}
2      {(45,1),(46,1),(51,2),(57,1),(61,1),(65,2),(66,3),(67,2),(69,3),(71,4),(72,2),(73,6)
    ,(74,7),(75,1),(76,8),(77,2),(78,3),(80,1)}
3      {(58,1),(59,1),(60,1),(61,1),(62,1),(64,1),(65,2),(67,2),(68,1),(69,5),(70,1),(71,3)
    ,(72,7),(73,2),(74,4),(75,6),(76,2),(77,4),(78,3),(79,1),(81,1)}
4      {(55,1),(60,1),(61,1),(62,1),(64,1),(66,1),(67,3),(68,2),(69,1),(70,7),(71,2),(72,1)
    ,(73,4),(74,2),(75,2),(76,2),(77,2),(78,3),(79,7),(80,4),(81,2)}
5      {(51,1),(52,2),(54,1),(58,2),(62,2),(63,1),(66,3),(68,4),(70,1),(71,1),(72,2),(73,3)
    ,(74,1),(75,8),(76,1),(77,5),(78,1),(79,6),(80,3),(81,3)}
...
```

NOTE: this example script is equivalent to executing from the command line:

```
$ pig -param inputfile=input.bam -param outputfile=outputfile_basequalstats.txt $SEQPIG_HOME/
    scripts/basequal_stats.pig
```

3.1.11 Filtering reads by mappability threshold

The script `filter_mappability.pig` filters reads in a given BAM file based on a given mappability threshold. Both input BAM and mappability file need to reside inside HDFS

```
$ pig -param inputfile=/user/hadoop/input.bam -param outputfile=/user/hadoop/output.bam -
    param regionfile=/user/hadoop/mappability.100kbp.txt -param threshold=90 $SEQPIG_HOME/
    scripts/filter_mappability.pig
```

Note that since the script relies on distributing the bam file header and the mappability file via Hadoop's distributed cache, it is not possible to run it in Pig's local mode.

3.2 Processing Qseq and Fastq data

SeqPig supports the import and export of non-aligned reads stored in Qseq and Fastq data. Due to Pig's philosophy that all records correspond to tuples, which form bags, reads can be processed in very much the same way independent on for example whether they are stored in Qseq or Fastq.

3.2.1 Converting Qseq to Fastq and vice versa

The following two lines simply convert an input Qseq into Fastq.

```
grunt> reads = load 'input.qseq' using QseqUDFLoader();
grunt> STORE reads INTO 'output.fastq' using FastqUDFStorer();
```

The other direction works analogously.

3.2.2 Clipping bases and basequalities

Assuming there were some problems in certain cycles of the sequencer, it may be useful to clip bases from reads. This example removes the last 3 bases and their qualities and stores the data under a new filename. Note that here we rely on the SUBSTRING and LENGTH string functions, which is part of the PiggyBank (see Section 2.2).

```
grunt> DEFINE SUBSTRING org.apache.pig.piggybank.evaluation.string.SUBSTRING();
grunt> DEFINE LENGTH org.apache.pig.piggybank.evaluation.string.LENGTH();
grunt> reads = load 'input.qseq' using QseqUDFLoader();
grunt> B = FOREACH A GENERATE instrument, run_number, flow_cell_id, lane, tile, xpos, ypos,
    read, qc_passed, control_number, index_sequence, SUBSTRING(sequence, 0, LENGTH(sequence)
    - 3) AS sequence, SUBSTRING(quality, 0, LENGTH(quality) - 3) AS quality;
grunt> store B into 'output.qseq' using QseqUDFStorer();
```

NOTE: this example script is equivalent to executing from the command line:

```
$ pig -param inputfile=input.qseq -param outputfile=output.qseq -param backclip=3
    $SEQPIG_HOME/scripts/clip_reads.pig
```

4 Further information

4.1 Hadoop parameters

Some scripts (such as `pileup2.pig`) require an amount of memory that depends on the choice of command line parameters. If you are in need for performance tuning, consider the following Hadoop specific parameters in `mapred-site.xml`:

- `mapred.tasktracker.map.tasks.maximum`:
  the maximum number of mappers per worker node
- `mapred.tasktracker.reduce.tasks.maximum`:
  the maximum number of reducers per worker node
- `mapred.child.java.opts`:
  Java options passed to child virtual machines at the worker nodes; this may contain for example `-Xmx1000M` for setting the maximum Heap size to 1GB, depending on the cluster resources

Aditionally, it is possible to pass command line parameters (such as mapper and reducer memory limits) along. Consider for example the pig call (see `tools/tun_all_pileup2.sh`)

```
${PIG_HOME}/bin/pig -Dpig.additional.jars=${SEQPIG_HOME}/lib/hadoop-bam-5.0.jar:${SEQPIG_HOME
    }/build/jar/SeqPig.jar:${SEQPIG_HOME}/lib/seal.jar:${SEQPIG_HOME}/lib/picard-1.76.jar:${
    SEQPIG_HOME}/lib/sam-1.76.jar -Dmapred.job.map.memory.mb=${MAP_MEMORY} -Dmapred.job.
    reduce.memory.mb=${REDUCE_MEMORY} -Dmapred.child.java.opts=-Xmx${CHILD_MEMORY}M -Dudf.
    import.list=fi.aalto.seqpig -param inputfile=$INPUTFILE -param outputfile=$OUTPUTFILE -
    param pparallel=${REDUCESLOTS} ${SEQPIG_HOME}/scripts/pileup2.pig
```

By setting appropriate values for `MAP_MEMORY, REDUCE_MEMORY, CHILD_MEMORY` and for the number of available reduce slots `REDUCESLOTS` different performance can be achieved.

4.2 Compression

For performance reasons it can be advisable to enable compression of Hadoop map (and possible reduce) output, as well as temporary data generated by Pig. The details depend on which compression codecs are used, but it can be enabled by passing parameters along the lines of

```
 -Djava.library.path=/opt/hadoopgpl/native/Linux-amd64-64
 -Dpig.tmpfilecompression=true -Dpig.tmpfilecompression.codec=lzo
 -Dmapred.output.compress=true
 -Dmapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec
```

to the pig command. Note that currently not all Hadoop compression codecs are supported by Pig.