Introduction to *GenomicFiles*

Valerie Obenchain, Michael Love, Martin Morgan

Last modified: October 2014; Compiled: February 7, 2015

Contents

1 Introduction

This vignette illustrates how to use the *GenomicFiles* package for distributed computing across files. The functions in `GenomicFiles` manipulate and combine data subsets via two user-supplied functions, MAP and REDUCE. These are similar in spirit to `Map` and `Reduce` in *base R*. Together they provide a flexible interface to extract, manipulate and combine data. Both functions are executed in the distributed step which means results are combined on a single worker, not across workers.

We assume the reader has some previous experience with *R* and with basic manipulation of ranges objects such as `GRanges` and `GAlignments` and file classes such as `BamFile` and `BigWigFile`. See the vignettes and documentation in *GenomicRanges*, *GenomicAlignments*, *Rsamtools* and *rtracklayer* for an introduction to these classes.

The *GenomicFiles* package is available at bioconductor.org and can be downloaded via `biocLite`:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("GenomicFiles")
```

2 Quick Start

GenomicFiles offers functions for the parallel extraction and combination of data subsets. A user-defined MAP function extracts and manipulates data while an optional REDUCE function consolidates the output of MAP.

```
> library(GenomicFiles)
```

Ranges can be a `GRanges`, `GRangesList` or `GenomicFiles` class.

```
> gr <- GRanges("chr14", IRanges(c(19411500 + (1:5)*20), width=10))
```

File are supplied as a character vector or list of *File classes such as `BamFile`, `BigWigFile` etc.

```
> library(RNAseqData.HNRNPC.bam.chr14)
> fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES
```

The MAP function extracts and manipulates data subsets. Here we compute pileups for a given range and file.

```
> MAP <- function(range, file, ...) {
+     library(Rsamtools)
+     pileup(file, scanBamParam=ScanBamParam(which=range))
+ }
```

`reduceByFile` sends each file to a worker where MAP is applied to each file / range combination. When `summarize=TRUE` the output is a `SummarizedExperiment` object.

```
> se <- reduceByFile(gr, fls, MAP, summarize=TRUE)
> se
```

```
class: SummarizedExperiment
dim: 5 8
exptData(0):
assays(1): data
rownames: NULL
rowData metadata column names(0):
colnames(8): ERR127306 ERR127307 ... ERR127304 ERR127305
colData names(1): filePath
```

Results are stored in the assays slot.

```
> dim(assays(se)$data)  ## ranges x files
```

```
[1] 5 8
```

`reduceByRange` sends each range to a worker and extracts the same range from all files. Adding a reducer to this example combines the pileups from each range across files.

```
> REDUCE <- function(mapped, ...) {
+     cmb = do.call(rbind, mapped)
+     xtabs(count ~ pos + nucleotide, cmb)
+ }
> lst <- reduceByRange(gr, fls, MAP, REDUCE, iterate=FALSE)
```

The result is a list where each element is a summary table of counts for a single range across all 8 files.

```
> head(lst[[1]], 3)
```

```
          nucleotide
pos        A C G T N = -
  19411520 2 0 0 0 0 0 0
  19411521 0 2 0 0 0 0 0
  19411522 0 0 0 2 0 0 0
```

3 Overview of classes and functions

3.1 `GenomicFiles` class

The `GenomicFiles` class is a matrix-like container where rows represent ranges of interest and columns represent files. The object can be subset on files and / or ranges to perform different experimental runs. The class is a lightweight version of the `SummarizedExperiment` class. It has slots for `files`, `rowData`, `colData` and `exptData` but does not contain the `assays` slot.

```
> GenomicFiles(gr, fls)
```

```
GenomicFiles object with 5 ranges and 8 files:
files: ERR127306_chr14.bam, ERR127307_chr14.bam, ..., ERR127304_chr14.bam, ERR127305_chr14.bam
detail: use files(), rowData(), colData(), ...
```

A `GenomicFiles` can be used as the `ranges` argument to the functions in this package. When `summarize=TRUE`, data from the common slots are transferred to the `SummarizedExperiment` result. NOTE: Results can only be put into a `SummarizedExperiment` when no reduction is performed because of the matching dimensions requirement (i.e., a REDUCE collapses the results in one dimension).

3.2 Functions

Functions in `GenomicFiles` manipulate and combine data across or within files using the parallel infrastructure provided in `BiocParallel`. Files and ranges are sent to workers along with MAP and REDUCE functions. The MAP extracts and/or manipulates data and REDUCE consolidates the results from MAP. Both MAP and REDUCE are executed in the distributed step and therefore reduction occurs on data from the same worker, not across workers.

The chart in Figure 1 represents the division of labor in `reduceByRange` and `reduceRanges` with 3 files and 4 ranges. These functions split the problem by range which allows subsets (i.e., the same range) to be combined across different files. `reduceByRange` invokes MAP for each range / file combination whereas `reduceRanges` invokes MAP a single time with one file and all ranges. `reduceRanges` passes all files as an argument to MAP.

In contrast to the 'byRange' approach, `reduceByFile` and `reduceFiles` (Figure 2) split the problem by file. This division allows subsets (i.e., multiple ranges) to be combined within the same file. `reduceByFile` invokes MAP for each file / range combination and `reduceFiles` invokes MAP a single time with one range and all files. `reduceFiles` is a useful approach when distinguishing between ranges is not important.

4 Queries across files: `reduceByRange` and `reduceRanges`

The `reduceByRange` and `reduceRanges` functions are designed for analyses that compare or combine data subsets across files. The first example in this section computes pileups on subsets from individual files then sums over all files. The second example computes coverage on a group of ranges for each file then performs a basepair-level t-test across files. The t-test example also demonstrates how to use a blocking factor to differentiate files by experimental group (e.g., case vs control).

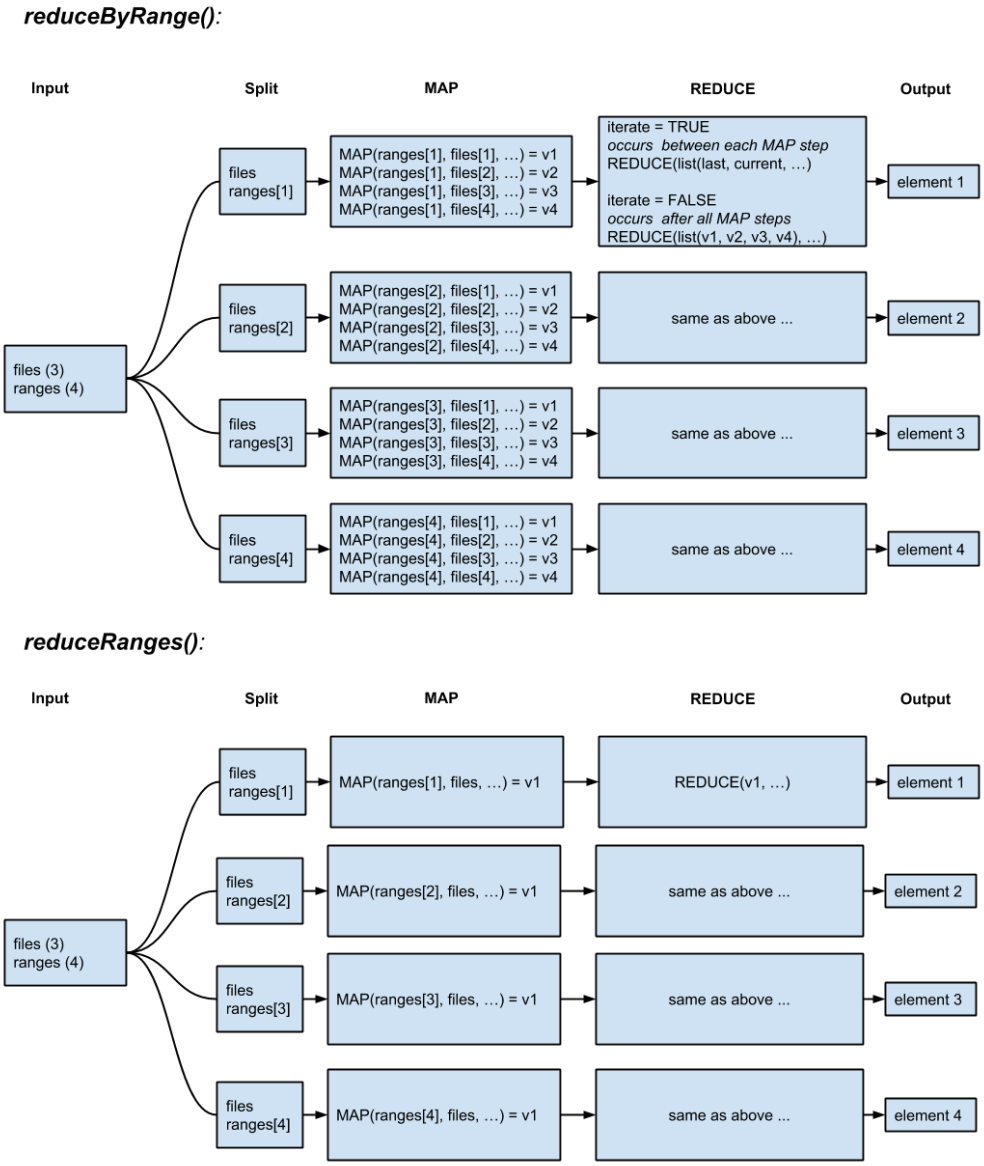

Figure 1: Mechanics of `reduceByRange` and `reduceRanges`

4.1 Pileup summaries

In this example nucleotide counts (pileups) are computed for the same ranges in each file (MAP step). Pileups are then summed by position resulting in a single table for each range across all files (REDUCE step).

Create a *GRanges* with regions of interest:

```
> gr <- GRanges("chr14", IRanges(c(19411677, 19659063, 105421963,
+                105613740), width=20))
```

The `bam2R` function from the *deepSNV* package is used to compute the statistics. The MAP invokes `bam2R` and retains only the nucleotide counts (see `?bam2R` for other output fields). Counts from the reference strand are uppercase and counts from the complement are lowercase.

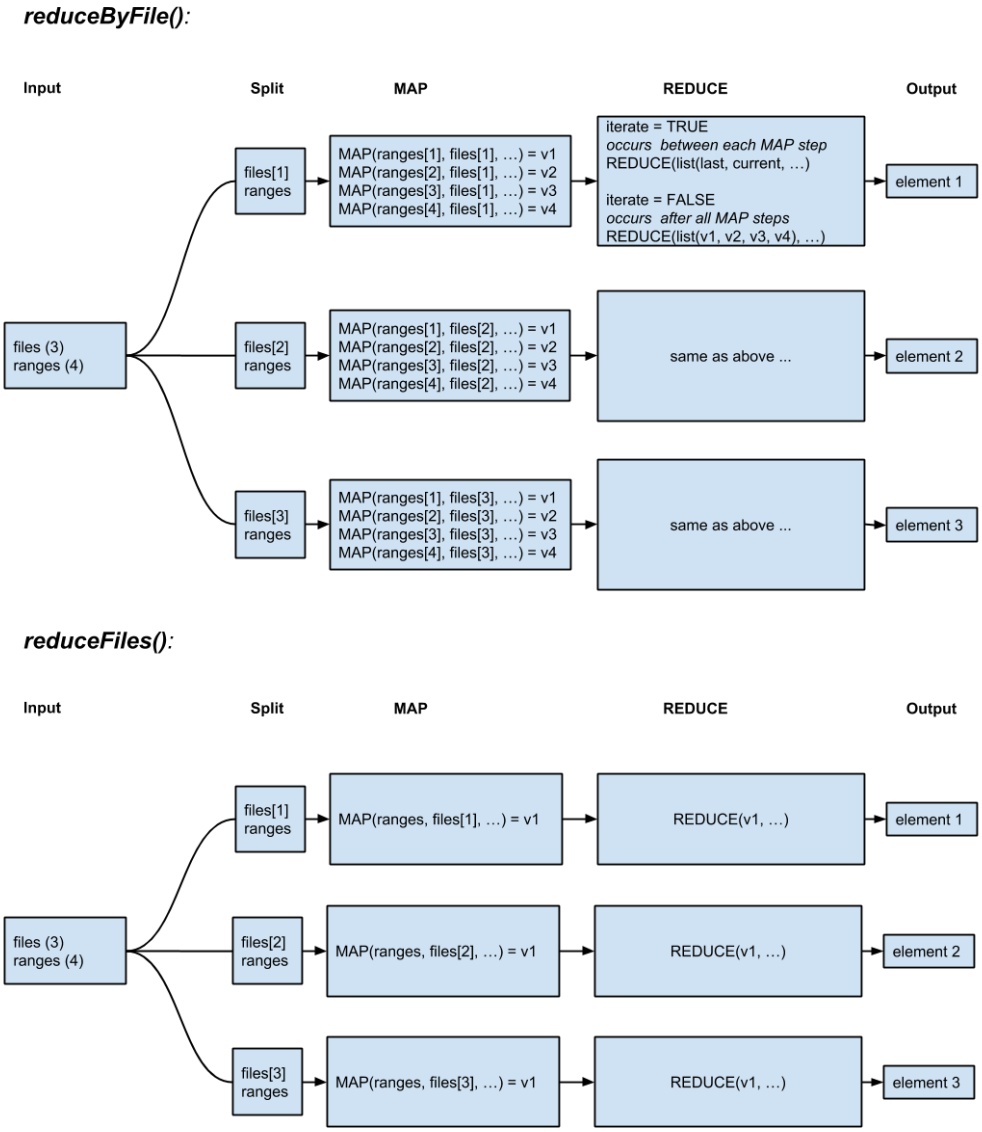

Figure 2: Mechanics of `reduceByFile` and `reduceFiles`

Because the `bam2R` function is not explicitly passed through the MAP, `deepSNV` must be loaded on each worker so the function can be found.

```
> MAP <- function(range, file, ...) {
+     require(deepSNV)
+     ct = bam2R(file, seqlevels(range), start(range), end(range), q=0)
+     ct[, c("A", "T", "C", "G", "a", "t", "c", "g")]
+ }
```

With no REDUCE function, the output is a list the same length as the number of ranges where each list element is the length of the number of files.

```
> pile1 <- reduceByRange(gr, fls, MAP)
> length(pile1)
```

```
[1] 4
```

```
> elementLengths(pile1)
```

```
[1] 8 8 8 8
```

Next add a REDUCE to sum the counts by position.

```
> REDUCE <- function(mapped, ...)
+     Reduce("+", mapped)
```

The output is again a list with the same length as the number of ranges but the element lengths have been reduced to 1.

```
> pile2 <- reduceByRange(gr, fls, MAP, REDUCE)
> length(pile2)
```

```
[1] 4
```

```
> elementLengths(pile2)
```

```
[1] 20 20 20 20
```

Each element is a matrix of counts (position by nucleotide) for a single range summed over all files.

```
> head(pile2[[1]])
```

```
      A T  C  G  a t  c  g
[1,] 14 0  0  0 21 0  0  0
[2,] 16 0  0  0 21 0  0  0
[3,] 16 0  0  0 20 0  0  0
[4,]  0 0  0 16  0 0  0 20
[5,]  0 0 20  0  0 0 19  0
[6,] 19 0  0  0 19 0  0  0
```

4.2 Basepair-level t-test with case / control groups

In this example coverage is computed for a region of interest in multiple files. A grouping variable that defines case / control status is passed as an extra argument to `reduceByRange` and used in the reduction step to perform the t-test.

Define ranges of interest,

```
> roi <- GRanges("chr14", IRanges(c(19411677, 19659063, 105421963,
+                105613740), width=20))
```

and assign the case, control grouping of files. (Grouping is arbitrary in this example.)

```
> grp <- factor(rep(c("A","B"), each=length(fls)/2))
```

The MAP reads in alignments from each BAM file and computes coverage. Coverage is coerced from an RleList to numeric vector for later use in the t-test.

```
> MAP <- function(range, file, ...) {
+     library(Rsamtools)
+     param <- ScanBamParam(which=range)
+     as.numeric(unlist(coverage(file, param=param)[range], use.names=FALSE))
+ }
```

REDUCE combines the coverage vectors into a matrix, identifies all-zero rows, and performs row-wise t-testing using the `rowttests` function from the *genefilter* package. The index of which rows correspond to which basepair of the original range is stored as a column `offset`.

```
> REDUCE <- function(mapped, ..., grp) {
+     mat = simplify2array(mapped)
+     idx = which(rowSums(mat) != 0)
+     df = genefilter::rowttests(mat[idx,], grp)
+     cbind(offset = idx - 1, df)
+ }
```

The file grouping is passed as an extra argument to `reduceByRange`. `iterate=FALSE` postpones the reduction until coverage vectors for all files have been computed. This delay is necessary because REDUCE uses the file grouping factor to perform the t-test and relies on the coverage vectors for all files to be present.

```
> ttest <- reduceByRange(roi, fls, MAP, REDUCE, iterate=FALSE, grp=grp)
```

The result is a list of summary tables of basepair-level t-test statistics for each range across all files.

```
> head(ttest[[1]], 3)
```

```
  offset statistic   dm   p.value
1      0 1.1489125 2.75 0.2943227
2      1 0.9761871 2.25 0.3666718
3      2 0.8320503 1.50 0.4372365
```

These tables can be added to the `roi` GRanges as a metadata column.

```
> mcols(roi)$ttest <- ttest
> head(roi)
```

```
GRanges object with 4 ranges and 1 metadata column:
      seqnames                ranges strand |    ttest
         <Rle>             <IRanges>  <Rle> |   <list>
  [1]    chr14 [ 19411677,  19411696]      * | ########
  [2]    chr14 [ 19659063,  19659082]      * | ########
  [3]    chr14 [105421963, 105421982]      * | ########
  [4]    chr14 [105613740, 105613759]      * | ########
  -------
  seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

5 Queries within files: `reduceByFile` and `reduceFiles`

`reduceByFile` and `reduceFiles` compare or combine data subsets within files. `reduceByFile` allows for more fine-tuned manipulation over the subset for each range / file combination. If differentiating between ranges is not important, `reduceFiles` can be used to treat the ranges as a group.

In this section read junctions are counted for individual subsets within a file then combined based on user-defined selection criteria. Another example computes coverage over complete BAM files by streaming over a set of continuous ranges. The coverage example is performed with both `reduceByFile` and `reduceFiles` to demonstrate the passing ranges to MAP individually vs all at once. The last example uses a MAP function to chunk through subsets when the data are too large for available memory.

5.1 Counting read junctions

This example highlights how `reduceByFile` allows detailed control over the combination of data subsets from distinct ranges within the same file.

Define ranges of interest.

```
> gr <- GRanges("chr14", IRanges(c(19100000, 106000000), width=1e7))
```

The MAP produces a table of junction counts (i.e., 'N' operations in the CIGAR) for each range.

```
> MAP <- function(range, file, ...) {
+     library(GenomicAlignments) ## for readGAlignments() and ScanBamParam()
+     param = ScanBamParam(which=range)
+     gal = readGAlignments(file, param=param)
+     table(njunc(gal))
+ }
```

Create a GenomicFiles object.

```
> gf <- GenomicFiles(gr, fls)
> gf
```

```
GenomicFiles object with 2 ranges and 8 files:
files: ERR127306_chr14.bam, ERR127307_chr14.bam, ..., ERR127304_chr14.bam, ERR127305_chr14.bam
detail: use files(), rowData(), colData(), ...
```

The GenomicFiles object or any subset of the object can be used as the `ranges` argument to functions in `GenomicFiles`. Here the object is subset on 3 files and both ranges.

```
> counts1 <- reduceByFile(gf[,1:3], MAP=MAP)
> length(counts1)          ## 3 files
```

```
[1] 3
```

```
> elementLengths(counts1)  ## 2 ranges
```

```
ERR127306 ERR127307 ERR127308
        2         2         2
```

Each list element has a table of counts for each range.

```
> counts1[[1]]
```

```
[[1]]

     0      1      2
110630  33527    944

[[2]]

   0    1
2329   57
```

Add a reducer that combines counts for records in each range with exactly 1 junction.

```
> REDUCE <- function(mapped, ...)
+     sum(sapply(mapped, "[", "1"))
> reduceByFile(gr, fls, MAP, REDUCE)
```

```
$ERR127306
[1] 33584

$ERR127307
[1] 36388

$ERR127308
[1] 36710

$ERR127309
[1] 32620
```

```
$ERR127302
[1] 30348

$ERR127303
[1] 31800

$ERR127304
[1] 35358

$ERR127305
[1] 35369
```

Next invoke `reduceFiles` with the same files and MAP function. `reduceFiles` treats all ranges as a group and counts junctions for all ranges simultaneously.

```
> counts2 <- reduceFiles(gf[,1:3], MAP=MAP)
```

In the `reduceByFile` example junctions were counted for each range individually which allowed us to see results for the individual ranges and combine them on the fly based on specific criteria. In contrast, `reduceFiles` counts junctions for all ranges simultaneously.

```
> ## reduceFiles returns counts for all ranges.
> counts2[[1]]
```

```
[[1]]

     0      1      2
112959  33584    944
```

```
> ## reduceByFile returns counts for each range separately.
> counts1[[1]]
```

```
[[1]]

     0      1      2
110630  33527    944

[[2]]

   0    1
2329   57
```

5.2 Coverage 1: `reduceByFile`

Files that are too large to fit in memory can be streamed over by creating 'tiles' or ranges that span the whole file. The `tileGenome` function creates a set of continuous ranges that span a given seqlength(s). The sample BAM files contain only chr14 so we extract the appropriate seqlength from the BAM files and use it in `tileGenome`. In this example we create 5 ranges but the optimal value for `ntile` will depend on the application and the size of the chromosome (or genome) to be tiled.

```
> chr14_seqlen <- seqlengths(seqinfo(BamFileList(fls))["chr14"])
> tiles <- tileGenome(chr14_seqlen, ntile=5)
```

`tiles` is a GRangesList of length `ntile` with one range per element.

```
> tiles
```

```
GRangesList object of length 5:
[[1]]
GRanges object with 1 range and 0 metadata columns:
      seqnames        ranges strand
         <Rle>     <IRanges>  <Rle>
  [1]    chr14 [1, 21469908]      *

[[2]]
GRanges object with 1 range and 0 metadata columns:
      seqnames               ranges strand
  [1]    chr14 [21469909, 42939816]      *

[[3]]
GRanges object with 1 range and 0 metadata columns:
      seqnames               ranges strand
  [1]    chr14 [42939817, 64409724]      *

...
<2 more elements>
-------
seqinfo: 1 sequence from an unspecified genome
```

MAP computes coverage for each range. The sum of coverage across all positions is recorded along with the width of the range.

```
> MAP = function(range, file, ...) {
+     library(GenomicAlignments)  ## for ScanBamParam() and coverage()
+     param = ScanBamParam(which=range)
+     rle = coverage(file, param=param)[range]
+     c(width = width(range), sum = sum(runLength(rle) * runValue(rle)))
+ }
```

REDUCE sums the width and coverage for all ranges in 'tiles'.

```
> REDUCE = function(mapped, ...) {
+     Reduce(function(i, j) Map("+", i, j), mapped)
+ }
```

When `iterate=TRUE` REDUCE is applied after each MAP step. Iterating prevents the data from growing too large on the worker. The total width and coverage sum for all ranges are returned for each file.

```
> cvg1 <- reduceByFile(tiles, fls, MAP, REDUCE, iterate=TRUE)
```

```
> cvg1[1]
$ERR127306
$ERR127306$width
[1] 107349540

$ERR127306$sum.chr14
[1] 57633506
```

5.3 Coverage 2: `reduceFiles`

In the first coverage example we used `reduceByFile` to invoke MAP for each file / range combination. This approach is useful when analyses require data manipulation at the level of each file / range subset prior to reduction. For many applications, however, distinguishing between ranges is not important and the overhead of an lapply over all ranges may be costly.

An alternative is to use `reduceFiles` which passes all ranges as a single argument to MAP. The ranges can be used to create a 'param' or passed as an argument to another function that operates on multiple ranges at at time.

This MAP computes coverage on all ranges at once and returns an RleList.

```
> MAP = function(range, file, ...) {
+     library(GenomicAlignments)  ## for ScanBamParam() and coverage()
+     coverage(file, param=ScanBamParam(which=range))[range]
+ }
```

REDUCE extracts the RleList from 'mapped' and collapses the coverage. Note that reduction could have be done in the MAP step on the output of coverage. Because all ranges are passed as a single argument, MAP is only called once on each worker. Consequences of a single invocation are (1) reduction can be done at the end of the MAP or by REDUCE and (2) REDUCE cannot be applied iteratively (this requires more than a single output from MAP).

```
> REDUCE = function(mapped, ...) {
+     sapply(mapped, Reduce, f = "+")
+ }
```

Recall 'tiles' is a GRangesList with one range per list element. We have no need for the grouping in this example so we pass 'tiles' as a GRanges.

```
> cvg2 <- reduceFiles(unlist(tiles), fls, MAP, REDUCE)
```

Output is a list of length 8 where each element is a single Rle of coverage for all ranges.

```
> cvg2[1]
```

```
$ERR127306
$ERR127306[[1]]
integer-Rle of length 21469908 with 489540 runs
  Lengths: 6818    9    8    1    1    2    2    2 ...    1    3    5    8    1   10  863
  Values :    0   22   23   19   17   18   17   15 ...   21   20   22   21   23   22    0
```

5.4 Coverage 3: `reduceFiles` with chunking

Continuing with the same coverage example. Now let's assume the result from calling `coverage` with all ranges in 'tiles' does not fit in available memory. We need a way to chunk through the ranges.

One option is to use `reduceByFile` to lapply through each range in 'tiles' individually and then apply a reducer as we did in the first coverage example. Because the 'tiles' GRangesList has only one range per list element this approach may be inefficient for a large number of ranges. To reduce the number of iterations in the lapply, the ranges in 'tiles' could be re-grouped into a GRangesList with more than one range per element.

Another approach is to write your own MAP function that chunks through the ranges. This has the advantage that, if resources are available, an additional level of parallel dispatch can be implemented.

MAP creates an index over the ranges which are passed to `bplapply`. The data are subset on each worker, coverage is computed and reduced for the ranges in the chunk.

```
> MAP = function(range, file, ...) {
+     library(BiocParallel)  ## for bplapply()
+     nranges = 2
+     idx = split(seq_along(range), ceiling(seq_along(range)/nranges))
+     bplapply(idx,
+         function(i, range, file) {
+             library(GenomicAlignments)  ## for ScanBamParam() and coverage()
+             chunk = range[i]
+             param = ScanBamParam(which=chunk)
+             cvg = coverage(file, param=param)[chunk]
```

```
+             Reduce("+", cvg)            ## collapse coverage within chunks
+         }, range, file)
+ }
```

REDUCE extracts and collapses the RleList of coverage for all chunks.

```
> REDUCE = function(mapped, ...) {
+     sapply(mapped, Reduce, f = "+")
+ }
```

Again 'tiles' are passed as a GRanges so the chunking in MAP defines the groups, not the structure of the GRangesList. Output is a list of length 8 where each list element is a single Rle of coverage.

```
> cvg3 <- reduceFiles(unlist(tiles), fls, MAP, REDUCE)
```

```
> cvg3[1]
$ERR127306
$ERR127306[[1]]
integer-Rle of length 21469908 with 489540 runs
  Lengths: 6818    9    8    1    1    2    2 ...    3    5    8    1   10  863
  Values :    0   22   23   19   17   18   17 ...   20   22   21   23   22    0
```

6 Chunking

6.1 Ranges in a file

Both `reduceByFile` and `reduceByRange` process `ranges` one element at a time. When `ranges` is a GRanges the element is a single range and when it is a GRangesList the element can contain multiple ranges.

If the GRanges is very long (many ranges) working one range at a time can be inefficient. Splitting the GRanges into a GRangesList allows `reduceByFile` and `reduceByRange` to work on groups of ranges and will gain speed and efficiency in most applications. This approach works as long as the analysis does not depend on keeping the ranges separate (i.e., MAP and REDUCE can be written to operate on groups of ranges instead of a single range).

For applications that combine data *within* a file, chunking can be done with `reduceByFile` and a GRangesList. Similarly, when chunking through ranges to combine data *across* files use `reduceByRange` with a GRangesList.

6.2 Records in a file

`reduceByYield` iterates through records in a single file that would otherwise not fit in memory. It is similar to a one dimensional `reduceByFile` but the arguments and approach are slightly different.

Similar to other `GenomicFiles` functions, data are manipulated and reduced with `MAP` and `REDUCE` functions. What sets `reduceByYield` apart are the use of `YIELD` and `DONE` arguments. `YIELD` is a function that returns a chunk of data to work on and `DONE` is a function that defines a stopping criteria.

Records from a single file are read by `readGAlignments` and limited by the `yieldSize` set in the BamFile.

```
> library(GenomicAlignments)
> bf <- BamFile(fls[1], yieldSize=100000)
> YIELD <- function(x, ...) readGAlignments(x)
```

MAP counts overlaps between the reads and a GRanges of interest while REDUCE sums counts over the chunks.

```
> gr <- unlist(tiles, use.names=FALSE)
> MAP <- function(value, gr, ...) {
+     library(GenomicRanges)  ## for countOverlaps()
+     countOverlaps(gr, value)
+ }
> REDUCE <- `+`
```

When `DONE` evaluates to TRUE, iteration stops. 'value' is the object returned from calling YIELD on the BAM file. At the end of file the length of records will be 0 and `DONE` will evaluate to TRUE.

```
> DONE <- function(value) length(value) == 0L
```

The MAP step is run in parallel when `parallel=TRUE`. 'parallel' is currently implemented for Unix/Mac only so we use multicore workers.

```
> register(MulticoreParam(3))
> reduceByYield(bf, YIELD, MAP, REDUCE, DONE, gr=gr, parallel=TRUE)
```

```
[[1]]
[1]  21465 163154  75498 212593 327785
```

Taking this one step further, we can use bplapply to distribute files to workers and call `reduceByYield` on each file. If adequate resources are available this example could have 2 levels of parallel dispatch, one at the file level (`bplapply`) and one at the MAP level (`reduceByYield(..., parallel=TRUE)`. This example takes the conservative approach and runs `reduceByYield` in serial on each worker.

The function 'FUN' will be run on each worker.

```
> FUN <- function(file, gr, YIELD, MAP, REDUCE, tiles, ...) {
+     library(GenomicAlignments)  ## for BamFile, readGAlignments()
+     library(GenomicFiles)       ## for reduceByYield()
+     gr <- unlist(tiles, use.names=FALSE)
+     bf <- BamFile(file, yieldSize=100000)
+     YIELD <- function(x, ...) readGAlignments(x)
+     MAP <- function(value, gr, ...) {
+         library(GenomicRanges)  ## for countOverlaps()
+         countOverlaps(gr, value)
+     }
+     REDUCE <- `+`
+     reduceByYield(bf, YIELD, MAP, REDUCE, gr=gr, parallel=FALSE)
+ }
```

bplapply distributes the files to workers. Each worker uses `reduceByYield` to iteratively count and reduce overlaps in a BAM file.

```
> bplapply(fls, FUN, gr=gr, YIELD=YIELD, MAP=MAP, REDUCE=REDUCE, tiles=tiles)
$ERR127306
[1]  21465 163154  75498 212593 327785

$ERR127307
[1]  23544 181551  91702 236845 341670

$ERR127308
[1]  23236 178270  84027 234735 355353

$ERR127309
[1]  20890 160804  82120 208961 305701

$ERR127302
[1]  20636 140052  89834 208824 283432

$ERR127303
[1]  22198 149809 106987 226217 281000

$ERR127304
[1]  25718 150984  94198 223797 316043

$ERR127305
[1]  25646 145655  79854 219333 327909
```

7 sessionInfo()

```
> toLatex(sessionInfo())
```

- R version 3.1.2 (2014-10-31), i386-w64-mingw32
- Locale: LC_COLLATE=C, LC_CTYPE=English_United States.1252, LC_MONETARY=English_United States.1252, LC_NUMERIC=C, LC_TIME=English_United States.1252
- Base packages: base, datasets, grDevices, graphics, methods, parallel, splines, stats, stats4, utils
- Other packages: BiocGenerics 0.12.1, BiocParallel 1.0.3, Biostrings 2.34.1, GenomeInfoDb 1.2.4, GenomicAlignments 1.2.1, GenomicFiles 1.2.1, GenomicRanges 1.18.4, IRanges 2.0.1, RNAseqData.HNRNPC.bam.chr14 0.3.2, Rsamtools 1.18.2, S4Vectors 0.4.0, VGAM 0.9-6, VariantAnnotation 1.12.9, XVector 0.6.0, deepSNV 1.12.0, rtracklayer 1.26.2

- Loaded via a namespace (and not attached): AnnotationDbi 1.28.1, BBmisc 1.9, BSgenome 1.34.1, BatchJobs 1.5, Biobase 2.26.0, BiocStyle 1.4.1, DBI 0.3.1, GenomicFeatures 1.18.3, RCurl 1.95-4.5, RSQLite 1.0.0, XML 3.98-1.1, annotate 1.44.0, base64enc 0.1-2, biomaRt 2.22.0, bitops 1.0-6, brew 1.0-6, checkmate 1.5.1, codetools 0.2-10, digest 0.6.8, fail 1.2, foreach 1.4.2, genefilter 1.48.1, iterators 1.0.7, sendmailR 1.2-1, stringr 0.6.2, survival 2.37-7, tools 3.1.2, xtable 1.7-4, zlibbioc 1.12.0