Contents

1 Introduction ...................................................... 2
2 Quick start ......................................................... 2
3 The BiocParallel Interface ........................................ 4
  3.1 Classes .......................................................... 4
    3.1.1 BiocParallelParam ........................................ 4
    3.1.2 register()ing BiocParallelParam instances .......... 5
  3.2 Functions ...................................................... 7
    3.2.1 Parallel looping, vectorized and aggregate operations .... 7
    3.2.2 Parallel evaluation environment ....................... 7
    3.2.3 Error handling and logging ........................... 8
    3.2.4 Locks and counters ................................... 8
4 Use cases .......................................................... 8
  4.1 Single machine ............................................... 8
    4.1.1 Multi-core ............................................... 8
    4.1.2 Clusters ................................................ 9
  4.2 Ad hoc cluster of multiple machines ....................... 10
    4.2.1 Sockets ................................................ 10
    4.2.2 MPI ................................................... 11
  4.3 Clusters with schedulers .................................... 12
    4.3.1 Cluster-centric ....................................... 12
    4.3.2 R-centric ........................................... 13
  4.4 Bioconductor Amazon Machine Image (AMI) ................. 14
5 Analyzing genomic data in Bioconductor .................... 17
6 For developers ................................................... 17
7 sessionInfo() .................................................... 18
1 Introduction

Numerous approaches are available for parallel computing in R. The CRAN Task View for high performance and parallel computing provides useful high-level summaries and package categorization. [http://cran.r-project.org/web/views/HighPerformanceComputing.html](http://cran.r-project.org/web/views/HighPerformanceComputing.html) Most Task View packages cite or identify one or more of snow, Rmpi, multicore or foreach as relevant parallelization infrastructure. Direct support in R for parallel computing started with release 2.14.0 with inclusion of the parallel package which contains modified versions of multicore and snow.

A basic objective of BiocParallel is to reduce the complexity faced when developing and using software that performs parallel computations. With the introduction of the BiocParallelParam object, BiocParallel aims to provide a unified interface to existing parallel infrastructure where code can be easily executed in different environments. The BiocParallelParam specifies the environment of choice as well as computing resources and is invoked by ‘registration’ or passed as an argument to the BiocParallel functions.

BiocParallel offers the following conveniences over the ‘roll your own’ approach to parallel programming.

- unified interface: BiocParallelParam instances define the method of parallel evaluation (multi-core, snow cluster, etc.) and computing resources (number of workers, error handling, cleanup, etc.).

- parallel iteration over lists, files and vectorized operations: bplapply, bpmapply and bpvec provide parallel list iteration and vectorized operations. bpiterate iterates through files distributing chunks to parallel workers.

- cluster scheduling: When the parallel environment is managed by a cluster scheduler through BatchJobs, job management and result retrieval are considerably simplified.

- support of foreach: The foreach and iterators packages are fully supported. Registration of the parallel back end uses BiocParallelParam instances.

2 Quick start

The BiocParallel package is available at bioconductor.org and can be downloaded via biocLite:

```r
source("http://bioconductor.org/biocLite.R")
biocLite("BiocParallel")
```

Load BiocParallel.

```r
library(BiocParallel)
```

The test function simply returns the square root of "x".

```r
FUN <- function(x) { round(sqrt(x), 4) }
```

Functions in BiocParallel use the registered back-ends for parallel evaluation. The default is the top entry of the registry list.
registered()

## $MulticoreParam
## class: MulticoreParam
## bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bptreshold: INFO; bpstop0nError: TRUE
## bpttimeout: 2592000; bpprogressbar: FALSE
## bpRNGseed:
## bplogdir: NA
## bpresultdir: NA
## cluster type: FORK

## $SnowParam
## class: SnowParam
## bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bptreshold: INFO; bpstop0nError: TRUE
## bpttimeout: 2592000; bpprogressbar: FALSE
## bpRNGseed:
## bplogdir: NA
## bpresultdir: NA
## cluster type: SOCK

## $SerialParam
## class: SerialParam
## bpisup: TRUE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bptreshold: INFO; bpstop0nError: TRUE
## bpttimeout: 2592000; bpprogressbar: FALSE
## bplogdir: NA

Configure your R session to always use a particular back-end configure by setting options named after the back ends in an .Rprofile file, e.g.,

```r
options(MulticoreParam=quote(MulticoreParam(workers=4)))
```

When a `BiocParallel` function is invoked with no `BPPARAM` argument the default back-end is used.

`bplapply(1:4, FUN)`

Environment specific back-ends can be defined for any of the registry entries. This example uses a 2-worker SOCK cluster.

```r
param <- SnowParam(workers = 2, type = "SOCK")
bplapply(1:4, FUN, BPPARAM = param)
```

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

[[2]]
[1] 1.4142

[[3]]
[1] 1.7321
```
3 The BiocParallel Interface

3.1 Classes

3.1.1 BiocParallelParam

BiocParallelParam instances configure different parallel evaluation environments. Creating or registering a 'Param' allows the same code to be used in different parallel environments without a code re-write.Params listed are supported on all of Unix, Mac and Windows except MulticoreParam which is Unix and Mac only.

- SerialParam:
  Supported on all platforms.
  Evaluate BiocParallel-enabled code with parallel evaluation disabled. This approach is useful when writing new scripts and trying to debug code.

- MulticoreParam:
  Supported on Unix and Mac. On Windows, MulticoreParam dispatches to Serial Param.
  Evaluate BiocParallel-enabled code using multiple cores on a single computer. When available, this is the most efficient and least troublesome way to parallelize code. Windows does not support multi-core evaluation (the MulticoreParam object can be used, but evaluation is serial). On other operating systems, the default number of workers equals the value of the global option mc.cores (e.g., getOption("mc.cores")) or, if that is not set, the number of cores returned by parallel::detectCores() - 2.
  MulticoreParam uses 'forked' processes with 'copy-on-change' semantics – memory is only copied when it is changed. This makes it very efficient to invoke compared to other back-ends. A subtle cost, though, is that R's garbage collector runs periodically, and 'marks' memory as in use. This effectively triggers a copy of the marked memory. R's generational garbage collector is triggered at difficult-to-predict times; the effect in a long-running forked process is that the memory is eventually copied. See this post for additional details.
  Based on facilities originally implemented in the multicore package and subsequently the parallel package in base R.

- SnowParam:
  Supported on all platforms.
Introduction to \textit{BiocParallel}

Evaluate \textit{BiocParallel}-enabled code across several distinct \textit{R} instances, on one or several computers. This is a straightforward approach for executing parallel code on one or several computers, and is based on facilities originally implemented in the \textit{snow} package. Different types of \textit{snow} ‘back-ends’ are supported, including socket and MPI clusters.

- \textbf{BatchJobsParam}
  
  Applicable to clusters with formal schedulers.

  Evaluate \textit{BiocParallel}-enabled code by submitting to a cluster scheduler like SGE.

- \textbf{DoparParam}

  Supported on all platforms.

  Register a parallel back-end supported by the \textit{foreach} package for use with \textit{BiocParallel}.

The simplest illustration of creating \textit{BiocParallelParam} is

```
serialParam <- SerialParam()
serialParam
## class: SerialParam
## bpisup: TRUE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bpnlogdir: NA
```

Most parameters have additional arguments influencing behavior, e.g., specifying the number of ‘cores’ to use when creating a \textit{MulticoreParam} instance

```
multicoreParam <- MulticoreParam(workers = 8)
multicoreParam
## class: MulticoreParam
## bpisup: FALSE; bpnworkers: 8; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bpRNGseed: NA
## bpnlogdir: NA
## bpresultdir: NA
## cluster type: FORK
```

Arguments are described on the corresponding help page, e.g., \texttt{?MulticoreParam}.

\subsection*{3.1.2 register()ing BiocParallelParam instances}

The list of registered \textit{BiocParallelParam} instances represents the user’s preferences for different types of back-ends. Individual algorithms may specify a preferred back-end, and different back-ends maybe chosen when parallel evaluation is nested.

The registry behaves like a ‘stack’ in that the last entry registered is added to the top of the list and becomes the “next used” (i.e., the default).

\texttt{registered} invoked with no arguments lists all back-ends.
registered()

```r
## $MulticoreParam
## class: MulticoreParam
## bpin Sup: FALSE; bpnWorkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bprNGseed:
## bplogdir: NA
## bpregresultdir: NA
## cluster type: FORK
##
## $SnowParam
## class: SnowParam
## bpin Sup: FALSE; bpnWorkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bprNGseed:
## bplogdir: NA
## bpregresultdir: NA
## cluster type: SOCK
##
## $SerialParam
## class: SerialParam
## bpin Sup: TRUE; bpnWorkers: 1; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bplogdir: NA
```

 bpparam() returns the default from the top of the list.

```r
bpparam()
```

```r
## $MulticoreParam
## class: MulticoreParam
## bpin Sup: FALSE; bpnWorkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bprNGseed:
## bplogdir: NA
## bpregresultdir: NA
## cluster type: FORK
```

Add a specialized instance with `register`. When `default` is TRUE, the new instance becomes the default.

```r
default <- registered()
register(BatchJobsParam(workers = 10), default = TRUE)
```

BatchJobsParam has been moved to the top of the list and is now the default.

```r
names(registered())
```

```r
## [1] "BatchJobsParam" "MulticoreParam" "SnowParam" "SerialParam"
```
### Introduction to BiocParallel

```r
bpparam()
## class: BatchJobsParam
## bpiosup: TRUE; bpnworkers: 10; bptasks: 0; bpjobname: BPJOB
## bpiosup: FALSE; bpthreshold: INFO; bpiosup: TRUE
## bpiosup: 2592000; bpiosup: TRUE
## cleanup: TRUE

Restore the original registry
```

```r
for (param in rev(default))
  register(param)
```

#### 3.2 Functions

##### 3.2.1 Parallel looping, vectorized and aggregate operations

These are used in common functions, implemented as much as possible for all back-ends. The functions (see the help pages, e.g., `?bplapply` for a full definition) include

- **bplapply(X, FUN, ...):**
  Apply in parallel a function `FUN` to each element of `X`. `bplapply` invokes `FUN length(X)` times, each time with a single element of `X`.

- **bpmapply(FUN, ...):**
  Apply in parallel a function `FUN` to the first, second, etc., elements of each argument in `...`.

- **bpiterate(ITER, FUN, ...):**
  Apply in parallel a function `FUN` to the output of function `ITER`. Data chunks are returned by `ITER` and distributed to parallel workers along with `FUN`. Intended for iteration though an undefined number of data chunks (i.e., records in a file).

- **bpvec(X, FUN, ...):**
  Apply in parallel a function `FUN` to subsets of `X`. `bpvec` invokes function `FUN` as many times as there are cores or cluster nodes, with `FUN receiving a subset (typically more than 1 element, in contrast to `bplapply`) of `X`.

- **bpaggregate(x, data, FUN, ...):**
  Use the formula in `x` to aggregate `data` using `FUN`.

##### 3.2.2 Parallel evaluation environment

These functions query and control the state of the parallel evaluation environment.

- **bpisup(x):** Query a BiocParallelParam back-end `x` for its status.

- **bpworkers, bpnworkers:** Query a BiocParallelParam back-end for the number of workers available for parallel evaluation.
**Introduction to BiocParallel**

- **bptasks**: Divides a job (e.g., single call to *lapply* function) into tasks. Applicable to MulticoreParam only. DoparParam and BatchJobsParam have their own approach to dividing a job among workers.

- **bpstart(x)**: Start a parallel back end specified by BiocParallelParam x, if possible.

- **bpstop(x)**: Stop a parallel back end specified by BiocParallelParam x.

### 3.2.3 Error handling and logging

Logging and advanced error recovery is available in BiocParallel 1.1.25 and later. For more details see the vignette titled "Error Handling and Logging":

```r
browseVignettes("BiocParallel")
```

### 3.2.4 Locks and counters

Inter-process (i.e., single machine) locks and counters are supported using `ipclock()`, `ipcyield()`, and friends. Use these to synchronize computation, e.g., allowing only a single process to write to a file at a time.

### 4 Use cases

Sample data are BAM files from a transcription profiling experiment available in the `RNAseqData.HNRNPC.bam.chr14` package.

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

#### 4.1 Single machine

##### 4.1.1 Multi-core

There are substantial benefits, such as shared memory, to be had using multiple cores on a single machine. On a single non-Windows machine the recommended approach is multi-core, or forked processes.

This example counts overlaps between BAM files and a defined set of ranges. First create a GRanges with regions of interest (in practice this could be large).

```r
library(GenomicAlignments) ## for GenomicRanges and readGAlignments()
gr <- GRanges("chr14", IRanges((1000:3999)*5000, width=1000))
```

A `ScanBamParam` defines regions to extract from the files.

```r
param <- ScanBamParam(which=range(gr))
```

FUN counts overlaps between the ranges in `gr` and the files.
Introduction to *BiocParallel*

```r
FUN <- function(fl, param) {
    gal <- readGAlignments(fl, param = param)
    sum(countOverlaps(gr, gal))
}
```

All parameters necessary for running a job in a multi-core environment are specified in the `MulticoreParam` instance.

```r
MulticoreParam()
## class: MulticoreParam
## bpisup: FALSE; bpnworkers: 4; bptasks: 0; bpjobname: BPJOB
## bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
## bptimeout: 2592000; bpprogressbar: FALSE
## bpRNGseed:
## bplogdir: NA
## bpresultdir: NA
## cluster type: FORK
```

The *BiocParallel* functions, such as `bplapply`, use information in the `MulticoreParam` to set up the appropriate back-end and pass relevant arguments to low-level functions.

```r
> bplapply(fls[1:3], FUN, BPPARAM = MulticoreParam(), param = param)
$ERR127306
[1] 1185

$ERR127307
[1] 1123

$ERR127308
[1] 1241
```

Shared memory environments eliminate the need to pass large data between workers or load common packages. Note that in this code the GRanges data was not passed to all workers in `bplapply` and `FUN` did not need to load `GenomicAlignments` for access to the `readGAlignments` function.

### 4.1.2 Clusters

Both Windows and non-Windows machines can use the cluster approach to spawn processes. *BiocParallel* back-end choices for clusters on a single machine are `SnowParam` for configuring a Snow cluster or the `DoparParam` for use with the `foreach` package.

To re-run the counting example, `FUN` needs to modified such that ‘gr’ is passed as a formal argument and required libraries are loaded on each worker. (In general, this is not necessary for functions defined in a package name space, see Section 6.)

```r
FUN <- function(fl, param, gr) {
    suppressPackageStartupMessages({
        library(GenomicAlignments)
    })
    gal <- readGAlignments(fl, param = param)
    sum(countOverlaps(gr, gal))
}
Define a 2-worker SOCK Snow cluster.

```r
snow <- SnowParam(workers = 2, type = "SOCK")
```

A call to `bplapply` with the `SnowParam` creates the cluster and distributes the work.

```r
bplapply(fls[1:3], FUN, BPPARAM = snow, param = param, gr = gr)
```

The FUN written for the cluster adds some overhead due to the passing of the GRanges and the loading of `GenomicAlignments` on each worker. This approach, however, has the advantage that it works on most platforms and does not require a coding change when switching between windows and non-windows machines.

## 4.2 Ad hoc cluster of multiple machines

We use the term ad hoc cluster to define a group of machines that can communicate with each other and to which the user has password-less log-in access. This example uses a group of compute machines ("the rhinos") on the FHCRC network.

### 4.2.1 Sockets

On Linux and Mac OS X, a socket cluster is created across machines by supplying machine names as the `workers` argument to a `BiocParallelParam` instance instead of a number. Each name represents an R process; repeat names indicate multiple workers on the same machine.

Create a `SnowParam` with 2 cpus from 'rhino01' and 1 from 'rhino02'.

```r
hosts <- c("rhino01", "rhino01", "rhino02")
param <- SnowParam(workers = hosts, type = "SOCK")
```

Execute FUN 4 times across the workers.

```r
> FUN <- function(i) system("hostname", intern=TRUE)
> bplapply(1:4, FUN, BPPARAM = param)
```

```r
[[1]]
[1] "rhino01"

[[2]]
[1] "rhino01"

[[3]]
```
Introduction to BiocParallel

When creating a cluster across Windows machines workers must be IP addresses (e.g., "140.107.218.57") instead of machine names.

4.2.2 MPI

An MPI cluster across machines is created with `mpirun` or `mpiexec` from the command line or a script. A list of machine names provided as the -hostfile argument defines the mpi universe.

The hostfile requests 2 processors on 3 different machines.

```
rhino01 slots=2
rhino02 slots=2
rhino03 slots=2
```

From the command line, start a single interactive R process on the current machine.

```
mpiexec --np 1 --hostfile hostfile R --vanilla
```

Load BiocParallel and create an MPI Snow cluster. The number of workers in SnowParam should match the number of slots requested in the hostfile. Using a smaller number of workers uses a subset of the slots.

```
> library(BiocParallel)
> param <- SnowParam(workers = 6, type = "MPI")
```

Execute FUN 6 times across the workers.

```
> FUN <- function(i) system("hostname", intern=TRUE)
> bplapply(1:6, FUN, BPPARAM = param)
```

6 slaves are spawned successfully. 0 failed.

Batch jobs can be launched with mpiexec and R CMD BATCH. Code to be executed is in 'Rcode.R'.
4.3 Clusters with schedulers

Computer clusters are far from standardized, so the following may require significant adaptation; it is written from experience here at FHCRC, where we have a large cluster managed via SLURM. Nodes on the cluster have shared disks and common system images, minimizing complexity about making data resources available to individual nodes. There are two simple models for use of the cluster, Cluster-centric and R-centric.

4.3.1 Cluster-centric

The idea is to use cluster management software to allocate resources, and then arrange for an R script to be evaluated in the context of allocated resources. NOTE: Depending on your cluster configuration it may be necessary to add a line to the template file instructing workers to use the version of R on the master / head node. Otherwise the default R on the worker nodes will be used.

For SLURM, we might request space for 4 tasks (with `salloc` or `sbatch`), arrange to start the MPI environment (with `orterun`) and on a single node in that universe run an R script `BiocParallel-MPI.R`. The command is

```
$ salloc -N 4 orterun -n 1 R -f BiocParallel-MPI.R
```

The R script might do the following, using MPI for parallel evaluation. Start by loading necessary packages and defining FUN work to be done

```r
library(BiocParallel)
library(Rmpi)
FUN <- function(i) system("hostname", intern=TRUE)
```

Create a `SnowParam` instance with the number of nodes equal to the size of the MPI universe minus 1 (let one node dispatch jobs to workers), and register this instance as the default

```r
param <- SnowParam(mpi.universe.size() - 1, "MPI")
register(param)
```

Evaluate the work in parallel, process the results, clean up, and quit

```r
xx <- bplapply(1:100, FUN)
table(unlist(xx))
mpi.quit()
```

The entire session is as follows:

```
$ salloc -N 4 orterun -n 1 R --vanilla -f BiocParallel-MPI.R
salloc: Job is in held state, pending scheduler release
salloc: Pending job allocation 6762292
salloc: job 6762292 queued and waiting for resources
salloc: job 6762292 has been allocated resources
salloc: Granted job allocation 6762292
## ...
```
Introduction to **BiocParallel**

```r
> FUN <- function(i) system("hostname", intern=TRUE)
> library(BiocParallel)
> library(Rmpi)
> param <- SnowParam(mpi.universe.size() - 1, "MPI")
> register(param)
> xx <- bplapply(1:100, FUN)
> 4 slaves are spawned successfully. 0 failed.
> table(unlist(xx))
gizmof13 gizmof71 gizmof86 gizmof88
   25   25   25   25
> mpi.quit()
salloc: Relinquishing job allocation 6762292
salloc: Job allocation 6762292 has been revoked.
```

One advantage of this approach is that the responsibility for managing the cluster lies firmly with the cluster management software – if one wants more nodes, or needs special resources, then adjust parameters to salloc (or sbatch).

Notice that workers are spawned within the `bplapply` function; it might often make sense to more explicitly manage workers with `bpstart` and `bpstop`, e.g.,

```r
param <- bpstart(SnowParam(mpi.universe.size() - 1, "MPI"))
register(param)
xx <- bplapply(1:100, FUN)
bpstop(param)
mpi.quit()
```

### 4.3.2 R-centric

A more R-centric approach might start an R script on the head node, and use **BatchJobs** to submit jobs from within the R session. One way of doing this is to create a file containing a template for the job submission step, e.g., for SLURM

```bash
#!/bin/bash
#
## file: slurm.tmpl
## Job Resource Interface Definition
##
## ntasks [integer(1)]: Number of required tasks.
## ncpus [integer(1)]: Number of required cpus per task.
## walltime [integer(1)]: Walltime for this job, in minutes.
##
## 'resources' is an argument provided to BatchJobsParam()

#SBATCH --job-name=<%= job.name %>
#SBATCH --output=<%= log.file %>
#SBATCH --error=<%= log.file %>
#SBATCH --ntasks=<%= resources$ntasks %>
```
Introduction to \textit{BiocParallel}

```r
#SBATCH --cpus-per-task=<\%= resources$ncpus \%>
#SBATCH --time=0:10:0

## Run R: we merge R output with stdout from SLURM, which gets then
## logged via --output option
R CMD BATCH --no-save --no-restore "<\%= rscript \%>" /dev/stdout
```

The \emph{R} script, run interactively or from the command line, might then look like

```r
## define work to be done
FUN <- function(i) system("hostname", intern=TRUE)

library(BiocParallel)
library(BatchJobs)

## register SLURM cluster instructions from the template file
funs <- makeClusterFunctionsSLURM("slurm.tmpl")
param <- BatchJobsParam(4, resources=list(ncpus=1),
                        cluster.functions=funs)

register(param)

## do work
xx <- bplapply(1:100, FUN)

table(unlist(xx))
```

The code runs on the head node until \emph{bplapply}, where the \emph{R} script interacts with the SLURM scheduler to request a SLURM allocation, run jobs, and retrieve results. The argument 4 to \texttt{BatchJobsParam} specifies the number of workers to request from the scheduler; \texttt{bplapply} divides the 100 jobs among the 4 workers. If \texttt{BatchJobsParam} had been created without specifying any workers, then 100 jobs implied by the argument to \texttt{bplapply} would be associated with 100 tasks submitted to the scheduler.

Because cluster tasks are running in independent \emph{R} instances, and often on physically separate machines, a convenient ‘best practice’ is to write \texttt{FUN} in a ‘functional programming’ manner, such that all data required for the function is passed in as arguments or (for large data) loaded implicitly or explicitly (e.g., via an \emph{R} library) from disk.

### 4.4 \textit{Bioconductor} Amazon Machine Image (AMI)

An AMI for running \textit{Bioconductor} in the Elastic Compute Cloud (EC2) is available at http://www.bioconductor.org/help/bioconductor-cloud-ami/. The documentation provides detailed instructions for setting up accounts, launching instances and starting sessions via RStudio or SSH.

Motivation for running jobs in the cloud may be access to additional CPUs and / or memory. Amazon Web Services (AWS) has a variety of instances ranging from general purpose to those optimized for compute, memory or I/O intensive jobs. See http://aws.amazon.com/ec2/instance-types/ for a full listing of instances.

Single or multiple instances can be requested when starting up the AMI. When a single instance is requested, interaction with the resource is the same as described in the ‘Single Machine’ section above. Requesting multiple instances is essentially creating a cluster.
do this with the Bioconductor AMI the StarCluster toolkit must be installed and a config file must be modified to specify the number and type of instances. Walk-through examples are provided in the Bioconductor AMI web documentation http://www.bioconductor.org/help/bioconductor-cloud-ami/.

This example uses a cluster of three m3.large instances each of which has 2 virtual CPUs. The name of the cluster in the config file is 'smallcluster' and can be started with the 'start' command:

```
    starcluster start smallcluster
```

List the cluster nodes:

```
~ >starcluster listclusters
```

---

Launch time: 2014-07-21 08:44:11
Uptime: 0 days, 00:09:30
Zone: us-east-1b
Keypair: bioc-keypair-vobencha
EBS volumes: N/A
Cluster nodes:

```
    smallcluster-master running i-1f757234 ec2-54-91-126-83.compute-1.amazonaws.com
    smallcluster-node001 running i-1e757235 ec2-50-16-135-207.compute-1.amazonaws.com
    smallcluster-node002 running i-19757232 ec2-54-89-112-24.compute-1.amazonaws.com
```

Total nodes: 3

Use the hostname for the master node to connect to the cluster via RStudio or SSH.

The AMI cluster workers can communicate via SSH, SunGrid Engine (SGE) or MPI. For MPI, a SnowParam should be configured with `type = MPI`.

```r
> library(BiocParallel)
> param <- SnowParam(workers=3, type = "MPI")
```

Both SSH and SunGrid Engine require a BatchJobsParam but differ slightly in the set-up.

For SSH, the nodes are registered as SSH workers. The 'workers' argument to BatchJobsParam should be the number of nodes and 'ncpus' the number of processors per node.

```r
> library(BiocParallel)
> library(BatchJobs)
> funs <- makeClusterFunctionsSSH(  +   makeSSHWorker(nodename="smallcluster-master"),
+   makeSSHWorker(nodename="smallcluster-node001"),
+   makeSSHWorker(nodename="smallcluster-node002"))
> )
> param <- BatchJobsParam(workers=3,
+    resources=list(ncpus=2),
+    cluster.functions=funs)
```

SunGrid Engine only needs the number of 'workers' and 'ncpus.'
> library(BiocParallel)
> library(BatchJobs)
> param <- BatchJobsParam(3, resources=list(ncpus=2))

This example uses 8 paired-end RNA-Seq BAM files, 1 per sequencing run, subset on chromosome 14 only.

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

Reads with more than 2 gaps in the CIGAR are isolated and `locateVariants` is used to identify where these multi-gap reads fall with respect to the UCSC hg19 known gene model.

>`FUN <- function(file, ...) {
+   library(GenomicAlignments)
+   library(VariantAnnotation)
+   library(TxDB.Hsapiens.UCSC.hg19.knownGene)
+   txdb <- TXDb.Hsapiens.UCSC.hg19.knownGene
+   gal <- readGAlignmentPairs(file)
+   gal_with_gaps <- gal[njunc(gal) > 2L]
+   locateVariants(granges(gal_with_gaps), txdb, AllVariants())
> }

Define a `BatchJobsParam` with 3 workers, 2 processors each.

>`param <- BatchJobsParam(3, resources=list(ncpus=2))

Execute in parallel over the 8 files.

>`res <- bplapply(fls, FUN, BPPARAM=param)

The return value is a list of GRanges, one for each file. The length of each GRanges will vary.

>`lengths(res)
ERR127306   ERR127307    ERR127308    ERR127309    ERR127302    ERR127303    ERR127304    ERR127305
   5228      5202      5721       4326       4494       4909       4636       4538

Summarize the LOCATION and TXID output of `locateVariants` with `xtabs`.

>`mdat <- lapply(res, function(x) mcols(x)[c("LOCATION", "TXID")])
>`xtab <- xtabs(~ LOCATION + TXID, do.call(rbind, mdat))

The xtab object can be subset to isolate transcripts that meet a criteria. These are the top 5 transcripts hit by the multi-gap reads.

>`xtab[,xtab["spliceSite", ] > 500]

TXID
LOCATION 51581 51582 53506 53507 53508
spliceSite 994 994 520 548 548
intron 0 0 0 0 0
fiveUTR 0 0 0 0 0
threeUTR 0 0 0 0 0
coding 0 0 0 0 0
intergenic 0 0 0 0 0
promoter 0 0 0 0 0

Extract the transcripts with multi-gap reads in promoter regions:
Introduction to **BiocParallel**

```r
> xtab[, xtab["promoter", ] > 0]
```

<table>
<thead>
<tr>
<th>TXID</th>
<th>LOCATION</th>
<th>spliceSite</th>
<th>intron</th>
<th>fiveUTR</th>
<th>threeUTR</th>
<th>coding</th>
<th>intergenic</th>
<th>promoter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>152</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5 Analyzing genomic data in **Bioconductor**

General strategies exist for handling large genomic data that are well suited to R programs. A manuscript titled *Scalable Genomics with R and Bioconductor* ([http://arxiv.org/abs/1409.2864](http://arxiv.org/abs/1409.2864)) by Michael Lawrence and Martin Morgan, reviews several of these approaches and demonstrate implementation with **Bioconductor** packages. Problem areas include scalable processing, summarization and visualization. The techniques presented include restricting queries, compressing data, iterating, and parallel computing.

Ideas are presented in an approachable fashion within a framework of common use cases. This is a beneficial read for anyone anyone tackling genomics problems in R.

6 For developers

Developers wishing to use **BiocParallel** in their own packages should include **BiocParallel** in the DESCRIPTION file

```
Imports: BiocParallel
```

and import the functions they wish to use in the NAMESPACE file, e.g.,

```r
importFrom(BiocParallel, bplapply)
```

Then invoke the desired function in the code, e.g.,

```r
system.time(x <- bplapply(1:3, function(i) { Sys.sleep(i); i }))
```

```r
## user system elapsed
## 0.012 0.052 3.111
## [1] 1 2 3
```

This will use the back-end returned by bpparam(), by default a MulticoreParam() instance or the user’s preferred back-end if they have used register(). The MulticoreParam back-end does not require any special configuration or set-up and is therefore the safest option for developers. Unfortunately, MulticoreParam provides only serial evaluation on Windows.

Developers should document that their function uses **BiocParallel** functions on the man page, and should perhaps include in their function signature an argument `BPARAM=bpparam()`.
Developers wishing to invoke back-ends other than MulticoreParam need to take special care to ensure that required packages, data, and functions are available and loaded on the remote nodes.

In `bplapply()`, the environment of `FUN` (other than the global environment) is serialized to the workers. A consequence is that, when `FUN` is inside a package name space, other functions available in the name space are available to `FUN` on the workers.

```
7
```