Scalability in R: Parallelization and Big Data

Kylie Ariel Bemis

5 February 2020

Scalability: parallel computing in R

Motivating example: processing high-throughput MS experiments

Suppose we want to process a large number of mass spectra

```r
library(MSExample) # See "3-OOP" slides if you still need to ins

set.seed(2020) # set a seed for random number generator
spectra <- simSpectra(n=500) # simulate 100 mass spectra

spectra
```

```
## MassSpectraList of length 500
```

Motivating example: processing high-throughput MS experiments

```r
system.time({
  result <- lapply(spectra, function(s) {
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  })
})
```

```
##    user  system elapsed
##   7.249   0.346   7.876
```

Not bad, but how will this scale if we wanted to process millions of spectra?

Can we fit this in parallel?

The easiest problems to parallelize are **embarrassingly parallel**:

- Independent tasks requiring no communication between them
- Data can be split into similarly-sized, independent subsets
- Almost anything that can be done with `lapply()` or `purrr::map()`

This applies perfectly to processing millions of mass spectra!

Processing each spectrum is independent of processing any other spectra, so this is a prime candidate for paralellization.

Using BiocParallel

The BiocParallel package on Bioconductor provides a bplapply() function that implements a parallel version of lapply().

BiocParallel allows users to register() different parallel backends, including a fallback to serial evaluation.

```r
BiocManager::install("BiocParallel")
```

```r
library(BiocParallel)
```

Documentation and vignettes:
http://bioconductor.org/packages/BiocParallel/

Parallel backends

You can see the registered backends with `registered()`.

```r
registered()
```

```
## $MulticoreParam
## class: MulticoreParam
##   bpisup: FALSE; bpnworkers: 2; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: FORK
##
## $SnowParam
## class: SnowParam
##   bpisup: FALSE; bpnworkers: 2; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
##   cluster type: SOCK
##
## $SerialParam
## class: SerialParam
##   bpisup: FALSE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
```

Serial backend

```r
SerialParam()
```

`SerialParam()` provides a backend for serial evaluation.

This is useful if a user doesn't want to run code in parallel. Because there is often some amount of overhead associated with running a job in parallel, there are many situations where it may be faster to just use serial execution.

Some options, such as debugging, cannot be done in parallel and are only possible when using the serial backend.

`SerialParam()` should be available on all platforms.

SNOW backend

```
SnowParam()
```

`SnowParam()` provides a backend for simple-network-of-workstations (SNOW) style parallelism.

A SNOW cluster works by starting worker R processes (potentially on other machines) and communicating with them via socket connections.

Because the worker R processes are fresh R sessions, there is significant overhead associated with starting a SNOW cluster, and you may need to load required packages and send data to the workers.

`SnowParam()` should be available on all platforms.

Summary: SNOW backend

- Create new worker R sessions manually
- Relies on built-in R networking utilities, works on all OS's
- Works over a network cluster
- Communication between processes depends on network
- Worker processes are independent, fresh R sessions
  - Each process has an independent, separate memory space
  - Data must be moved manually to the worker processes
  - Libraries must be reloaded

Multicore backend

```r
MulticoreParam()
```

`MulticoreParam()` provides a backend for FORK clusters on supported operating systems.

A FORK cluster works by forking the original R session. This essentially creates clones of the original R session that initially share the same memory (including all loaded data and packages), until changes are made.

This is typically fast, with less overhead than SNOW, but only works on a single machine, and isn't available on Windows due its reliance on Unix forking.

`MulticoreParam()` is only availabe on Unix-alike platforms such as macOS and Linux.

Summary: Multicore backend

- Clones the original R session to create worker processes
- Requires OS forking support, only works on macOS and Linux
- Only works on a single machine, not over a network cluster
- Minimal overhead in communication between processes
- Worker processes share memory with the original R session
  - In theory, this is memory-efficient, as the data is shared and does not need to be exported or duplicated from the host R session
  - In practice, forked memory management is OS-dependent, and R's garbage collection frequently triggers duplication of objects

Other backends

`BiocParallel` supports a few other backends that we won't discuss, because they're less common, and aren't always available. A user may always `register()` them, though.

```r
DoparParam()
```

- Provides support for backends registered through the `foreach` package

```r
BatchtoolsParam()
```

- Provides backend for `batchtools` package for managing high-performance computing clusters

Registering a default backend

By default, `BiocParallel` will use the first backend on the list of `registered()` backends. A user can register a new backend with `register()`, putting it at the top of the list.

```r
register(SerialParam())
```

`bpparam()` returns the first backend, which is used as the default.

```r
bpparam()
```

```
## class: SerialParam
##   bpisup: FALSE; bpnworkers: 1; bptasks: 0; bpjobname: BPJOB
##   bplog: FALSE; bpthreshold: INFO; bpstopOnError: TRUE
##   bpRNGseed: ; bptimeout: 2592000; bpprogressbar: FALSE
##   bpexportglobals: TRUE
##   bplogdir: NA
##   bpresultdir: NA
```

Using `bplapply()`

`BiocParallel::bplapply()` is a parallelized version of `lapply()` that uses the currently-registered backend for parallel execution.

```
bplapply(X, FUN, ..., BPPARAM = bpparam())
```

- `X` is a list-like object with `length(X)`, `X[i]`, and `X[[i]]` methods
- `FUN` is a function to be applied to each element of `X` in parallel
- `...` are additional arguments passed to `FUN` (and which will be exported to the cluster workers)
- `BPPARAM` is the parallel backend to use, where `bpparam()` uses the default registered backend

`BPPARAM` can be used to override the default backend for any particular call to `bplapply()`.

Using SerialParam()

We'll start by changing `lapply()` to `bplapply()` in our code.

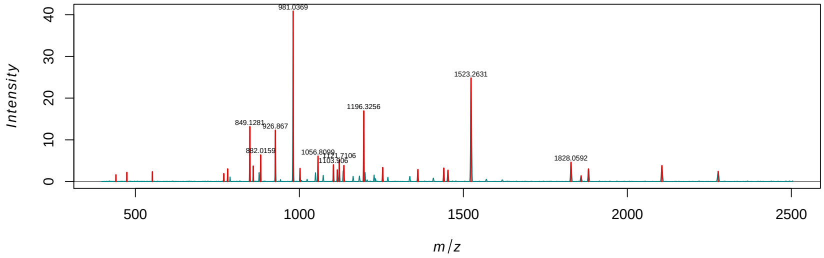
```
serial <- SerialParam()

system.time({
  result <- bplapply(spectra, function(s) {
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, BPPARAM=serial)
})
```

```
##    user  system elapsed
##  10.091   2.549  13.173
```

Note that because it must allow for the possibility of parallel execution, `bplapply()` may be slower than `lapply()` when run serially.

```r
plot(result[[1]])
```



Using SnowParam()

A SNOW backend requires starting new R sessions, so in order to be worth it, the problem should take significantly longer than the time it takes to start the workers and send the data to them.

```r
snow <- SnowParam(workers=4)

system.time({
  result <- bplapply(spectra, function(s) {
    require(MSExample)
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, BPPARAM=snow)
})
```

```
##    user  system elapsed
##   0.423   0.081   8.357
```

Using MulticoreParam()

A multicore backend forks the original R session. This is typically much faster than starting a SNOW cluster, but still entails some overhead. Only Unix-alike platforms have access to a multicore backend.

```r
mc <- MulticoreParam(workers=4)

system.time({
  result <- bplapply(spectra, function(s) {
    require(MSExample)
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, BPPARAM=mc)
})
```

```
##    user  system elapsed
##  12.088   3.045   5.481
```

Note on using `BiocParallel`

- It is recommended to always expose a `BPPARAM = bpparam()` argument in any function you use that eventually calls `BiocParallel` code
  - Allows users to specify the backend they want to use to BPPARAM
  - Using `bpparam()` as default parameter ensures the function will use the default registered backend
- No assumptions should be made about the backend that will be used. Your users may use a different parallel backend than you do

Note on using `BiocParallel`

- Make sure all packages and data you need are available on the workers
  - The arguments and environment of `FUN()` (but NOT the global environment) and all of its parameters are serialized to the workers
  - Load any required packages in your function – if you're using `BiocParallel` in your package, you shouldn't need to do this (because the environment of `FUN()` is your package)
  - If you need to make other variables available on the workers, it is easiest to do so by constructing `FUN()` to take them as parameters
- Parallelization isn't always worth it
  - In this case, the parallel backends outperform the serial backend, but `bplapply()` itself is slower than `lapply()` in the serial case
  - Parallelize stuff that will probably take a long time, and where the effort is worth it

Scalability: “big” data in R

Motivating example: processing high-throughput MS experiments

Suppose we want to process a large number of mass spectra

```r
library(MSExample) # See "3-OOP" slides if you still need to ins

set.seed(2020) # set a seed for random number generator
spectra <- simSpectra(n=500, baseline=4) # simulate 100 mass spe

spectra
```

```
## MassSpectraList of length 500
```

Motivating example: processing high-throughput MS experiments

We can use `BiocParallel:bplapply()` to do the processing in parallel.

```r
library(BiocParallel)
register(MulticoreParam())
```

```r
result <- bplapply(spectra, function(s) {
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  })
```

But what if `spectra` is very large, and can't be loaded into memory?

R, “big” data, and parallel computation

- Most R functions expect the data to be available in-memory
- R provides core functionality for working with large data
- Parallel computation compounds this memory issue
  - Worker R sessions cannot share the same R objects, so duplicates must be made
  - SNOW backends require sending copies of segments of the data to workers
  - In theory, forked multicore backends efficiently share memory with the master R session, but in practice R’s garbage collection may trigger divergent memory spaces, creating duplicates of your big data object
- Ideally, we would like to use memory-efficient data structures that can be easily shared between R sessions without needing to serialize huge amounts of data

File-based data backends on Bioconductor

- The HDF5Array and DelayedArray packages on Bioconductor provide infrastucture for HDF5-based matrices.
    - A DelayedArray wraps a “seed” object which could be an in-memory matrix or a file-based backend like HDF5.
    - Operations such as subsetting or applying functions to the matrix are “delayed” and applied only when data is “realized”
    - An HDF5Matrix is an HDF5-based matrix
- The matter package on Bioconductor provides a flexible framework for representing binary files as vectors, matrices, lists, etc., in R
    - A matter_mat matrix can be composed of any number of binary files, and indexed like an ordinary R matrix
    - Operations such as basic arithmetic are “delayed” and applied only when the data is accessed
    - Useful for providing flexible, direct access to binary file formats such as the “.ibd” portion of imzML

Using a file-based matrix in R

File-based backends typically have some additional limitations on what kind of data can be represented.

Therefore, we can't use our `MassSpectrum` objects directly.

However, since all of our spectra share the same m/z values, we can simply turn the intensities into a matrix that is easily stored.

```r
intensities <- simplify2array(lapply(spectra, intensity))
dim(intensities)
```

```
## [1] 4608  500
```

Using HDF5Array

```r
library(HDF5Array)

intensity_h5 <- writeHDF5Array(intensities,
                               filepath="intensities.h5",
                               name="intensity")
intensity_h5
```

```
## <4608 x 500> matrix of class HDF5Matrix and type "double":
##                [,1]       [,2]       [,3] ...     [,499]      [
##    [1,]   3.818100   3.851453   3.847049   .   4.005328   4.0
##    [2,]   3.971078   4.032019   3.968237   .   3.897515   3.9
##    [3,]   3.855013   4.011285   3.898457   .   3.960060   4.0
##    [4,]   4.022535   4.020253   3.955200   .   4.017306   3.9
##    [5,]   4.003855   3.977321   4.042420   .   3.992513   4.2
##     ...          .          .          .   .          .
## [4604,] 0.00000000 0.00000000 0.08562387   . 0.10242031 0.022
## [4605,] 0.11993057 0.07900605 0.00000000   . 0.00000000 0.000
## [4606,] 0.00000000 0.00000000 0.08516333   . 0.08747652 0.015
## [4607,] 0.00000000 0.00000000 0.03232444   . 0.00000000 0.000
## [4608,] 0.31258571 0.02145411 0.00000000   . 0.22324191 0.000
```

Using HDF5Array with bplapply()

We can access a column of an HDF5Matrix like we would an ordinary matrix.

```r
result <- bplapply(1:ncol(intensity_h5), function(i, ints, mz)
  {
    require(MSExample)
    require(HDF5Array)
    s <- MassSpectrum(mz, intensity=ints[,i])
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, ints=intensity_h5, mz=mz(spectra[[1]]))
```

While the file-based I/O is slower, this approach is highly memory-efficient.

Using matter

```r
library(matter)

intensity_mat <- as.matter(intensities)
intensity_mat
```

```
## <4608 row, 500 column> matter_matc :: out-of-memory numeric m
##                    [,1]             [,2]             [,3]
## [1,]  3.8180997954235 3.85145278544294 3.84704868749508  3.99
## [2,]  3.9710779099342 4.03201884360081 3.96823705620642 4.076
## [3,] 3.85501251800813 4.01128510676636 3.89845688283532 4.027
## [4,] 4.02253518181121 4.02025331358464 3.95520007966507 3.827
## [5,] 4.00385477930748 3.97732118413487 4.04242037663899 4.001
## [6,] 3.87566747639985 4.14400858903673 3.93607504862066 3.896
## ...                 ...              ...              ...
##                    [,5]             [,6] ...
## [1,]  3.8611785215622 3.98989680146283 ...
## [2,]  3.8437552523714 3.91370526952774 ...
## [3,]  4.1087806556719 3.92956581989088 ...
## [4,] 3.89122138272723 3.89916691758026 ...
## [5,]  3.9728111050763 4.00226140199709 ...
```

Using matter with bplapply()

We can access a column of a matter_mat like we would an ordinary matrix.

```
result <- bplapply(1:ncol(intensity_mat), function(i, ints, mz)
  {
    require(MSExample)
    require(matter)
    s <- MassSpectrum(mz, intensity=ints[,i])
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, ints=intensity_mat, mz=mz(spectra[[1]]))
```

Again, the file-based I/O is slower than if the data were in memory, but this approach is highly memory-efficient, and may be the only way to acess larger-than-memory data.

Using matter for non-matrix data

What if we couldn't represent our spectra as a matrix? For example, what if each spectrum had different m/z values?

```
set.seed(2019)
spectra2 <- replicate(100, simSpectra())
head(lengths(spectra2)) # each spectrum is a different length
```

```
## [1] 4177 3350 3697 4230 3967 4058
```

```
head(mz(spectra2[[1]])) # m/z of first spectrum
```

```
## [1] 500.5322 500.7324 500.9328 501.1332 501.3337 501.5343
```

```
head(mz(spectra2[[2]])) # m/z of second spectrum
```

```
## [1] 495.1546 495.3527 495.5509 495.7491 495.9475 496.1459
```

One advantage of `matter` is that it was designed to be flexible for such situations, and has certain capabilities that were designed specifically with mass spectral data in mind.

List in matter

We can create file-based lists of m/z values and intensities.

```
mz2 <- as.matter(lapply(spectra2, mz))
intensity2 <- as.matter(lapply(spectra2, intensity))

mz2
```

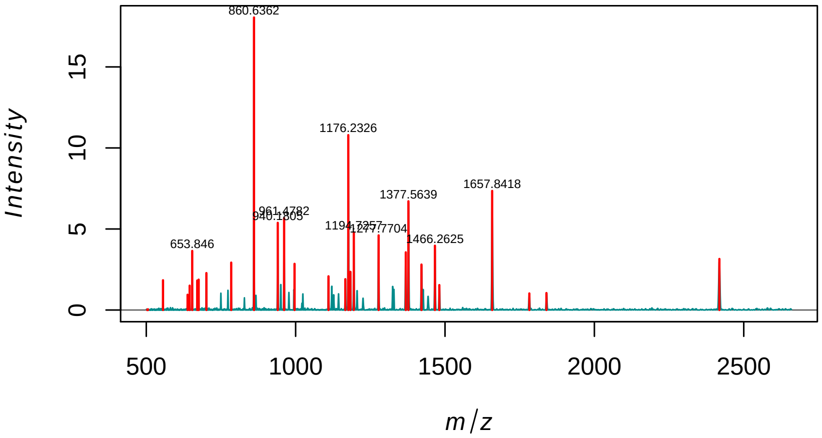
```
## <100 length> matter_list :: out-of-memory list
##                   [1]              [2]              [3]
## [[1]] 500.53219276886 500.732445696553 500.932778741441 501.13319193
##                    [5]              [6] ...
## [[1]] 501.333685311025 501.534258899867 ...
##                    [1]              [2]              [3]
## [[2]] 495.154606532526 495.352707995432 495.550888714774 495.7491487
##                    [5]              [6] ...
## [[2]] 495.947488049615 496.145906728571 ...
##                    [1]              [2]              [3]
## [[3]] 452.936710715115 453.117921641587 453.29205066928 453.4805610
##                    [5]              [6] ...
## [[3]] 453.661989530256 453.843490626287 ...
##                    [1]              [2]              [3]
## [[4]] 418.063706372359 418.230965306695 418.398291157988 418.5656839
##                    [5]              [6] ...
```

Using matter lists with bplapply()

We can use our matter_lists of m/z values and intensities with bplapply() to process spectra of different lengths.

```r
result <- bplapply(1:length(mz2), function(i, ints, mz)
  {
    require(MSExample)
    require(matter)
    s <- MassSpectrum(mz[[i]], intensity=ints[[i]])
    s <- smoothNoise(s)
    s <- removeBaseline(s)
    findPeaks(s)
  }, ints=intensity2, mz=mz2)
```

```r
plot(result[[1]])
```



Sparse matrices with `matter`

Suppose we want to represent mass spectra of different lengths as a matrix.

There's no obvious way to do so without binning every spectra to the same m/z values.

However, `matter` provides sparse matrices with on-the-fly binning of its elements.

```r
mzr <- range(sapply(mz2, range))

keys <- seq(from=mzr[1], to=mzr[2], by=1) # m/z bins
data <- list(keys=mz2, values=intensity2) # key-values
tol <- 0.5 # half-bin width

intensities2 <- sparse_mat(data, keys=keys,
                           tolerance=tol, combiner="sum")
```

Sparse matrices with `matter` (cont'd)

Both the "keys" (m/z values) and "values" (intensities) are stored as file-based `matter_list` objects.

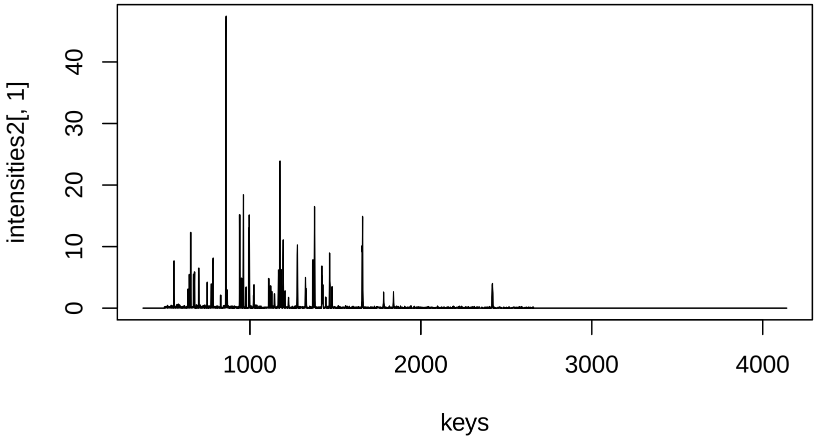
Elements of the sparse matrix are binned on-the-fly as they are accessed, according to the row `keys`, within a given `tolerance`.

```r
intensities2
```

```
## <3766 row, 100 column> sparse_matc :: sparse numeric matrix
##                          [,1] [,2] [,3] [,4] [,5] [,6] ...
## [374.844390857363,]         0    0    0    0    0    0 ...
## [375.844390857363,]         0    0    0    0    0    0 ...
## [376.844390857363,]         0    0    0    0    0    0 ...
## [377.844390857363,]         0    0    0    0    0    0 ...
## [378.844390857363,]         0    0    0    0    0    0 ...
## [379.844390857363,]         0    0    0    0    0    0 ...
## ...                       ...  ...  ...  ...  ...  ... ...
## (381266/376600 non-zero elements: 101.24% density)
## (67.1 KB real | 6.1 MB virtual)
```

```r
head(keys(intensities2)) # row m/z values
```

```r
plot(keys, intensities2[,1], type='l')
```



Summary: File-based backends in R

- `HDF5Array` on Bioconductor
  - Uses HDF5 files for data storage
  - Supports delayed block processing
  - https://bioconductor.org/packages/HDF5Array/
- `matter` on Bioconductor
  - Uses flat files for data storage
  - Supports custom formats and some delayed operations
  - https://bioconductor.org/packages/matter/