Streaming Data, Concurrency And R

Rory Winston

rory@theresearchkitchen.com

About Me

- Independent Software Consultant
- M.Sc. Applied Computing, 2000
- M.Sc. Finance, 2008
- Apache Committer
- Working in the financial sector for the last 7 years or so
- Interested in practical applications of functional languages and machine learning
- Relatively recent convert to R ($\approx$ 2 years)

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

R - Pros and Cons

Pro

- Designed by statisticians
- Can be extremely elegant
- Comprehensive extension library
- Open-source
- Huge parallelization effort
- Fantastic reporting capabilities
- Incredibly Popular

Con

- Designed by statisticians
- Can be clunky (S4)
- Bewildering array of overlapping extensions
- Inherently single-threaded
- Incredibly Popular

Parallelization vs. Concurrency

- R interpreter is single threaded
- Some historical context for this (BLAS implementations)
- Not necessarily a limitation in the general context
- Multithreading can be complex and problematic
- Instead a focus on parallelization:
  - Distributed computation: `gridR`, `nws`, `snow`
  - Multicore/multi-cpu scaling: `Rmpi`, `Romp`, `pnmath/pnmath0`
  - Interfaces to Pthreads/PBLAS/OpenMP/MPI/Globus/etc.
- Parallelization suits cpu-bound large data processing applications

Other Scalability and Performance Work

- JIT/bytecode compilation (`Ra`)
- Implicit vectorization a la Matlab (code analysis)
- Large ($\geq$ RAM) dataset handling (`bigmemory`,`ff`)
- Many incremental performance improvements (e.g. less internal copying)
- Next: GPU/massive multicore...?

What Benefit Concurrency?

- Real-time (streaming to be more precise) data analysis
- Growing Interest in using R for streaming data, not just offline analyis
- GUI toolkit integration
- Fine-grained control over independent task execution
- "I believe that explicit concurrency management tools (i.e. a threads toolkit) are what we really need in R at this point." - Luke Tierney, 2001

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

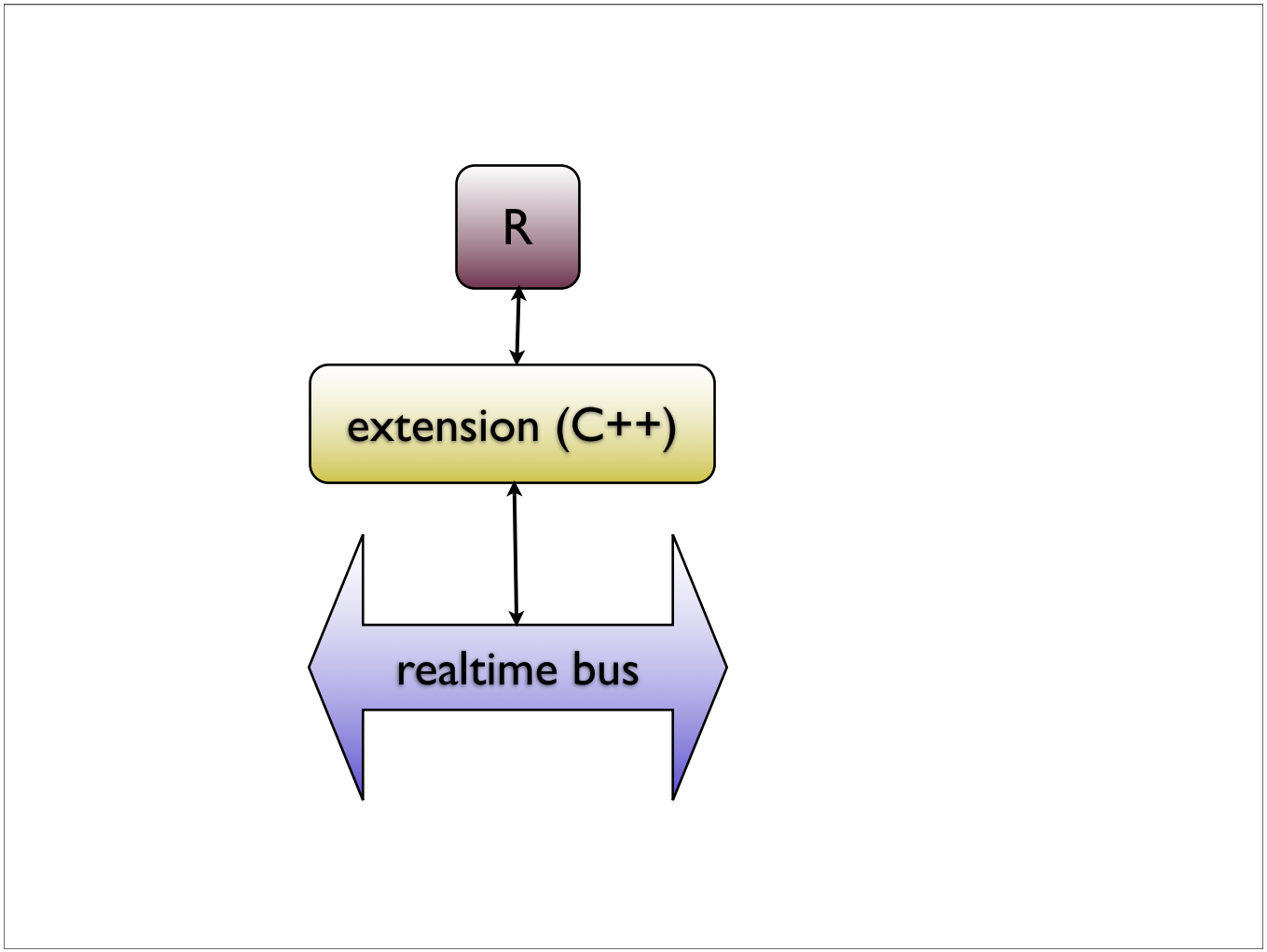
Will There Be A Multithreaded R?

- Short answer is: probably not
- At least not in its current incarnation
- Internal workings of the interpreter not particularly amenable to concurrency:
  - Functions can manipulate caller state («- vs. <-)
  - Lazy evaluation machinery (promises)
  - Dynamic State, garbage collection, etc.
  - Scoping: global environments
  - Management of resources: streams, I/O, connections, sinks
- Implications for current code
- Possibly in the next language evolution (cf. Ihaka?)
- Large amount of work (but potentially do-able)

Example Application

- Based on work I did last year and presented at UseR! 2008
- Wrote a real-time and historical market data service from Reuters/R
- The real-time interface used the Reuters C++ API
- R extension in C++ that spawned listening thread and handled updates

Simplified Architecture



Example Usage

```
rsub <- function(duration, items, callback)
```

The call `rsub` will subscribe to the specified rate(s) for the duration of time specified by `duration` (ms). When a tick arrives, the callback function `callback` is invoked, with a data frame containing the fields specified in `items`.

Multiple market data items may be subscribed to, and any combination of fields may be be specified.

Uses the underlying RFA API, which provides a C++ interface to real-time market updates.

Real-Time Example

```
# Specify field names to retrieve
fields <- c("BID","ASK","TIMCOR")

# Subscribe to EUR/USD and GBP/USD ticks
items <- list()
items[[1]] <- c("IDN_SELECTFEED", "EUR=", fields)
items[[2]] <- c("IDN_SELECTFEED", "GBP=", fields)

# Simple Callback Function
callback <- function(df) { print(paste("Received",df)) }

# Subscribe for 1 hour
ONE_HOUR <- 1000*(60)^2
rsub(ONE_HOUR, items, callback)
```

Issues With This Approach

- As R interpreter is single threaded, cannot spawn thread for callbacks
- Thus, interpreter thread is locked for the duration of subscription
- Not a great user experience
- Need to find alternative mechanism

Alternative Approach

- If we cannot run subscriber threads in-process, need to decouple
- Standard approach: add an extra layer and use some form of IPC
- For instance, we could:
  - Subscribe in a dedicated R process (A)
  - Push incoming data onto a socket
  - R process (B) reads from a listening socket
- Sockets could also be another IPC primitive, e.g. pipes
- Also note that R supports asynchronous I/O (`?isIncomplete`)
- Look at the `ibrokers` package for examples of this

The bigmemoRy package

- From the description: `"Use C++ to create, store, access, and manipulate massive matrices"`
- Allows creation of large matrices
- These matrices can be mapped to files/shared memory
- It is the shared memory functionality that we will use
- The next version (3.0) will be unveiled at UseR! 2009

```
big.matrix(nrow, ncol, type = "integer", ....)
shared.big.matrix(nrow, ncol, type = "integer", ...)
filebacked.big.matrix(nrow, ncol, type = "integer", ...)
```

Sample Usage

```
> library(bigmemory)  # Note: I'm using pre-release
> X <- shared.big.matrix(type="double", ncol=1000, nrow=1000)
> X
An object of class “big.matrix”
Slot "address":
<pointer: 0x7378a0>
```

Create Shared Memory Descriptor

```
> desc <- describe(X)
> desc
$sharedType
[1] "SharedMemory"

$sharedName
[1] "53f14925-dca1-42a8-a547-e1bccae999ce"

$nrow
[1] 1000

$ncol
[1] 1000

$rowNames
```

Export the Descriptor

In R session 1:

```
> dput(desc, file="~/matrix.desc")
```

In R session 2:

```
> library(bigmemory)
> desc <- dget("~/matrix.desc")
> X <- attach.big.matrix(desc)
```

Now R sessions A and B share the same `big.matrix` instance

Share Data Between Sessions

R session 1:

```
> X[1,1] <- 1.2345
```

R session 2:

```
> X[1,1]
[1] 1.2345
```

Thus, streaming data can be continuously fed into session A
And concurrently processed in session B

Summary

- Lack of threads not a barrier to concurrent analysis
- Packages like `bigmemory`, `nws`, etc. facilitate decoupling via IPC
- `nws` goes a step further, with a distributed workspace
- Many applications for streaming data:
  - Data collection/monitoring
  - Development of pricing/risk algorithms
  - Low-frequency execution (??)
  - ...

References

- http://cran.r-project.org/web/packages/bigmemory/
- http://www.cs.uiowa.edu/ luke/R/thrgui/
- http://www.milbo.users.sonic.net/ra/index.html
- http://www.cs.kent.ac.uk/projects/cxxr/
- http://www.theresearchkitchen.com/blog