

# Some Performance Improvements for the R Engine

Luke Tierney

Department of Statistics & Actuarial Science  
University of Iowa

May 16, 2014





- R is widely used in the field of statistics and beyond, especially in university environments.
- The S language, on which R is based, was originally developed at Bell Labs to support flexible data analysis.
- As S evolved, it was developed into a full language that also supports development of software for new methodology.
- R has become the primary framework for developing and making available new statistical methodology.
- Many (over 5,000) extension packages are available through CRAN or similar repositories.



- Many powerful features are incorporated in S and R, including
  - vectorized arithmetic
  - missing data support
  - atomic vectors (conceptually) passed by value
  - first class functions
  - lexical scope (a key addition in R)
  - lazy evaluation of arguments
- These features are valuable for specifying analysis and developing new data analysis software.
- These features also present challenges to the implementation of R.
- This talk will outline
  - some directions in which the implementation is being improved
  - some tools to help with developing good software in R



# Byte Code Compilation

## Background

- The standard R evaluation mechanism
  - parses code into a *parse tree* when the code is read
  - evaluates code by interpreting the parse trees.
- Most lower level languages (e.g. C, Fortran) compile their source code to native machine code.
- Some intermediate level languages (e.g. Java, C#) and many scripting languages (e.g. Perl, Python) compile to byte code for a virtual machine.
- Many other strategies are possible.



# Byte Code Compilation

## Background

- Byte code is the machine code for a *virtual machine*.
- Virtual machine code can then be interpreted by a simpler, more efficient interpreter.
- Virtual machines, and their machine code, are usually specific to the languages they are designed to support.
- Various strategies for further compiling byte code to native machine code are also sometimes used.



# Byte Code Compilation

## Background

- Efforts to add byte code compilation to R have been underway for some time.
- The first release of the compiler occurred with R 2.13.0.
- The current compiler and virtual machine produce good improvements in a number of cases.
- Better results should be possible with some improvements to the virtual machine and are currently being explored.



# Byte Code Compilation

## Compiler Operation

- The compiler can be called explicitly to compile single functions or files of code:
  - `cmpfun` compiles a function
  - `cmpfile` compiles a file to be loaded by `loadcmp`
- It is also possible to have package code compiled when a package is installed.
  - Use `--byte-compile` when installing or specify the `ByteCompile` option in the `DESCRIPTION` file.
  - Since R 2.14.0 R code in all base and recommended packages is compiled by default.
- Alternatively, the compiler can be used in a JIT mode where
  - functions are compiled on first use
  - loops are compiled before they are run



# Byte Code Compilation

## Compiler Operation

- The current compiler includes a number of optimizations, such as
  - constant folding
  - special instructions for most SPECIALs, many BUILTINS
  - inlining simple `.Internal` calls: e.g.  
`dnorm(y, 2, 3)`  
is replaced by  
`.Internal(dnorm(y, mean = 2, sd = 3, log = FALSE))`
  - special instructions for many `.Internals`
- The compiler is currently most effective for code used on scalar data or short vectors where interpreter overhead is large relative to actual computation.





# Byte Code Compilation

## A Simple Example

### *R Code*

```
f <- function(x) {  
  s <- 0.0  
  for (y in x)  
    s <- s + y  
  s  
}
```

### *VM Assembly Code*

```
LDCONST 0.0  
SETVAR s  
POP  
GETVAR x  
STARTFOR y L2  
L1: GETVAR s  
GETVAR y  
ADD  
SETVAR s  
POP  
STEPFOR L1  
L2: ENDFOR  
POP  
GETVAR s  
RETURN
```



# Byte Code Compilation

## Some Performance Results

Timings for some simple benchmarks on an x86\_64 Ubuntu laptop:

<i>Benchmark</i>	<i>Interp.</i>	<i>Comp.</i>	<i>Speedup</i>	<i>Exper.</i>	<i>Speedup</i>
p1	32.19	7.98	4.0	1.47	21.9
sum	6.72	1.86	3.6	0.59	11.4
conv	14.48	4.30	3.4	0.81	17.9
rem	56.82	23.68	2.4	4.77	11.9

*Interp.*, *Comp.* are for the current released version of R

*Exper.*: experimental version using

- separate instructions for vector, matrix indexing
- typed stack to avoid allocating intermediate scalar values



# Byte Code Compilation

## Future Directions

- The current virtual machine uses a stack based design.
- An alternative approach might use a register-based design.
- Some additional optimizations currently being explored:
  - avoiding the allocation of intermediate values when possible
  - more efficient variable lookup mechanisms
  - more efficient function calls
  - possibly improved handling of lazy evaluation

Some promising preliminary results are available.

- Other possible directions include
  - Partial evaluation when some arguments are constants
  - Intra-procedural optimizations and inlining
  - Declarations (sealing, scalars, types, strictness)
  - Machine code generation using LLVM or other approaches



# Reducing Value Duplication

- Conceptually, arguments are passed to functions by value, not by reference.
- This means programmers can modify their local view of an object without corrupting the original value:

```
> x <- 1
> f <- function(y) { y[1] <- 2; y }
> f(x)
[1] 2
> x
[1] 1
```

- This helps greatly in writing reliable software.



# Reducing Value Duplication

- A price is that objects often need to be duplicated, which
  - takes time
  - increases memory use
- This does not matter much for small objects, but can be prohibitive for large ones.
- Up to R 3.0.3 R used a simple mechanism to avoid duplicating:
  - if an object might be reached from more than one R variable then it is duplicated before modifying it.
- This mechanism has two drawbacks:
  - full duplication is often not necessary
  - it is too conservative



# Reducing Value Duplication

- R 3.1.0 includes changes contributed by Michael Lawrence that use *shallow duplication* in many cases.
- This only duplicates the parts of larger hierarchical objects that need to be modified.
- This significantly improves speed and memory use in particular in Bioconductor applications.



# Reducing Value Duplication

- An experiment currently underway is to replace the internal mechanism to detect when duplication might be needed by reference counting.
- This will allow duplicating objects to be avoided in many more situations.
- It may allow replacement functions like `[<-.data.frame` that are written in R to avoid duplicating in some cases
- Reference counting will also likely be easier to maintain than the current mechanism.
- This may adopted for R 3.2.0.



# Large Vector Support

- Data sets can now be quite large.
- Some categories:
  - fit into memory
  - fit on one machine's disk storage
  - require multiple machines to store
- Smaller large data sets can be handled by standard methods if enough memory is available.
- Very large data sets require specialized methods and algorithms.
- R should be able to handle smaller large data problems on machines with enough memory.





# Large Vector Support

## Initial Objectives

- The R integer data type is equivalent to C `int`.
- This is now essentially universally a signed 32-bit type.
- This type is also used for the length of a vector or total size of an array.
- This design decision made sense when R started out nearly 20 years ago:
  - most machines and operating systems were 32-bit
  - this matched the interface provided by external C/FORTRAN code



# Large Vector Support

## Initial Objectives

- This design limits the number of elements in an array to  $2^{31} - 1 = 2,147,483,647$ .
- For numeric (double precision) data this means the largest possible vector is about 16 GB.
- This is not yet a major limitation for typical users.
- It is a limitation for some users and will become more limiting over time.
- We need a way to raise this limit that meets several goals:
  - avoid having to rewrite too much of R itself
  - avoid requiring package authors to rewrite too much C code
  - avoid having existing compiled C code fail if possible
  - allow incrementally adding support for procedures where it makes sense
- For now, keep  $2^{31} - 1$  limit on matrix rows and columns.



# Large Vector Support

## Current Design

- C level changes:
  - Preserve existing memory layout
  - Use special marker in length field to identify long vectors
  - `LENGTH` accessor (returning `int`) signals an error for long vectors
  - Long vector aware code uses `XLENGTH` to return `R_xlen_t`.
- R code should not need to be changed:
  - double precision indices can be used for subsetting
  - `length` will return double for long vectors
  - `.C` and `.Fortran` will signal errors for long vectors.
- A document describing how to add long vector support to a package should be available soon.



# Large Vector Support

## Progress So Far

- A number of internal functions now support long vectors.
- Some statistical functions with long vector support:
  - random number generators
  - `mean`
  - `sort`
  - `fivenum`
  - `lm.fit`
  - `glm.fit`
- The function `dist` can handle more than  $2^{16}$  observations by returning a long vector result.
- Many matrix and array functions already support large arrays:
  - `colSums`, `colMeans`
  - `rowSums`, `rowMeans`



# Large Vector Support

## Open Issues

- Converting existing methods to support large vectors is fairly straight forward, however:
  - more numerically stable algorithms may be needed
  - faster/parallel algorithms may be needed
  - the ability to interrupt computations may become important
  - statistical usefulness may not scale to larger data
- The size where these issues become relevant is likely much lower!
- Future work will consider
  - whether to add a separate 64-bit integer type, or change the basic R integer type to 64 bits
  - possibly adding 8 and 16 bit integer types
  - arithmetic and overflow issues that these raise
  - whether to allow numbers of rows and columns in matrices to exceed  $2^{31} - 1$  as well



# Parallelizing Vector and Matrix Operations

- Most modern computers feature two or more processor cores.
- It is expected that tens of cores will be available soon.
- Two ways to take advantage of multiple cores:
  - Explicit parallelization:
    - uses some form of annotation to specify parallelism
    - packages `snow`, `multicore`, `parallel`.
  - Implicit parallelization:
    - automatic, no user action needed
- Implicit parallelization is particularly suited to
  - basic vectorized math functions
  - basic matrix operations (e.g. `colSums`)
  - linear algebra computations (threaded BLAS)



# Parallelizing Vector and Matrix Operations

## Performance Implications

- Basic idea for a  $P$ -core system:
  - run  $P$  worker threads
  - place  $1/P$  of the work on each thread
- Idealized view: this produces a  $P$ -fold speedup.
- Actual speedup is less:
  - there is synchronization overhead
  - sequential code and use of shared resources (memory, bus, ...)
  - actual workloads are uneven
- Result: parallel code can be slower!
- Parallelizing will only pay off if data size  $n$  is large enough.
  - For some functions, e.g. `qbeta`,  $n \approx 10$  may be large enough.
  - For some, e.g. `qnorm`,  $n \approx 1000$  is needed.
  - For basic arithmetic operations  $n \approx 30000$  may be needed.



# Parallelizing Vector and Matrix Operations

## Implementation Issues

- **OpenMP** provides a convenient way to implement parallelism at the **C/FORTRAN** level.
- Good performance of the synchronization barrier is critical for fine-grained parallelization.
- On Linux/**gcc** **OpenMP** performance is very good.
- On Mac OS X and Windows **gcc's** **OpenMP** barrier performance is not adequate.
- Also, Apple is moving to **clang**, which does not yet support OpenMP.
- High performance on Linux is achieved by careful use of *spin waiting*.
- We can use the same approach on Mac OS X and Windows and achieve good performance.
- Unfortunately, this means abandoning **OpenMP**.





# Parallelizing Vector and Matrix Operations

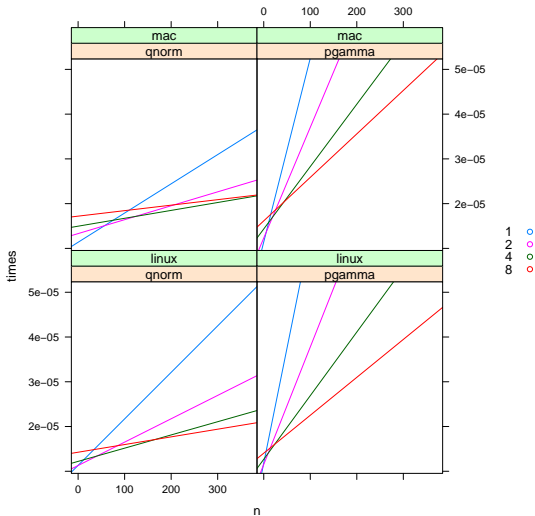
## Implementation Issues

- We are using a [pthreads](#)-based implementation using atomic integer operations for synchronization during the spin wait.
- We expect to make an interface to this framework available to package authors as well.
- Care is needed to make sure that all functions called from worker threads are thread-safe.
- Some things that are not thread-safe:
  - use of global variables
  - R memory allocation
  - signaling warnings and errors
  - user interrupt checking
  - creating internationalized messages (calls to [gettext](#))
- Random number generation is also problematic.



# Parallelizing Vectorized Operations

## Some Experimental Results





# Parallelizing Vectorized Operations

## Some Experimental Results

Some observations:

- Times are roughly linear in vector length.
- Intercepts on a given platform are roughly the same for all functions.
- Relative slopes of functions seem roughly independent of OS/architecture.

A simple calibration strategy:

- Compute relative slopes once, or average across several setups.
- For each OS/architecture combination compute the intercepts.

The appropriate time to run calibration code is still open.



# Parallelizing Vectorized Operations

## Some Notes

- An experimental package `pnmath0` that parallelizes many basic vectorized math functions is available at  
`http://www.stat.uiowa.edu/~luke/R/experimental/`
- The functions `colSums` and `dist` in the current R distribution can run in parallel but do not by default.
- Hopefully more will be included in the R distribution before too long.
- Still need to find clean way for a user to control the maximal number of threads allowed.
- Also need to resolve whether slight changes of results are acceptable, especially in reductions.



# Some Profiling Tools

- For many computations performance is not an issue.
- In cases where a computation is too slow, a first step is to identify the bottle neck.
- Profiling can be a valuable aid.
- R includes a sampling-based profiling mechanism.
- At regular intervals the functions on the call stack are recorded in a file.
- A recent addition allows the line and file information for each call to be recorded as well.
- A basic facility for examining R profile data is provided by [summaryRprof](#).
- Joint work with Riad Jarjour is developing a more extensive set of tools.



# Some Profiling Tools

- Based on examining facilities in other languages we have identified a range of filtering, summary, and visualization tools that can be useful.
- Filtering allows the programmer to, for example,
  - focus on a subset of the functions called
  - drop outer functions that are not of direct interest
  - drop functions that are only called infrequently
- Summaries include
  - function level summaries
  - call level summaries
  - source line level summaries
  - source code annotation
  - hot path identification
- Visualizations include
  - call graphs
  - time graphs
  - call tree visualizations



# Some Profiling Tools

## Examples

- Read in profile data from a linear model fit using `lm.fit`:

```
> pd <- readProfileData("Rprof-lmfit-new.out")  
> pd0 <- filterProfileData(pd, select = "system.time", focus = TRUE)
```

- Function summaries:

```
> head(funSummary(pd0), 5)
```

	total.pct	gc.pct	self.pct	gcself.pct
system.time (lmfit.R:4)	89.32	18.50	0.00	0.00
lm.fit	89.21	18.39	0.00	0.00
.Call (lmsrc.R:30)	39.65	2.97	39.65	2.97
c (lmsrc.R:64)	20.93	10.57	20.93	10.57
structure (lmsrc.R:64)	7.60	0.77	7.60	0.77



# Some Profiling Tools

## Examples

- Hot path summary:

```
> hotPaths(pd)
path                total.pct self.pct
source              99.78      0.00
. withVisible      99.78      0.00
. . eval           99.78      0.00
. . . eval         99.78      0.00
. . . . system.time 89.32      0.00
. . . . . lm.fit    89.21      0.00
. . . . . . Call    39.65     39.65
. . . . . . c       25.55     25.55
. . . . . . . structure 7.60      7.60
. . . . . . . list   7.38      7.38
. . . . . . . rep.int 4.30      4.30
. . . . . . . names<- 2.53      2.53
. . . . . . . -      2.20      2.20
. . . . . . . gc     0.11      0.11
. . . . . . . rnorm  9.25      9.25
...

```





# Some Profiling Tools

## Examples

- Source summary:

```
> srcSummary(pd0)
```

	total.pct	gctotal.pct	source
lmfit.R:4	89.32	18.50	system.time(for (i in 1:5) lm.fit(X, y))
lmsrc.R:30	39.65	2.97	z <- .Call(C_Cdqr1s, x, y, tol)
lmsrc.R:39	8.92	0.66	nmeffects <- c(dn[pivot[r1]], rep.i ...
lmsrc.R:55	2.53	0.55	names(z\$effects) <- nmeffects
lmsrc.R:58	2.20	0.66	r1 <- y - z\$residuals
lmsrc.R:64	35.90	13.55	c(z[c("coefficients", "residuals", ...

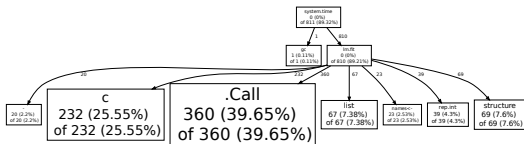
- `annotateSource` shows a full file with line annotation.



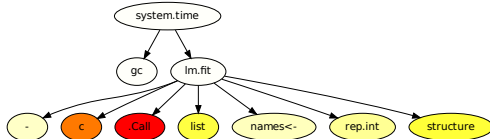
# Some Profiling Tools

## Examples

- A call graph:



- An alternative style:





# Some Profiling Tools

## Notes

- These profiling tools are a work in progress.
- They should be available in a package [proftools](#) by the end of summer.
- We are also working on a graphical interface based on [GWidgets2](#).
- This GUI should be available in a package [proftools-GUI](#) at the end of summer as well.



# Conclusions

## Synergy

- There is synergy among these areas of development; for example:
  - Many functions applied to large data are excellent candidates for parallelization.
  - The compiler may be able to fuse operations and allow more efficient parallelization at the fused operation level.
  - The compiler may also be able to compile certain uses of `sweep` and `apply` functions.
  - Profiling tools will help in refining where our implementations need to be improved.
- Exploring these opportunities will be a goal of work over the coming year.



# Conclusions

## Maintainability

- R is currently developed and maintained by statisticians for statisticians.
- More sophisticated approaches may be needed to move R forward.
- More sophisticated implementation approaches have to be balanced with maintainability.
- To be successful a novel approach needs either
  - longer term developer commitment
  - sufficient training for those with a longer term commitment
- Getting the balance right represents an interesting challenge.
- We are starting some collaborations with computer scientists that will allow us to explore these issues.